Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterthesis

Nils Ludolf Weiss

Entwurf eines Cloud-basierten Simulationsframeworks für heterogene Infrastrukturen

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Nils Ludolf Weiss

Entwurf eines Cloud-basierten Simulationsframeworks für heterogene Infrastrukturen

Masterthesis eingereicht im Rahmen der Masterprüfung
im Masterstudiengang Informations- und Kommunikationstechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Wolfgang Renz
Zweitgutachter : Prof. Dr. -Ing. Sebastian Rohjans

Abgegeben am 11. Mai 2017

Nils Ludolf Weiss

Thema der Masterthesis

Entwurf eines Cloud-basierten Simulationsframeworks für heterogene Infrastrukturen

Stichworte

Simulation, Simulationsframework, Smart-Grids, verteilte Systeme, Smart-Market, OS4ES

Kurzzusammenfassung

Das Projekt Open System for Energy Services stellt einen Smart-Market vor, an dem Energieerzeuger an einer zentralen Registry Flexibilität anbieten können. Aggregatoren durchsuchen die Registry nach passenden Produkten für ihr Portfolio und erwerben diese bei den Erzeugern. In dieser Arbeit wurde ein Simulationsframework entworfen, in dem ein solches Szenario, das den Handel im OS4ES darstellt, simuliert werden kann. Das Framework wird als verteiltes System realisiert und nutzt eine Cloud-Umgebung als Infrastruktur. Dies soll Nutzern ermöglichen, das Verhalten der Rollen innerhalb eines solchen Marktes untersuchen zu können.

Nils Ludolf Weiss

Title of the paper

Design of a cloud-based simulation framework for heterogeneous infrastructures

Keywords

simulation, simulation framework, smart-grids, distrubuted computing, smart-market, OS4ES

Abstract

The Open System for Energy Services project presents a smart-market in which energy producers are able to offer flexibility through a central registry. Aggregators search the registry for products matching their portfolio and acquire them from producers. In this thesis a proposal for a simulation framework capable of simulating such szenarios represention the OS4ES-Market is presented. The framework is designed as a distributed system and uses cloud-enviroments for infrastructure. It will enable users to research the behaviour of actors in this kind of market.

Danksagung

„There are no shortcuts to any place worth going.“

– Beverly Sills

Viele Stunden flossen in die Erstellung dieser Arbeit. Ohne die Unterstützung, der Personen denen ich hier danken möchte, wäre dies nicht möglich gewesen.

Ich bedanke mich herzlich bei meinem betreuenden Professor Dr. Wolfgang Renz und den Mitarbeitern des MMLabs Tim Dethlefs, Thomas Preisler und Ole Behncke für viele persönliche und fachliche Ratschläge. Ebenso möchte ich meinem Zweitgutachter Prof. Dr.-Ing. Sebastian Rohjans für Ratschläge sowie der Zweitprüfung meiner Arbeit danken.

Ohne meine Familie wäre diese Arbeit nie zustande gekommen. Daher gebührt ihr besonderer Dank. Ich Danke meinen Eltern und Großeltern für ihren Beistand und ihre Unterstützung. Meiner Frau und meinem Sohn für die Unterstützung und den Rückhalt, den ich in mancher Stunde benötigte, nicht nur für diese Arbeit sondern für das gesamte Masterstudium.

Inhaltsverzeichnis

Abkürzungsverzeichnis **8**

Tabellenverzeichnis **10**

Abbildungsverzeichnis **11**

1. Einleitung **13**
- 1.1. Motivation . . . 14
- 1.2. Zielsetzung . . . 15
- 1.3. Aufbau der Arbeit . . . 16

2. Grundlagen **18**
- 2.1. Energie und Energieversorgung . . . 18
  - 2.1.1. Energieversorgung . . . 19
  - 2.1.2. Frequenz-Leistungs-Regelung (Regelleistung) . . . 20
  - 2.1.3. Marktintegration von dezentralen Energieressourcen . . . 21
  - 2.1.4. Forschungsprojekt OS4ES . . . 22
- 2.2. Virtualisierung . . . 24
  - 2.2.1. Virtual Machine . . . 24
  - 2.2.2. Container . . . 25
  - 2.2.3. Docker . . . 26
  - 2.2.4. Rancher . . . 27
- 2.3. Bestehende Simulationsplattformen . . . 29
  - 2.3.1. Mosaik . . . 29
  - 2.3.2. Jadex Active Components . . . 32
  - 2.3.3. GridSpice . . . 35

3. Anforderungen und Analyse **38**
- 3.1. Systembeschreibung . . . 38
- 3.2. Anforderungen an die Plattform . . . 46
- 3.3. Anforderungen an die Kommunikation . . . 47

4. Architektur und Design des Frameworks **49**
4.1. Beschreibung der Framework-Architektur 50
4.2. Komponenten der Konfigurationsschicht 53
4.2.1. Docker-Plattform und Rancher 53
4.2.2. Front-End 54
4.2.3. Deployment-Modul 54
4.3. Komponenten der Simulationsschicht 56
4.3.1. Simulationsmanager 57
4.3.2. Kommunikationsrahmen 60
4.4. Beschreibung der Kommunikationsschnittstellen 61
4.4.1. Datenmodelle 62
4.4.2. Front-End-UI 73
4.4.3. Sim-API 77

5. Validierung und Bewertung **84**
5.1. Anforderungen 84
5.1.1. Simulationsplattform 84
5.1.2. Kommunikation 87
5.2. Abgrenzung 90
5.3. Fazit 92

6. Prototypische Implementierung des Kommunikationsrahmens **93**
6.1. Eingesetzte Werkzeuge 93
6.2. Architektur des Prototypen 94
6.2.1. RestController 96
6.2.2. Handler 98
6.2.3. Kommunikationsmodul (High-Level-API) 99
6.2.4. ApiHandler (High-Level-API) 100

7. Zusammenfassung und Ausblick **102**
7.1. Zusammenfassung 102
7.2. Ausblick 104

Literaturverzeichnis **106**

A. Weitere Nachrichtentypen **110**
A.1. Definition der Config-Response-Nachricht 110
A.2. Definition der Step-Response-Nachricht 111
A.3. Definition der Result-Request-Nachricht 112
A.4. Definition der Result-Nachricht 113
A.5. Definition der Fehlernachricht 114

B. Definition der Datenbankstruktur **115**
B.1. Szenariodatenbank 115
B.2. Ergebnisdatenbank 116

Abkürzungsverzeichnis

AP	Access Point
API	Application Programming Interface
AWS	Amazon Web Services
BDI	Belief Desire Intention
BGV	Bilanzgruppenverantwortlicher
CLI	Command Line Interface
DER	Dezentrale Energieressource
DNS	Domain Name System
DSM	Demand-Side-Management
EEG	Erneuerbare-Energien-Gesetz
EEX	European Energy Exchange
ES	Energy Service
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IKT	Informations- und Kommunikationstechnik
IP	Internet Protokoll
JSON	JavaScript Object Notation
LRE	Logical Registry Entity
OS4ES	Open System for Energy Services
PV	Photovoltaik
REST	Representational State Transfer
RIS	Registry Index Service
ÜNB	Übertragungsnetzbetreiber
URL	Uniform Resource Locator
UI	User Interface
UUID	Universally Unique Identifier

VK Virtuelles Kraftwerk

VM Virtual Machine

VNB Verteilnetzbetreiber

Tabellenverzeichnis

2.1. Auszug an der EEX gehandelter Kontrakte . . . 20
2.2. Aktivierungsschwellen der Primärregelung . . . 21

3.1. Anforderungen an die Simulationsplattform . . . 47
3.2. Kommunikationsstrecken im OS4ES . . . 48
3.3. Anforderungen an die Kommunikation . . . 48

4.1. Datenfelder der Simulatorenliste . . . 63
4.2. Datenfelder der Front-End-Konfigurationsnachricht . . . 65
4.3. Datenfelder der Deployment-Modul-Konfigurationsnachricht . . . 67
4.4. Datenfelder der Container-Konfigurationsnachricht . . . 69
4.5. Datenfelder der Step-Befehlsnachricht . . . 71
4.6. Datenfelder der IntraSim-Nachricht . . . 72

5.1. Anforderungen an die Simulationsplattform . . . 85
5.2. Anforderungen an die Kommunikation . . . 88
5.3. Validierung der Frameworkarchitektur . . . 89
5.4. Gegenüberstellung Anforderungen/verwandte Produkte . . . 90

6.1. Liste der umgesetzten URL-Pfade . . . 95
6.2. Liste der umgesetzten Port-Mappings . . . 96

A.1. Datenfelder der Config-Response-Nachricht . . . 110
A.2. Datenfelder der Step-Response-Nachricht . . . 111
A.3. Datenfelder der Result-Request-Nachricht . . . 112
A.4. Datenfelder der Result-Nachricht . . . 113
A.5. Datenfelder der Fehlernachricht . . . 114

B.1. Struktur der Szenariodatenbank, Tabelle 1 . . . 115
B.2. Struktur der Szenariodatenbank, Tabelle 2 . . . 115
B.3. Struktur der Szenariodatenbank, Tabelle 3 . . . 115
B.4. Struktur der Szenariodatenbank, Tabelle 4 . . . 116
B.5. Struktur der Ergebnisdatenbank, Tabelle 1 . . . 116

Abbildungsverzeichnis

2.1. Ansprechverhalten der Regelleistungsarten 21
2.2. Struktur des Smart Markets im OS4ES-Projekt 23
2.3. Aufbau des OS4ES 24
2.4. Infrastruktur von VM's 25
2.5. Infrastruktur von Containern 26
2.6. Architektur der Docker Umgebung 27
2.7. Architektur der Docker Engine 28
2.8. Aufbau einer Rancher Umgebung 29
2.9. Integrationsproblem von Simulationen 30
2.10. Komponenten der Mosaik-Plattform 31
2.11. Vergleich der Mosaik APIs 32
2.12. Kommunikationsablauf bei Simulationsschritt 32
2.13. Definition eines Jadex Agenten 33
2.14. Jadex Time Service Protokoll 34
2.15. Architektur des GridSpice-Frameworks 35
2.16. Aufbau eines GridSpice Simulations-Clusters 36
2.17. Gridspice: Methode zur Findung des nächsten Zeitschrittes 37

3.1. Simulierter Bereich des OS4ES 39
3.2. Architektur der Registry-Komponenten 40
3.3. Organisatorische Architektur der Registry 41
3.4. Komponenten einer PV-Anlage 42
3.5. Kommunikationswege der PV-Anlage 42
3.6. Flussdiagramm eines Aggregators 43
3.7. Struktur des Handelsplatzes 44
3.8. Sequenzdiagramm des Handels im OS4ES 45

4.1. Programmablaufplan der Simulation 50
4.2. Schichtenmodell des Simulationsframeworks 51
4.3. Architektur der Infrastruktur-Ebene 51
4.4. Architektur der Framework-Ebene 52
4.5. Funktionslogik der Front-End Komponente 55
4.6. Funktionslogik der Front-End Komponente 56

4.7. Architektur und Kommunikationswege des Simulationsmanagers 57
4.8. Diskrete Zeitschritte der Simulation 59
4.9. Stepper-Funktion des Zeitmoduls 60
4.10. Stepper-Funktion des Zeitmoduls 61
4.11. Diskrete Zeitschritte der Simulation 78
4.12. Interaktion von sendIntraSim() und handleIntraSim() 82

6.1. Spezifikation der Protoyp-Architektur 95
6.2. IntraSim-Methode des DerControllers 97
6.3. handleIS-Methode des Handlers 98
6.4. ZMQ-Controller der High-Level-API 99
6.5. Sequenzdiagramm einer Anfrage am Webservice 101

1. Einleitung

Die Bundesrepublik Deutschland befindet sich in einem Wandel der Energieversorgung. Die Bundesregierung plant den Ausstieg aus der Kernenergie bis 2022 sowie eine Reduktion des Anteils fossiler Energieträger an der Stromerzeugung bis 2035. Weiterhin soll der Anteil an erneuerbaren Energien bis zum Jahre 2035 zwischen 55 und 60 Prozent betragen (Bundesregierung (2017)).
Die Erneuerbaren Energieträger setzen sich dabei aus Wasserkraft, Biomasse sowie den wichtigsten erneuerbaren Energieträgern Wind- und Solarkraft zusammen (BMWi (2017)).
Gerade der Umstieg auf Wind- und Solarenergie bringt allerdings auch besondere Herausforderungen mit sich. Durch ihre Abhängigkeit von den Wetterverhältnissen ist die Stromproduktion über die Sonnenenergie und Windkraft unbeständiger und ist auf Vorhersagen angewiesen. Des Weiteren wird der Strom hier, im Gegensatz zu Großkraftwerken mit fossilen und nuklearen Energieträgern, durch sehr viele Anlagen mit geringerer Leistung erzeugt. Gerade Windkraftwerke werden im Norden Deutschlands an den Küsten oder auf See installiert. Dies führt dazu, dass Strom teilweise über weite Strecken übertragen werden muss, um Kunden zu erreichen. Traditionelle Kraftwerke, mit fossilen oder nuklearen Energieträgern, konnten über Deutschland verteilt angelegt werden, sodass dieser Stromtransport über weite Strecken deutlich seltener notwendig war. Produzenten erneuerbarer Energien können, aufgrund der geografisch begrenzten Standorte, häufig nicht überall platziert werden. Es entstehen zeitliche und geografische Engpässe. Daher muss der Strom in den Spitzenzeiten der Produktion, also etwa wenn die Sonne stark scheint, effizient verteilt und eventuell gespeichert werden, damit dieser in den Flautezeiten genutzt werden kann (Bundesregierung (2017)).
Aktuell wird versucht diese Problematik durch Smart-Grids, wie nachfolgend beschrieben, zu bewältigen. Durch Ausstattung der Netzteilnehmer mit Informations- und Kommunikationstechnik (IKT) zur Messung der Stromproduktion und Mitteilung der Momentanwerte an andere wird ein Stromnetz zum Smart-Grid (Bundesnetzagentur (2011)).
Mit einem solchen Ansatz erhalten Stromproduzenten, Prosumer (Producer und Consumer), Konsumenten und Dienstleister neue Fähigkeiten, mit denen die Netzauslastung und -stabilität in Zukunft automatisiert kontrolliert und gesteuert werden kann. Eine Steuerung von Großverbrauchern etwa könnte es ermöglichen den erzeugten Strom gerade zu Spitzenzeiten zu nutzen, indem planbare und zeitlich flexible Großverbraucher dann aktiviert werden, wenn viel Strom produziert wird. Diese Steuerung der Verbraucherseite wird als

Demand-Side-Management (DSM) bezeichnet (Gellings (1985)). Auf der Produzentenseite gibt es mit dem Einspeisemanagement einen Regelungsmechanismus, der bei Engpässen im Stromnetz eingesetzt wird. Mit diesem können Anlagen in drei Schaltstufen auf 60%, 30% oder 0% ihrer Leistung abgeregelt werden, um eine Überlastung der Netze zu vermeiden (Next (2016)).
Ein zusätzlicher Aspekt eines Smart-Grids ist die mögliche Marktintegration von Kleinerzeugern erneuerbarer Energien. Zukünftig sollen diese keine Festvergütung mehr für den erzeugten Strom erhalten wie es momentan durch das Erneuerbare-Energien-Gesetz (EEG) der Fall ist. Vielmehr soll der Strom direkt am Markt gehandelt werden können (Bundesregierung (2017)).
Da allerdings viele Kleinerzeuger nicht die Energievolumina bereitstellen können, die den Produkten an den Strommärken entsprechen, wird ein Dienstleister als Mittelsmann eingesetzt. Die „Aggregatoren“ genannten Dienstleister agieren an den Strommärken und bieten dort den Strom an, der in einem Virtuellen Kraftwerk (VK) erzeugt wird. Ein VK ist dabei eine Gruppe von Kleinerzeugern, die gebündelt (aggregiert) Produkte für Strommärkte bereitstellen können (Dielman und van der Velden (2003)).
Heutzutage werden Kleinerzeuger dabei fest einem VK zugeordnet. Damit entstehen statische Portfolios, die nicht an aktuelle Anforderungen angepasst werden können. In Dethlefs u. a. (2016b) wird ein Aggregationverfahren vorgestellt, bei dem das Portfolio des Aggregators dynamisch angelegt ist und speziell für bestimmte Anwendungen zusammengestellt werden kann. Im Rahmen dieses Entwurfes wird ein Marktplatz vorgestellt der es Kleinerzeugern erlaubt, den von ihnen erzeugten Strom anbieten zu können. Dieser im OS4ES-Projekt entwickelte Marktplatz basiert auf einer Registry als zentraler Vermittlungseinheit. An dieser können Energiedienstleistungen angeboten und von Aggregatoren gebucht werden. Um diesen Marktplatz weiterhin zu untersuchen und zu validieren sollen Simulationen des Handels an diesem Marktplatz durchgeführt werden.

1.1. Motivation

Simulationslösungen für Smart-Grid-Anwendungen sind in der Regel dafür ausgelegt Szenarios mit spezifischen Architekturen zu simulieren. Dabei sind diese zumeist auf die Simulation des physischen Stromnetzes und angeschlossenen Akteuren ausgerichtet. Die Komplexität der Stromnetze macht es notwendig, die Auswirkungen von Veränderungen des Stromnetzes zu simulieren, da diese nicht trivial ermittelt werden können. Durch die Einführung von Smart-Grids erhält das Stromnetz weitere Domänen, die es zu betrachten gilt. Neben dem physischen Stromnetz und dessen Be- bzw. Auslastung besitzen Akteure im Smart-Grid die Fähigkeit, über Netzwerke miteinander zu kommunizieren. So können Optimierungsmechanismen und Automatismen eingeführt werden.

Bestehende Simulationslösungen für Smart-Grid-Anwendungen bieten zumeist keine Möglichkeit, Marktplätze für Energiedienstleistungen zu simulieren. Anderson u. a. (2014) bieten mit GridSpice diese Möglichkeit, bauen diesen Handel allerdings als zusätzliche Funktion zur Simulation des Stromnetzes auf. Des Weiteren sind die von Smart-Grid-Simulationslösungen unterstützten Parameter ohne Bedeutung für einen vom physischen Stromnetz abstrahierten Handel.
Um nun Marktplätze für Energiedienstleistungen, wie sie im zum Beispiel im OS4ES-Projekt verwendet werden, zu simulieren, werden dedizierte Simulationslösungen benötigt, die Szenarios mit geringem Konfigurationsaufwand ausführen können. Es besteht die Möglichkeit solche Simulationen spezifisch zu entwickeln oder zum Beispiel eine Co-Simulation als verteiltes System aufzusetzen. Dies erfordert allerdings einen großen Konfigurationsaufwand, da die Simulationssysteme erst noch mit den benötigten Kommunikationsstrukturen versehen werden und Mechanismen zur Steuerung der Simulatoren aufgesetzt werden müssen.

Labor- und Feldtests für neue Produkte und Verfahren sind aufwändig und mit wirtschaftlichem Risiko verbunden. Daher ist es nicht möglich groß angelegte Feldtests des OS4ES durchzuführen. Eine Möglichkeit Projekte, wie das OS4ES, ohne großes Risiko zu testen und zu validieren bietet eine Simulation. Von großem Nutzen wäre also eine Simulationslösung, über die ein Test bzw. eine Validierung des OS4ES vorgenommen werden kann.
Aktuelle Entwicklungen zeigen die Möglichkeiten, Cloud-Umgebungen zur Ausführung von verteilten Anwendungen zu verwenden. Durch eine Abstraktionsschicht zwischen der Infrastruktur und den Applikationen lassen sich heterogene Infrastrukturen zu Cloud-Umgebungen zusammenschließen oder kommerzielle Cloudservices nutzen, um dies umzusetzen. Dies bietet die Möglichkeit ein Simulationsframework zu entwickeln, das die Vorteile und Möglichkeiten von Cloud-Umgebungen ausnutzt. Gerade für ein Framework mit einer verteilten Architektur eignen sich Cloud-Umgebungen, durch die Nutzung von Virtuellen Systemen für die Komponenten. Aus diesem Ansatz ergibt sich das Ziel dieser Arbeit.

1.2. Zielsetzung

Im Rahmen dieser Arbeit soll ein Simulationsframework entworfen und spezifiziert werden das dafür ausgelegt ist unter anderem den Handel mit Energiedienstleistungen am OS4ES-Marktplatz zu simulieren. Dieses Simulationsframework soll eine Cloud-Infrastruktur nutzen, auf der die Komponenten des Frameworks betrieben werden. Die Simulationen sollen dabei durch Co-Simulation von Modellen in einer verteilten Anwendung durchgeführt werden. Dazu werden die entworfenen Komponenten in ihrer Architektur und Funktionsweise sowie genutzte Kommunikationsprotokolle und Schnittstellen beschrieben.

Durch eine Abstraktionsschicht zwischen einer heterogenen Infrastruktur und dem Framework soll die Cloud-Infrastruktur umgesetzt werden. Da Modelle in verschiedenen Programmiersprachen entwickelt werden, soll das Framework in der Lage sein diese anzubinden und dafür eine Simulator-seitige API anbieten. Außerdem soll es eine Front-End-Komponente besitzen, über die Szenario-Konfigurationen vorgenommen werden können. Die Kommunikation zwischen den Simulatoren soll dezentral gestaltet werden und direkt zwischen den Simulatoren stattfinden. Für diese Kommunikation sollen Webservices genutzt werden.
Das entworfene Simulationsframework soll validiert werden, indem theoretisch überprüft wird ob es den gestellten Anforderungen entspricht.

Im nächsten Schritt, soll der Kommunikationsrahmen, eine Komponente des Simulationsframeworks, prototypisch implementiert werden. Die technische Umsetzung und der Programmablauf werden erläutert und grafisch dargestellt.

1.3. Aufbau der Arbeit

Die Arbeit ist in sieben Kapitel unterteilt. Nach der Einleitung werden im zweiten Kapitel die wichtigsten Grundlagen skizziert und bestehende Simulationslösungen im Umfeld dieser Arbeit beschrieben. Dabei wird zuerst auf die Energieversorgung in Deutschland eingegangen und die Verbindung zur Problematik der Einbindung von erneuerbaren Energien in das bestehende System hergestellt. Zusätzlich wird der Zusammenhang zur Problemstellung der Arbeit hergestellt. Ferner werden die technischen Grundlagen der Virtualisierung vom Computern beschrieben und Produkte im Umfeld dieser Arbeit beleuchtet.

Im dritten Kapitel werden die Anforderungen an das zu entwickelnde Simulationsframework beschrieben und analysiert. Die Anforderungen und deren Analyse sind in zwei Bereiche geteilt. Zum einen werden die funktionalen Anforderungen an das Framework untersucht, zum Anderen die Anforderungen an die Kommunikationsbedarfe der Komponenten innerhalb des Simulationsframeworks.
Dazu wird zunächst das benötigte System beschrieben und daraufhin analysiert, welche Anforderungen sich aus den zu simulierenden Szenarios, bzw. deren Architektur, ergeben.

Das vierte Kapitel behandelt die Spezifikation der Framework-Architektur. Es ist dabei in vier Teile unterteilt. Zunächst wird ein Überblick über die Framework-Architektur gegeben und daraufhin werden die beteiligten Komponenten eingehend erläutert, die an der Szenario-Konfiguration und Bereitstellung der Simulationskomponenten beteiligt sind. Daraufhin werden dann die Komponenten des Simulationsframeworks beschrieben, die an der Ausführung der Simulation beteiligt sind. Zuletzt werden die Datenmodelle und Kommunikationsschnittstellen spezifiziert und erläutert, die den Datenaustausch der Komponenten ermöglichen.

Im fünften Kapitel wird eine Validierung der im vierten Kapitel vorgestellten Framework-Architektur vorgenommen indem diese mit den im dritten Kapitel aufgestellten Anforderungen gegenübergestellt wird. Außerdem wird eine Abgrenzung des, in dieser Arbeit entwickelten, Simulationsframeworks gegenüber verwandten Produkten vorgenommen, indem geprüft wird, ob diese die gestellten Anforderungen erfüllen.

Das sechste Kapitel beschreibt die prototypische Implementation des Kommunikationsrahmens, einer Framework-Komponente, die den Simulatoren standardisierte Kommunikationsschnittstellen über eine leicht implementierbare API zur Verfügung stellt. Dabei werden die verwendeten Mittel sowie die umgesetzte Architektur beschrieben.

Das siebte Kapitel fasst die Methoden und Ergebnisse der Arbeit zusammen und zieht aus diesen ein Fazit. Außerdem wird ein Ausblick auf mögliche folgende Schritte und Vorhaben gegeben.

2. Grundlagen

Bevor eine Analyse der Anforderungen durchgeführt wird, die an das hier zu entwerfende Simulationsframework gestellt werden, ist es notwendig mithilfe von Grundlagen eine Basis zu schaffen, die ein fundiertes Verständnis der Methoden und Ergebnisse erlaubt. In diesem Kapitel sollen daher Grundlagen dargestellt und erläutert werden, die zum Verständnis dieser Arbeit beitragen. Dabei werden die Energie- bzw. Stromversorgung, Grundlagen zur Virtualisierung von physischen Maschinen sowie bestehende Simulationsplattformen behandelt.
Als erstes werden dazu in Abschnitt 2.1.1 die Grundlagen zum Thema Energieversorgung dargestellt. Diese sollen dazu beitragen den Hauptnutzungsfall des Simulationsframeworks, in der Simulation des Handels von kleinen Energiemengen, nachvollziehen zu können. Des Weiteren wird in diesem Abschnitt auch das OS4ES-Projekt, an dessen Rahmen diese Arbeit angelehnt ist, beschrieben.
Daraufhin werden in Abschnitt 2.2 verschiedene Virtualisierungsmethoden und Prinzipien erläutert. Auf der Basis dieser Virtualisierungsmethoden werden Entscheidungen zum Design des Simulationsframeworks getroffen. Abschnitt 2.3 stellt daraufhin aktuelle Produkte im Umfeld dieser Arbeit vor, zu welchen eine Abgrenzung des hier entworfenen Frameworks durchgeführt wird.

2.1. Energie und Energieversorgung

Da in dieser Arbeit ein Simulationsframework entworfen werden soll, dessen zentrales Szenario der Handel mit Energiedienstleistungen nach dem in Punkt 2.1.4 beschriebenen Modell ist, werden die Grundlagen der Energieversorgung und des Handels mit Energiedienstleistungen beschrieben.
Daher, dass der Smart Market ein Produkt aktueller Entwicklungen im Sektor der Energieversorgung ist, wird zunächst der Status Quo des Energiehandels in Abschnitt 2.1.1 vorgestellt. Nachfolgend werden aktuelle Entwicklungen hinsichtlich einer Anpassung der bisherigen Stromversorgung an den wachsenden Anteil von grünen[1] Stromerzeugern in Abschnitt 2.1.3

[1] Erzeugung von Strom aus erneuerbaren Energien

beschrieben, mit deren Hilfe kleineren Anbietern Zugang zum Handel gewährt wird. Bei der Erzeugung von Strom durch erneuerbare Energien werden, anders als bei regulären Kraftwerken, kleinere Mengen Energie an einem Ort produziert. Daher nennt man diese auch Dezentrale Energieressourcen (DERs). DERs können durch das Vorhandensein von Batterien, neben der Fähigkeit der Einspeisung von Energie, auch die Aufnahme dieser unterstützen.

2.1.1. Energieversorgung

In Europa wird ein Stromnetz mit einer sinusförmigen Wechselspannung betrieben. Die Netzfrequenz beträgt dabei nach IEC 60038 (2009) 50 Hz. Einspeisung und Verbrauch müssen sich im Stromnetz immer im Gleichgewicht befinden. Ein Ungleichgewicht führt bei einem Energiedefizit dazu dass mechanische Energie aus den Erzeugergeneratoren entnommen wird und die Netzfrequenz sinkt. Bei einem Überschuss würde die Netzfrequenz dementsprechend steigen. Um eine Abweichung zu verhindern werden Fahrpläne für die Erzeugung und Verbrauch erstellt. Fahrpläne sind Bedarfsprognosen mit denen versucht wird den Energiebedarf vorauszusagen. Tritt nun trotzdem ein Ungleichgewicht auf wird die, in Abschnitt 2.1.2 beschriebene, Frequenz-Leistungs-Regelung eingesetzt. Ein solches Ungleichgewicht tritt bei konventioneller Stromerzeugung aufgrund von Kraftwerksausfall, unvorhergesehenen großen Lasten, Abweichungen vom Fahrplan o.ä. auf, bei erneuerbaren Energien hauptsächlich durch Abweichungen des eintretenden Wetters von den Vorhersagen.
Im Regelfall allerdings wird der Strombedarf der Verbraucher durch den Kauf von Strom bei Erzeugern gedeckt und somit die Balance des Systems erhalten. Der Stromhandel wird an Energiemärkten betrieben, die sich in den Terminmarkt und den Spotmarkt aufteilen. Am Terminmarkt werden große Energiemengen für Zeiträume gehandelt die bis zu Jahren in der Zukunft liegen können. Am Spotmarkt wird Energie am Day-Ahead-Markt für den nächsten Tag sowie am Intra-Day-Markt für den selben Tag gehandelt. (Konstantin, 2013, S.48-51)
Der Stromhandel für Deutschland findet an der Europäischen Strombörse, der European Energy Exchange (EEX)[2], statt. Interessant für DERs ist dabei der Intra-Day-Handel am Spotmarkt (EPEX SPOT[3]), da diese durch die Abhängigkeit vom Wetter und dessen Varianz keine Garantie für eine Energielieferung abgeben können, die zu weit in der Zukunft liegt. Gehandelt werden dort Verträge deren Dauer von 15 Minuten bis zu mehreren Stunden reicht. Einige der gehandelten Kontrakte werden in Tabelle 2.1 dargestellt. Das Mindestvolumen der gehandelten Energiekontrakte beträgt dabei 0,1 MW (EPEX SPOT). Dieses Mindestvolumen macht es für kleinere DERs (Private Photovoltaik (PV)-Anlagen, kleinere

[2]https://www.eex.com/de/
[3]http://www.epexspot.com/de/

Firmenanlagen oder Anlagen an Behörden/Schulen) unmöglich direkt am Stromhandel teilzunehmen. Diese können die erzeugte Energie noch gegen eine feste Vergütung abgeben, oder an der Direktvermarktung des Netzbetreibers teilnehmen (Konstantin, 2013, S.113-124).

Tabelle 2.1.: Auszug an der EEX gehandelter Kontrakte (nach (Konstantin, 2013, S. 51))

Kontrakt	**Kontraktvolumen**
Baseload-Kontrakt, Blockkontrakt für jeden, Tag Mo bis So	1 MW · 24h = 24 MWh
Peakload-Kontrakt, Blockkontrakt für Mo bis Fr, 8 Uhr bis 20 Uhr	1 MW · 12h = 12 MWh
Weekend-Baseload-Kontrakt, Blockkontrakt Sa 0 Uhr bis So 24 Uhr	1 MW · 48h = 48 MWh
Stundenkontrakt für jede Stunde eines Tages	0,1 MW · 1 h = 0,1 MWh
Kombination von Stundenkontrakten zu Stundenblöcken	0,1 MW · Anzahl h
EEX-Night, 0 Uhr bis 6 Uhr	0,1 MW · 6 h = 0,6 MWh
EEX-Morning, 6 Uhr bis 10 Uhr	0,1 MW · 4 h = 0,4 MWh
EEX-Business, 8 Uhr bis 16 Uhr	0,1 MW · 8 h = 0,8 MWh
Weitere 7 Kombinationen	

2.1.2. Frequenz-Leistungs-Regelung (Regelleistung)

Die Frequenz-Leistungs-Regelung ist ein Mechanismus mit dem Übertragungsnetzbetreiber (ÜNBs) sicherstellen dass die Netzfrequenz auch bei unerwarteten Ereignissen in ihrem Netzbereich nicht zu sehr vom Sollwert von 50,000 Hz abweicht. Die Frequenz-Leistungs-Regelung ist gestaffelt in die Primär-, Sekundärregelung sowie der Minutenreserve. Tabelle 2.2 listet die Aktivierungsschwellen der Primärregelleistung bei Abweichung von der nominalen Netzfrequenz nach UCTE (2004) auf.
Wie bereits in Abschnitt 2.1.1 erwähnt können Abweichungen der Netzfrequenz im positiven sowie im negativen Bereich auftreten. Dementsprechend muss Regelleistung auch für beide Fälle vorgehalten werden. Bei einem Abfall der Netzfrequenz wird dabei weitere Leistung in das Netz eingespeist und bei einem Anstieg dementsprechend entzogen. Eine Einspeisung kann dabei durch schnell einsetzbare Kraftwerke oder aus Batterien passieren. Ein Entzug von Leistung wird durch das Laden von Batterien, das Abschalten von Kraftwerken oder das Einschalten von Großverbrauchern bewerkstelligt.

Die Regelleistungsarten unterscheiden sich dabei in ihrer Einschaltverzögerung, der Trägheit (dem Ansprechverhalten) und der Vorhaltedauer. Primärregelleistung muss sofort abrufbar

Tabelle 2.2.: Aktivierungsschwellen der Primärregelung

Name	Wert	Einheit
Nominale Netzfrequenz	50	Hz
Aktivierung der Primärregelung	±20	mHz
Aktivierung aller Primärregelungsreserven	±200	mHz

sein und die volle Leistung nach 15 bis 30 Sekunden zur Verfügung stehen. Dabei muss sie für 15 Minuten vorgehalten werden können. Sekundärregelleistung löst die Primärregelleistung nach spätesten 15 Minuten ab. Sie muss mit 30 Sekunden Verzögerung abrufbar sein und die volle Leistung nach spätestens fünf Minuten bereitstellen. Die Vorhaltedauer beträgt für Sekundärregelleistung eine Stunde. Die Minutenreserve, auch Tertiärregelung genannt, muss nach höchstens 15 Minuten die volle Leistung liefern und die Sekundärregelung ablösen (Konstantin (2013)).
Abbildung 2.1 zeigt die beschriebenen Regelleistungsarten. Dabei ist die verfügbare prozentuale Leistung über die Zeit aufgetragen.

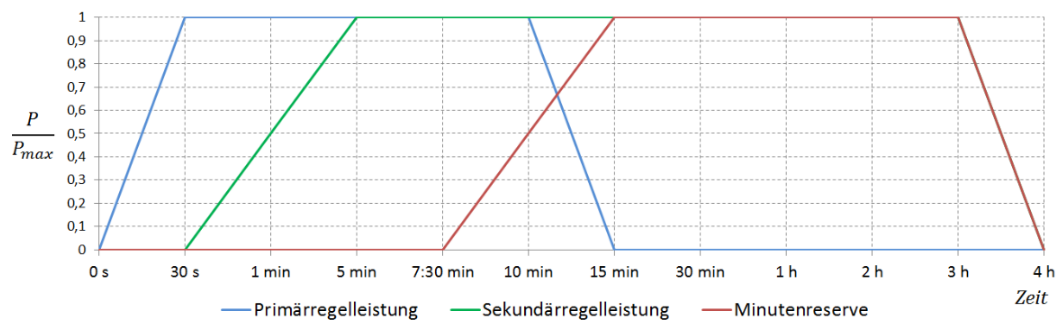

Abbildung 2.1.: Ansprechverhalten der Regelleistungsarten (aus (Dethlefs, 2014, S. 23))

2.1.3. Marktintegration von dezentralen Energieressourcen

Wie bereits in Abschnitt 2.1.1 erwähnt liegt die das Mindestvolumen für am Spotmarkt handelbare Energie bei 0,1 MW. Da diese Energiemenge durch DERs häufig nicht bereitstellbar ist, werden diese aktuell in der Direktvermarktung durch Netzbetreiber zu VK zusammengefasst. Dabei wird eine Anzahl von DERs zu einem VK aggregiert, um die Volumina der Kontrakte am Spotmarkt bereitstellen zu können. Zusätzlich zum Handel am Spotmarkt bieten Gruppen von DERs das Potential Regelenergie bereitzustellen (Schröder u. a. (2015)). Gerade DERs mit Batteriesystemen besitzen die Möglichkeit Regelenergie bereitzustellen,

da sie neben der Einspeisung von Energie in das Netz auch Energie aufnehmen und in den Batteriesystemen speichern können.

Da eine statische Gruppierung von DERs keine Adaption an Bedarfssituationen erlaubt, ist die flexible Aggregation von verschiedenen DERs ein Ziel aktueller Entwicklungen. Um beispielsweise Flexibilität für Regelenergie bereitzustellen ist es sinnvoll ein DER-Portfolio zu kreieren, bei dem jede DER ein Batteriesystem besitzt. Die im nächsten Abschnitt beschriebene Handelsplattform für Energiedienstleistungen (siehe Kapitel 2.1.4) bietet die Option DERs flexibel und dynamisch zu aggregieren, um so das geeignete Portfolio für verschiedenen Energiedienstleistungen (Regelenergie, Intraday-Handel) zu generieren.

Gehandelt wird mit Energiedienstleistungen, welche sich allerdings mit Hilfe von Flexibilität umsetzen lassen. Nach Lannoye u. a. (2012) ist Flexibilität definiert als die Fähigkeit eines Systems seine Ressourcen dem Netzbedarf nach einzusetzen. Die gehandelte Flexibiliät bedeutet also die Fähigkeit zu einem bestimmten Zeitpunkt bzw. einem bestimmten Zeitraum eine bestimmte Menge Energie in das Stromnetz einzuspeisen oder zu entnehmen bzw. den Betrieb der eigenen Anlage an äußere Bedürfnisse anzupassen. Somit kann eine Energiedienstleistung durch das Einsetzen der Flexibilität erbracht werden. Dies kann auf vielfältige Weise umgesetzt werden. Im Falle von PV-Anlagen mit Batteriesystemen zum Beispiel kann die Anlage Energie aus den Batterien in das Netz einspeisen oder Energie aus dem Netz entnehmen um damit die Batterien zu laden. Andere Wege gewünschte Lastverhalten zu bestimmten Zeitpunkten zu generieren werden unter dem Begriff DSM zusammengefasst. Darunter fallen das zeitliche Verschieben von Lasten, das Ausschalten von flexiblen Großverbrauchern und andere Methoden (Gellings (1985)).

2.1.4. Forschungsprojekt OS4ES

Das Projekt Open System for Energy Services (OS4ES) ist ein von der Europäischen Union gefördertes Forschungsprojekt, mit dem Augenmerk die Informations-, Kommunikations- und Kooperationslücke zwischen DERs und Verteilnetzbetreibern (VNBs) zu schließen. Das Projekt wird von neun Partnern ausgeführt und startete am 01. Juli 2014 mit einer Laufzeit von drei Jahren. Ziel des OS4ES-Projekts ist es, eine offene Plattform für Energiedienstleistungen zu entwickeln, um Smart-Grid-Akteuren, wie VNBs, Bilanzgruppenverantwortlichen (BGVs) und anderen, als Aggregatoren die Integration einer steigenden Anzahl von DERs in das Smart Grid zu ermöglichen.

Durch eine steigende Anzahl von DERs, in Form von Photovoltaik- und Windkraftanlagen, wächst auch die durch diese Erzeugte Leistung. Um die von den DERs erzeugte Energie am Strommarkt handeln zu können, müssen DERs aggregiert werden um handelbare Leistungsmengen zu erhalten. Daher wird im Rahmen des OS4ES Forschungsprojekts eine offene Plattform entwickelt, die einen dynamischen Handel zwischen DERs und Aggregatoren

ermöglicht.
Existierende Konzepte sehen Aggregatoren vor, die eine feste Gruppe von DERs vermarkten. Über das OS4ES sollen nun Aggregatoren befähigt werden bedarfsgerecht und flexibel DERs zu wählen und für verschiedene Energiedienstleistungen zu aggregieren (Schröder u. a. (2015)).

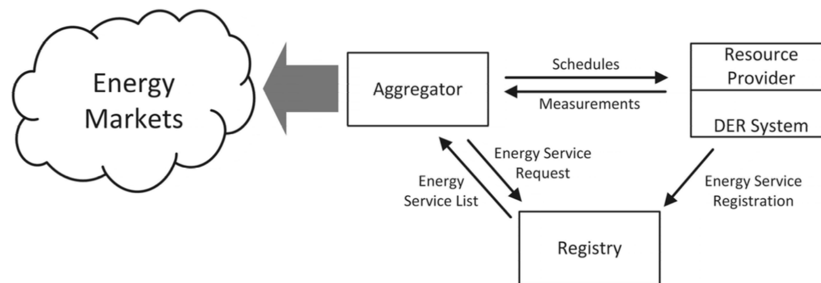

Abbildung 2.2.: Struktur des Smart Markets im OS4ES-Projekt (aus (Dethlefs u. a., 2016a, S. 2))

Abbildung 2.2 zeigt die Struktur der Handelsplattform, die im OS4ES umgesetzt wird. An dieser Plattform nehmen drei Rollen Teil: DERs, Aggregatoren und eine Registry. Zunächst registrieren DERs dabei Angebote über verfügbare Energiedienstleistungen (Energy Services) bei der Registry. Ein Aggregator kann dann in der Registry nach benötigten Energiedienstleistungen suchen und diese zu dem vom DER angegebenen Preis buchen. Die Abwicklung findet dann zwischen dem Aggregator und den DERs statt. (Dethlefs (2014)) Die Registry ist also ein Vermittler zwischen den DERs und Aggregatoren. Die von den Aggregatoren gebuchten Energiedienstleistungen können von diesen dann zusammengefasst und an regulären Strommärkten (besonders EPEX SPOT) gehandelt werden.

Aus der, im vorigen Absatz beschriebenen, Struktur wird im Forschungsprojekt OS4ES der in Abbildung 2.3 dargestellte Aufbau entwickelt. Auf der linken Bildseite sind hier die DERs dargestellt die ihre Energiedienstleistungen bei der Registry (Mitte) registrieren. Die rechts dargestellten Aggregatoren fragen die Dienstleistungen bei der Registry an und buchen diese gegebenenfalls. Weitere Informationen zum OS4ES können unter Dethlefs u. a. (2016a) und der Projektwebseite[4] eingesehen werden.

[4]http://os4es.eu/

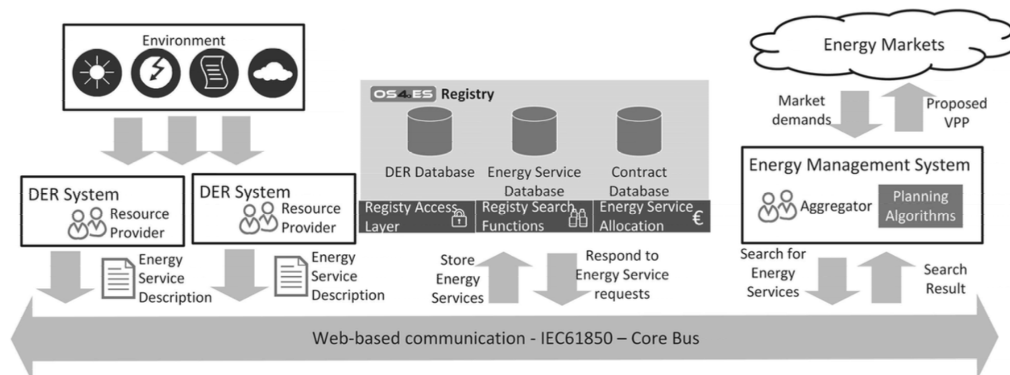

Abbildung 2.3.: Aufbau des OS4ES (aus (Dethlefs u. a., 2016a, S. 9))

2.2. Virtualisierung

Nachdem im vorangegangenen Kapitel die Grundlagen der Energieversorgung dargestellt wurden, werden nun in diesem Kapitel die Grundlagen der Virtualisierung aufgeführt. Diese ist ein grundlegender Bestandteil dieses Simulationsframeworks, weshalb zunächst zwei Arten der Virtualisierung beschrieben werden. Es gibt die Servervirtualisierung mit Virtual Machines und die Anwendungsvirtualisierung mit Softwarecontainern. Daraufhin werden für diese Arbeit relevate Produkte im Bereich der Virtualisierung beschrieben. Hier wird zunächst Docker als Lösung für Anwendungsvirtualisierung erläutert und daraufhin Rancher als Orchestrierungslösung für Softwarecontainer.

2.2.1. Virtual Machine

Eine Virtual Machine (VM) ist die virtuelle Nachstellung eines physischen Systems auf einem anderen physischen (oder virtuellen) System. Man nennt diese Art der Virtualisierung auch Servervirtualisierung. Damit wird es möglich mehrere virtuelle Maschinen auf einer physischen Maschine zu betreiben und so etwa verschiedene Betriebssysteme zu nutzen. Weiterhin ist es möglich mehrere getrennte Maschinen auf einer physischen Maschine bereitzustellen (Mandl, 2014, S.297-306).

Softwarelösungen die Virtualisierung auf der Basis von VMs anbieten (VMware[5], VirtualBox[6],

[5]http://www.vmware.com
[6]https://www.virtualbox.org

Parallels[7]) arbeiten mit einem Hypervisor. Der Hypervisor bildet die abstrahierende schicht zwischen dem realen und den virtuellen Systemen. Er stellt jeder VM eine Abbildung von Hardware (logische Laufwerke, Ressourcen) zur Verfügung. Auf diese Schicht werden die VMs aufgesetzt. Innerhalb einer VM wird immer ein eigenes Gast-Betriebssystem (Guest OS) installiert, auf welchem dann die Applikationen installiert und gestartet werden (siehe auch Mandl, 2014). Diese grundlegende Architektur von VMs auf einem Host-System wird in Abbildung 2.4 dargestellt.

Bei Virtuellen Maschinen werden also komplette virtuelle Systeme oberhalb eines realen Systems aufgesetzt und wirken für den Nutzer wie physische Maschinen. Ein Grundlegender Nachteil dieser Art von Virtualisierung ist der Leistungsbedarf des Hypervisors. Außerdem benötigen die Gast-Betriebssysteme Systemleistung und Speicherplatz, wodurch VMs teilweise sehr speicherintensiv sind (Mandl, 2014, S.299 f.).

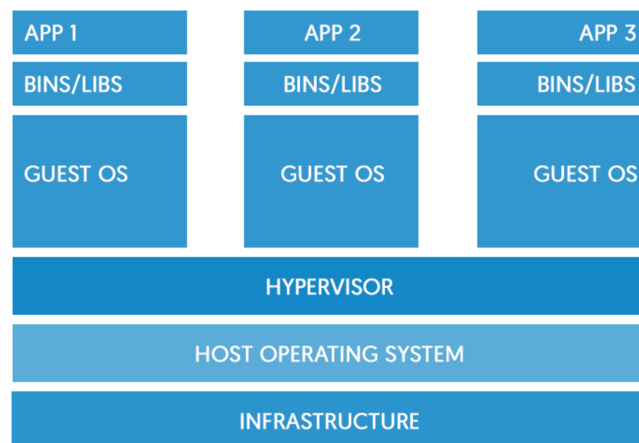

Abbildung 2.4.: Infrastruktur von VM's (aus DOCKER (c))

2.2.2. Container

Bei der Virtualisierung durch Container werden Applikationen in Containern isoliert. Dies nennt man auch Anwendungsvirtualisierung. Lösungen zur Anwendungsvirtualisierung sind zum Beispiel Solaris Containers[8], OpenVZ[9] und Docker.

[7] http://www.parallels.com/
[8] https://www.oracle.com/solaris/
[9] https://openvz.org/

Bei dieser Art von Virtualisierung wird jedem Container eine eigene Laufzeitumgebung im Host-Betriebssystem zur Verfügung gestellt. Daher nutzen alle Container das Host-Betriebssystem und nutzen gemeinsam die Ressourcen des Hosts. Sie besitzen allerdings eigene logische Laufwerke. Im Container selbst werden nun lediglich die benötigten Bibliotheken und die Applikation installiert (siehe Mandl, 2014). Diese Struktur wird in Abbildung 2.5 verdeutlicht.

Im Vergleich zu VMs sind die Container durch das fehlende Gast-Betriebsystem deutlich weniger Speicherintensiv. Durch die gemeinsame Verwaltung der Ressourcen durch das Host-Betriebssystem werden diese effizienter genutzt. Daher kann dieser Art von Virtualisierung auch als „leichtgewichtig“ beschrieben werden.

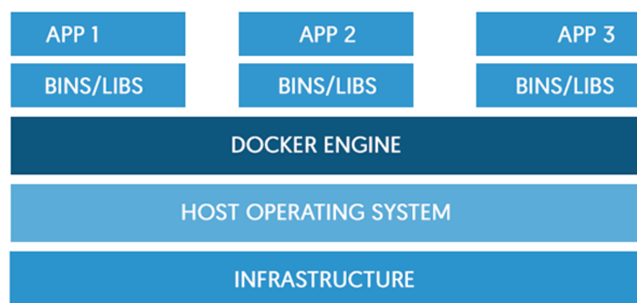

Abbildung 2.5.: Infrastruktur von Containern (aus DOCKER (c))

2.2.3. Docker

Bei Docker[10] handelt es sich um eine Virtualisierungssoftware, mit deren Hilfe Anwendungen in Containern isoliert, verteilt und ausgeführt werden können. Es handelt sich also um eine Lösung für Anwendungsvirtualisierung. Veröffentlicht wurde Docker im Jahre 2013 von der Firma dotCloud, die sich später in Docker umbenannte. Docker wird unter der Apache 2.0 Lizenz für Linux, OSX und macOS, Windows, sowie für die Cloudservices Amazon Web Services (AWS) und Azure bereitgestellt.

Neben der reinen Virtualisierung in Containern besitzt Docker zusätzliche Funktionalitäten im Management und der Verteilung der Container. Dabei besteht die Docker-Umgebung aus zwei Elementen: Der Docker Engine und dem Docker Hub. Abbildung 2.6 zeigt den Aufbau der Docker Umgebung.

[10] https://www.docker.com

Die Docker Engine ist die Docker-Applikation, die auf einem Computer ausgeführt werden muss um Docker zu nutzen. Die Architektur der Docker Engine wird in Abbildung 2.7 dargestellt. Der Kern der Engine ist der Docker Daemon welcher Docker-Objekte, wie Images, Container, Netzwerke (networks) und Laufwerke (data volumes), verwaltet. In der Docker Terminologie werden die Abbildungen von Containern Images gennant. Als Container wird dabei eine Instanz eines Images bezeichnet. Ein Container wird also durch das Erzeugen einer Instanz aus einem Image generiert. Angesprochen wird der Daemon vom Docker Client, der die API zur Verfügung stellt. Aus einem Docker-Cluster (Verbund von Docker-Daemons) lässt sich ein Docker Swarm erstellen, der eine virtuelle Docker Engine repräsentiert.

Das Docker Hub ist die offizielle Docker Registry, welche ein Repository für Docker Images darstellt. Es können anstelle des öffentlichen Docker Hubs auch private Registries genutzt werden. Docker Daemons greifen auf eingetragene Registries zu um benötigte Images herunterzuladen, die auf der lokalen Maschine nicht vorhanden sind (DOCKER (a)).

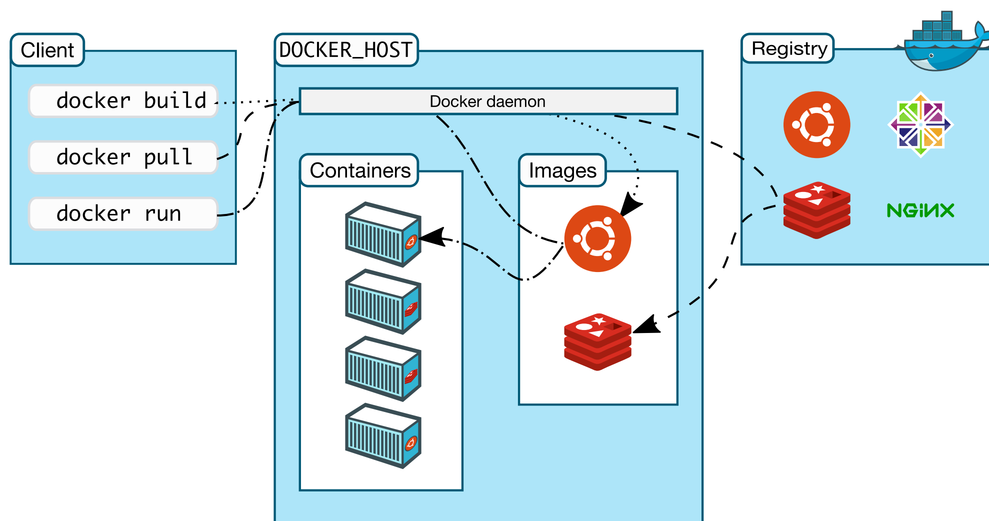

Abbildung 2.6.: Architektur der Docker Umgebung (aus DOCKER (b))

2.2.4. Rancher

Rancher[11] ist eine Open Source Container Management Plattform. Mit dieser ist es möglich Kubernetes[12], Apache Mesos[13] und Docker Swarm Container Cluster zu managen. Das Management von Container Clustern wird im Englischen Orchestration genannt. Rancher bietet

[11] http://rancher.com
[12] https://kubernetes.io
[13] http://mesos.apache.org

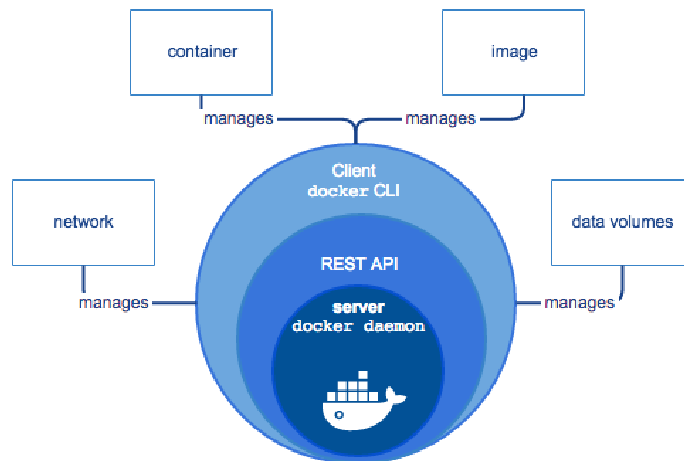

Abbildung 2.7.: Architektur der Docker Engine (aus DOCKER (b))

außerdem eine eigene Orchestration-Plattform für Docker Container namens Cattle an. Dabei managen Kubernetes und Docker Swarm jeweils Docker Container, wohingegen Apache Mesos eine eigenständige Containerlösung anbietet.

Neben dem Management und der Zeitplanung der Container Cluster bietet Rancher Funktionen für das Management der darunterliegenden Infrastruktur, Produktkataloge zum einfachen Verteilen und Starten von Anwendungen sowie Plugins für verschiedene Methoden der Nutzer-Authentifizierung.
Zum Management der Infrastruktur bietet Rancher unter Anderem die automatische Konfiguration der Container-Netzwerkschnittstellen, der verfügbaren Laufwerke (gemeinsame Laufwerke für die Umgebung oder Containereigene Laufwerke) und Load Balancing (Lastverteilung) der Kommunikation an. Außerdem besitzt Rancher einen DNS-Service, der die Container-IP-Adresse dem entsprechenden Servicenamen zuordnet. Für eine gesicherte Netzwerkübertragung, kann Rancher IPsec (siehe Kent und Seo (2005)) implementieren und somit eine vertrauliche und integre Netzwerkkommunikation bereitstellen.

Rancher wird selbst als Container auf Hosts betrieben, welche dann zu einer Infrastruktur zusammengeschlossen werden können. Eine Infrastruktur besteht in Rancher aus einer Anzahl von Hosts, von denen einer die Serverrolle übernimmt. Auf diesem wird der Rancher Server Container gestartet. Der Rancher server bietet eine Web-basierte GUI an, über welche die Rancher-Infrastruktur verwaltet werden kann. Hosts können mit Hilfe eines URL-Tokens automatisch zu einer Infrastruktur hinzugefügt werden. Am Server wird ein CLI-Befehl erzeugt, der den URL-Token enthält. Dieser wird auf dem neuen Host in der Container-Software ausgeführt und startet damit automatisch einen Rancher Agenten-Container, der sich selbst-

ständig beim zugehörigen Rancher Server registriert. Obwohl der Rancher Server selbst auch weitere Container ausführen könnte, empfiehlt Rancher eine der Host-Maschinen ausschließlich den Rancher Server Container ausführen zu lassen (siehe RANCHER).

Abbildung 2.8 zeigt den exemplarischen Aufbau einer Rancher Infrastruktur. In diesem Beispiel werden drei Hosts verwendet: Ein Server, sowie zwei Rancher-Hosts. Über die Rancher GUI, die Rancher API oder Rancher CLI des Servers können innerhalb dieser Infrastruktur Container auf allen Hosts verwaltet werden.

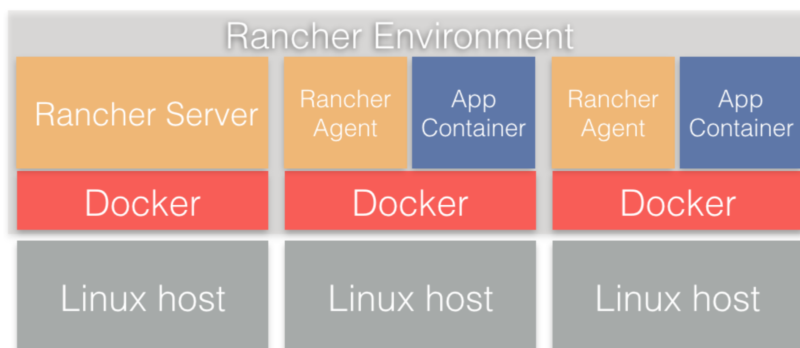

Abbildung 2.8.: Aufbau einer Rancher Umgebung

2.3. Bestehende Simulationsplattformen

Die Grundlagen der Energieversorgung und der Virtualisierung wurden bereits in den vorangegangenen Kapiteln aufgeführt. Nun sollen bestehende Simulationsplattformen im Umfeld dieser Arbeit beschrieben werden. Um eine Abgrenzung dieses Simulationsframeworks von bestehenden Lösungen durchführen zu können, ist es notwendig dass bestehende Simulationslösungen, deren Anwendungsfälle dem dieser Arbeit ähneln, in Funktionsumfang und Einsatzfeld veranschaulicht werden. Die beschriebenen Lösungen sind Mosaik, Jadex Active Components und GridSpice.

2.3.1. Mosaik

Mosaik ist eine Smart-Grid Simulationsplattform des OFFIS der Universtiät Oldenburg. Mosaik soll die automatische Erstellung einer Smart Grid Simulation unter Verwendung bereits bestehender Simulationsmodelle ermöglichen (OFFIS (2012)).

Im folgenden soll kurz das Prinzip und der Funktionsumfang dieser Simulationsplattform dargestellt werden.

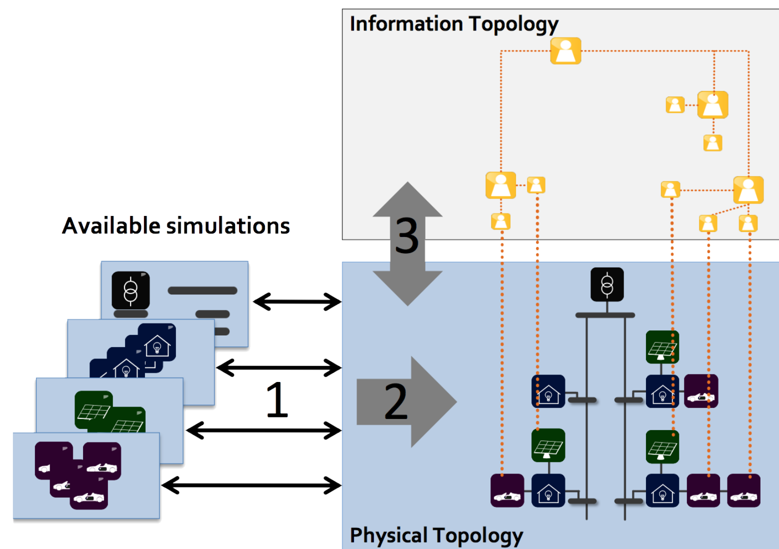

Abbildung 2.9.: Integrationsproblem von Simulationen (aus Scherfke und Schütte (2012))

Wie bereits kurz erwähnt ist eine zentrale Problemstellung der Mosaik Plattform die Nutzung von bestehenden Simulatoren in einem Co-Simulations-Szenario. Dabei besteht die Problematik darin, dass Simulatoren im Regelfall nicht für die Wiederverwendung in anderen Simulationen ausgelegt sind. Daher besitzen diese keine Schnittstelle, über welche die Simulation gesteuert werden kann. Diese Problemsituation wird in Abbildung 2.9 dargestellt.

Auf der linken Seite sind die bestehenden Einzelsimulationen dargestellt. Zu Nummer 2 müssen diese zur einem Szenario zusammengefasst und simuliert werden. Für Nummer 3 müssen die Simulatoren in eine Kommunikationsinfrastruktur integriert werden und die nötigen Kontrollstrukturen unterstützen (Scherfke und Schütte (2012)).

Zur Umsetzung diese Ziele werden die in Abbildung 2.10 dargestellten Komponenten verwendet. Auf einem zentralen Server wird ein Kontrollprogramm betrieben, das eine Anbindung zu grafischen Oberflächen unterstützt. Da Mosaik eine verteilte Architektur der Simulation erlaubt, gibt es Platform-Manager-Programme, die auf jeder Maschine gestartet werden müssen um diese in der Simulation zu verwenden. Platform Manager finden automatisch gestartete (und für Mosaik nutzbare) Simulatoren auf den jeweiligen Maschinen und sendet eine Liste dieser an das Kontrollprogramm. (Scherfke und Schütte (2012))

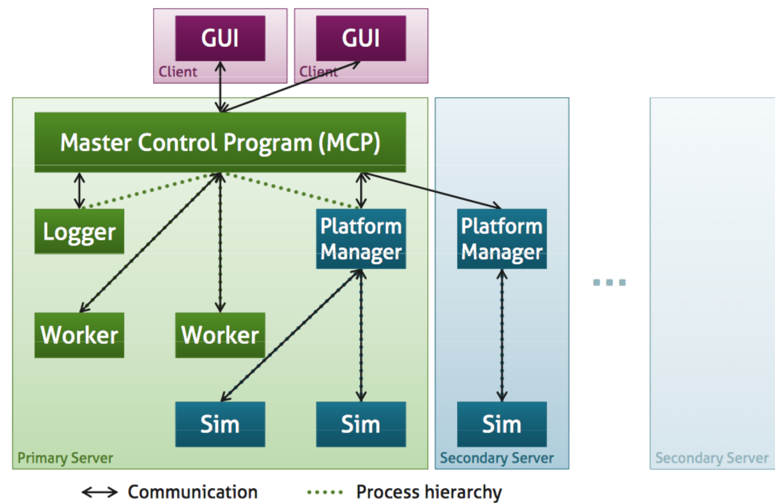

Abbildung 2.10.: Komponenten der Mosaik-Plattform (aus Scherfke und Schütte (2012))

Zur Integration von Simulatoren in eine Mosaik-Co-Simulation müssen Simulatoren die SimAPI implementieren. Damit bietet Mosaik eine Simulation in einem diskreten Zeitmodell mit fester Schrittgröße an. (Rohjans u. a. (2013)) Simulationsentwickler müssen dabei nur vier Methoden implementieren, die das Mosaik-Framework auf dem Simulator ausführt. Für Python wird eine High-Level-API bereitgestellt, über die Simulationsentwickler dieses Interface direkt implementieren können. Andere Programmiersprachen müssen die Low-Level-API verwendet und dabei Nachrichten der ZeroMQ-Bibliothek empfangen können und interpretieren. Der Vergleich der Low-Level- und der High-Level-API wird in Abbildung 2.11 dargestellt.

Der Kommunikationsablauf, der während einer Simulation zwischen dem Mosaik-Framework und einem Simulator der die SimAPI integriert hat abläuft, wird in Abbildung 2.12 gezeigt. Der Simulator erhält wiederholt Nachrichten mit den jeweiligen Parametern die für einen Simulationsschritt benötigt werden und der Aufforderung zur Simulation. Wenn der Simulator seinen Schritt abgeschlossen hat, sendet dieser seine Ergebnisse an Mosaik, woraufhin der nächste Simulationsschritt gestartet wird. Dies endet wenn das Simulationsziel erreicht ist und Mosaik einen Stop-Befehl an den Simulator sendet.

Zusammenfassend bietet Mosaik also eine Lösung für die Co-Simulation von Szenarios an, bei möglicher Wiederverwendung von Simulatoren, die um die Mosaik-SimAPI erweitert wurden. Es ist jedoch notwendig Maschinen manuell zu starten, darauf Simulatoren zu platzieren und zu starten sowie den Platform Manager darauf zu starten, um diese in einer Simulation verwenden zu können.

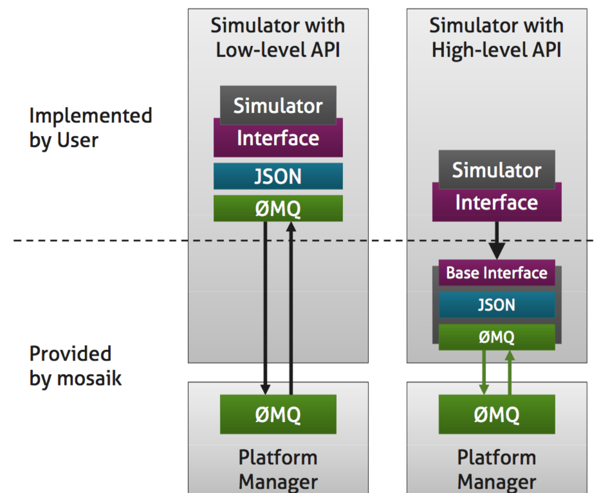

Abbildung 2.11.: Vergleich der Mosaik APIs (aus Scherfke und Schütte (2012))

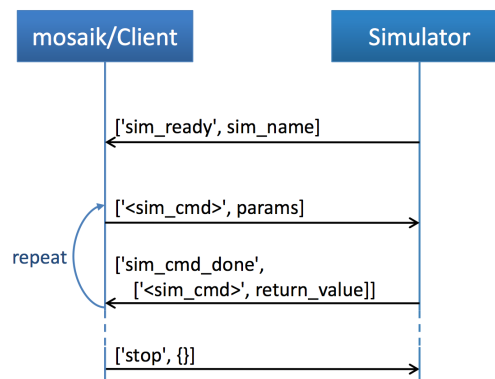

Abbildung 2.12.: Kommunikationsablauf bei Simulationsschritt (aus Schütte u. a. (2012))

2.3.2. Jadex Active Components

Jadex Active Components ist ein Belief Desire Intention (BDI) Agenten System das an der Universität Hamburg entwickelt wurde. Jadex stellt ein Agenten-Framework bereit, das als eine Erweiterung der JADE[14]-Middleware-Plattform realisiert wurde. Durch Jadex wird neben rein rational handelnden Agenten eine Unterstützung für Schlussfolgerungen (Reasoning) durch die Ausnutzung des BDI-Models realisiert Braubach u. a. (2004) .
Im Folgenden soll kurz das Prinzip und die Funktion von Jadex Active Components vorgestellt werden.

[14]http://jade.tilab.com

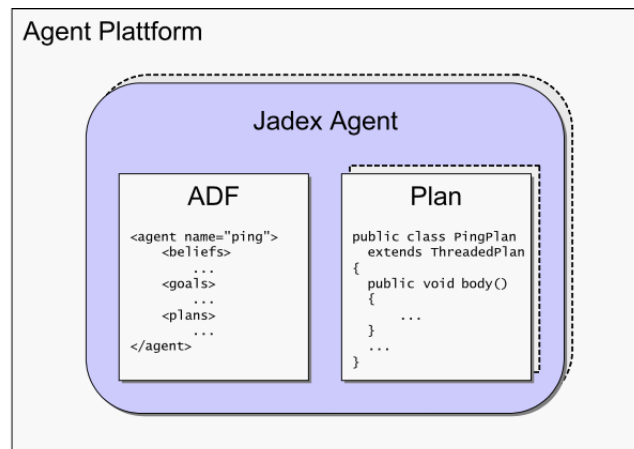

Abbildung 2.13.: Definition eines Jadex Agenten (aus Pokahr u. a. (2005))

Jadex Agenten bestehen aus zwei Komponenten, dem Agent Definition File (ADF) und dem Programmcode der implementierten Pläne. Diese Situation wird in Abbildung 2.13 dargestellt. Die Definition der Belives, Goals und Plans in einer ADF wird mithilfe eines XML-Schemas umgesetzt. Die Implementation der Pläne muss in Java durchgeführt werden. Die Agenten benötigen dabei diese beiden Komponenten um genutzt werden zu können Pokahr u. a. (2005).

Um nun Komponenten ausführen zu können muss auf einer Maschine (virtuell oder physisch) die Jadex-Plattform ausgeführt werden. Über diese Plattform kann dann der Agent gestartet werden. Es ist dabei möglich mehrere Agenten auf einer Maschine auszuführen oder auch ein verteiltes System zu erstellen. Die Agenten kommunizieren dabei über ein Overlay-Network, also eine durch JADE implementierte Middleware. Kommunikation zwischen Agenten wird als Services umgesetzt (Pokahr u. a. (2005)). Anhand der Beschreibung wird also deutlich, dass jede Maschine manuell aufgesetzt werden muss.

Durch den Aufbau als Mutiagentensystem (multi-agent system) lässt sich die Jadex-Plattform auch für Simulationen verwenden. Dabei müssen die Komponenten, wie zuvor erwähnt in der Programmiersprache Java erstellt werden und zusätzlich mit einer ADF ausgestattet werden. Das gewünschte Szenario müsste dabei manuell konfiguriert oder ein Agent erstellt werden der die Szenariokonfiguration vornimmt.
Dabei bietet Jadex einen Zeitservice (Time-Service) für Eventbasierte Simulation an. Dessen Funktionsprinzip wird in Abbildung 2.14 dargestellt. Es gibt eine zentrale Instanz des Zeitservices welche das Voranschreiten der Simulationszeit koordiniert und steuert. An diesem registrieren die Agenten bzw. Komponenten die Zeitpunkte an denen sie Aktivitäten ausführen müssen. Der Zeitservice fungiert dann als Scheduler und informiert die Agenten über das Erreichen der Zeitpunkte und fordert diese damit zum Ausführen der Aktivitäten auf.

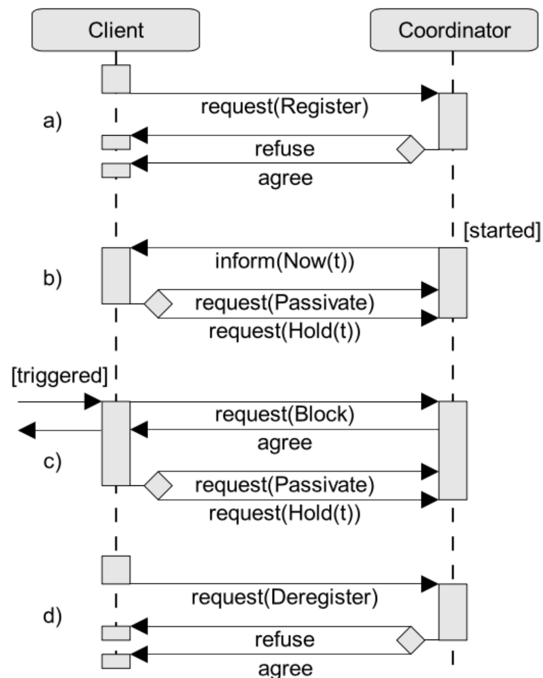

Abbildung 2.14.: Jadex Time Service Protokoll (aus Braubach u. a. (2006))

Zeitschritte in denen kein Agent eine Aktivität ausführen muss können daher übersprungen werden, was die Ausführung der Simulation beschleunigt (Braubach und Pokahr (2012)).

Damit stellt Jadex Active Components eine Multi-Agenten-Plattform bereit mit der Simulationen in Kooperation von Agenten ausgeführt werden können. Auf der Jadex Plattform können auch Services aufgesetzt werden die eine Konfiguration der Szenarioarchitektur und Komponenten vornehmen.
Jadex bietet dazu das Environment Support Paket an. Dieses Paket beinhaltet mehrere APIs, Framework-Klassen und Werkzeuge die das Erstellen von Simulationsapplikationen in Hinsicht auf Trennung der Umgebung und Simulationskomponenten, Einfache Erstellung und Konfiguration von virtuellen Umgebungen erleichtern sollen. Außerdem werden Werkzeuge zur Simulationsausführung, Simulationsüberwachung und Ergebnisanalyse bereitgestellt (Braubach und Pokahr (2011)).

Mit Jadex Active Components wird also eine Plattform bereitgestellt die die Ausführung von Simulationen durch ein Netzwerk von Simulationsagenten ermöglicht. Ebenfalls stellt Jadex mit EnvSupport (kurz für Environment Support) ein Paket bereit, mit dem die Konfiguration,

Ausführung und Überwachung von Simulationen vereinfacht wird.

2.3.3. GridSpice

In den vorigen Kapiteln wurden mit Jadex Active Components und Mosaik bereits zwei verwandte Arbeiten vorgestellt. In diesem Kapitel soll nun das GridSpice Framework der Stanford Universität erläutert werden. Dazu wird zunächst ein kurzer Überblick über das Framework gegeben und dann die Architektur dargestellt.

GridSpice ist ein skalierbares open-souce Simulationsframework, das für die Co-Simulation von Smart-Grid-Anwendungen entwickelt wurde. GridSpice integriert dabei GridLab-D[15] um Stromnetze zu simulieren und MATPOWER[16] um optimierungsprobleme dieser Netze zu lösen. Neben der Nutzung einer Cloud-Infrastruktur bietet GridSpice ein Web-Interface und eine Python-Bibliothek an, über die das Framework, sowie Simulationen, konfiguriert und ausgeführt werden können. Dabei ist GridSpice auf die Simulation von Stromnetzen und Produzenten/Verbrauchern ausgelegt. Es können zusätzlich weitere Funktionen simuliert werden (Anderson u. a. (2014)).

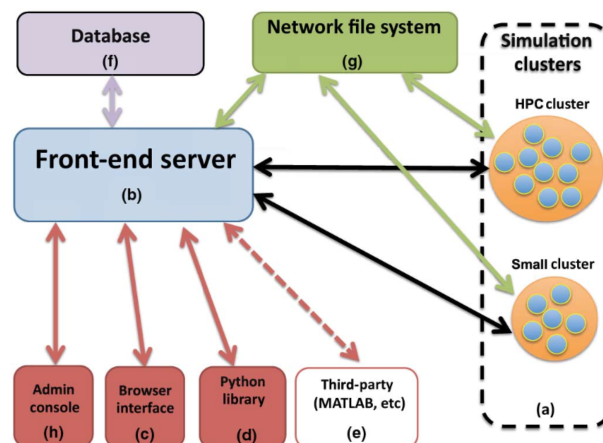

Abbildung 2.15.: Architektur des GridSpice-Frameworks (aus Anderson u. a. (2014))

Zur Veranschauchlichung der Framework Architektur wird diese in Abbildung 2.15 dargestellt. GridSpice verwendet einen zentralen Front-End-Server (b) zur Steuerung und Verwaltung eines Simulationssystems. Dieser Stellt eine REST-Schnittstelle zur Verfügung, über

[15]http://www.gridlabd.org
[16]http://www.pserc.cornell.edu/matpower/

die Steuerungskomponenten, wie die Browser-Schnittstelle (c) und die Python-Bibliothek (d) Zugriff auf das Simulationssystem erhalten. Die Simulatoren selbst befindet sich in Simulations-Clustern (a), auf die später noch genauer eingegangen wird.
Um Simulationsergebnisse zu übergeben und andere Simulationsdaten zwischenzuspeichern wird ein Netzwerk-Datensystem (g) verwendet. GridSpice nutzt dazu den Amazon S3[17] Cloudspeicher. Zur dauerhaften Speicherung werden Nutzerdaten, Simulationsdaten und -Ergebnisse in einer Datenbank (f) gespeichert (Anderson u. a. (2014)).

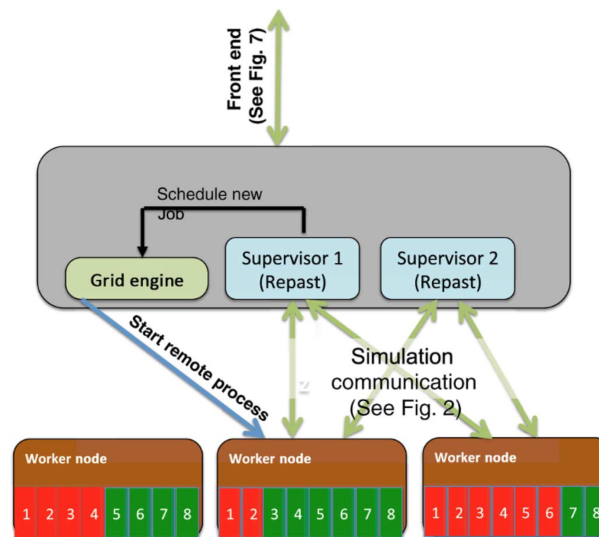

Abbildung 2.16.: Aufbau eines GridSpice Simulations-Clusters (aus Anderson u. a. (2014))

Die bereits kurz erwähnten Simulations-Cluster werden in einer Cloud-Umgebung ausgeführt. GridSpice nutzt dazu AWS. Innerhalb einer Simulation können mehrere Cluster gemeinsam verwendet werden. Abbildung 2.16 stellt einen solchen Simulations-Cluster dar. In jedem Cluster gibt es einen Master-Node (eine Hauptinstanz), dieser kreiert, je nach Auslastung, Arbeiterinstanzen. In Abbildung 2.16 bestehen drei Arbeiterinstanzen. Ein weiterere Prozess würde zunächst auf Arbeiterinstanz 2 gestartet werden, da diese die geringste Auslastung besitzt. Sollten alle Instanzen voll ausgelastet sein, wird eine neue Arbeiterinstanz gestartet. Ist eine Instanz im Leerlauf wird diese abgeschaltet (Anderson u. a. (2014)).

AWS ist ein Pay-As-You-Go-Service. Dies bedeutet dass der Kunde stundenbasiert für die Kapazitäten zahlt die er benutzt. Wir also kurzfristig mehr Leistung benötigt, kann diese so lange genutzt werden wie nötig und es muss auch nur für diesen Zeitraum gezahlt werden. So können bei Bedarf Serverinstanzen gestartet und beendet werden, je nachdem, wie viel Leistung momentan benötigt wird (Amazon Web Services (2017)).

[17]https://aws.amazon.com/s3/

Das in GridSpice umgesetzte Zeitmodel sieht eine diskrete globale Simulationsuhr vor. Das Voranschreiten der Simulationsuhr wird eventbasiert gesteuert. Sollten Schritte mit der minimalen Zeitspanne t_{min} keine neuen Ergebnisse bringen wird die Uhr Uhr bis zur nächsten Änderung der Ergebnisse weitergestellt. Die Zeitbasis t_{min} ist dabei je nach Simulation wählbar und kann minimal eine Sekunde betragen (Anderson u. a. (2014)).
Die Methode zur Findung des nächst nötigen Zeitschrittes in GridSpice wird in Abbildung 2.17 dargestellt. Hier geben die Simulatoren der Verteilnetze, Übertragungsnetze und Erzeuger an, zu welchem Zeitpunkt sich die Ergebnisse ändern. Hat sich zum vorangegangenen Zeitschritt an den Ergebnissen nichts geändert, so schreitet die Uhr um t_{next} voran. Gab es allerdings Änderungen schreitet die Uhr nur um t_{min} voran.

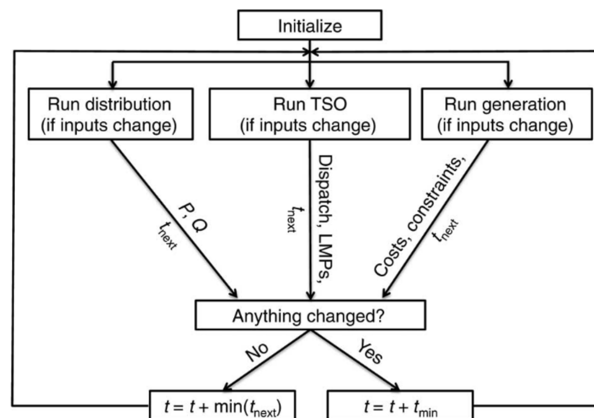

Abbildung 2.17.: Gridspice: Methode zur Findung des nächsten Zeitschrittes (aus Anderson u. a. (2014))

Somit bietet GridSpice ein Simulationsframework mit dem Cloud-basierte Simulationen von Stromnetzen und Netzanwendungen ausgeführt werden können. Es nutzt dabei ein Gruppe von Agenten, die als verteiltes System die Simulation ausführen. Es ist stark an die Nutzung mit Amazon Web Services gebunden. Womit keine private Infrastruktur zur Ausführung von Simulationen verwendet werden kann und auf die kostenpflichtige AWS-Plattform zurückgegriffen werden muss.

Nachdem in den vorigen Kapiteln die Grundlagen der Energieversorgung und Virtualisierung dargestellt wurden, wurden nun drei bestehende Simulationsumgebungen vorgestellt, die sich im Umfeld dieser Arbeit befinden. Die Funktion und Architektur dieser Simulationsumgebungen wurde bis zu einem gewissen Grad erläutert.
Nachfolgend soll nun das zu simulierende System beschrieben und die Anforderungen analysiert werden, die sich aus dem geplanten Nutzungsverhalten des zu entwickelnden Simulationsframeworks ergeben.

3. Anforderungen und Analyse

In diesem Kapitel werden die Anforderungen an das Simulationsframework analysiert. Zunächst wird daher in Abschnitt 3.1 das zu simulierende System beschrieben und daraufhin untersucht welche Anforderungen sich aus dem Simulationssystem und dessen Logik für den Simulationsablauf und die Kommunikationsschicht ergeben. Dazu werden in Kapitel 3.2 die Simulationslogik und der Simulationsablauf, die sich aus der Nutzung für das, in Abschnitt 2.1.4 beschriebene, OS4ES ableiten, eingehend analysiert und im Weiteren spezifiziert welche Anforderungen sich daraus für die Simulationsplattform ergeben. In Kapitel 3.3 wird dann abgeleitet welche Anforderungen sich aus den Gegebenheiten für die Kommunikationsebene darstellen.

3.1. Systembeschreibung

Nachdem in Kapitel 2.1.4 zunächst die grundlegende Architektur des OS4ES aufgezeigt wurde, wird hier nun eine eingehendere Beschreibung dargelegt, anhand welcher die Anforderungsanalyse durchgeführt wird.

Kern des OS4ES ist ein Handelsplatz für Energiedienstleistungen, an dem auch Kleinerzeuger teilnehmen können. Im Zusammenhang mit der Simulation des OS4ES liegt der Schwerpunkt auf der Simulation der Handelsplattform bzw. des Handels innerhalb dieser. Es werden daher keine weiteren Energiemärkte wie zum Beispiel EPEX SPOT simuliert. Abbildung 3.1 stellt dar welcher Bereich des in Abbildung 2.2 abgebildeten Energiehandels im Framework simuliert werden soll. Dabei nehmen generell drei verschiedene Rollen am Handel teil, deren Funktion nachfolgend erläutert wird: DERs, Aggregatoren und die Registry.

Dezentrale Energieressource

Ein DER kann im OS4ES ein Kleinerzeuger oder auch ein VK, bestehend aus mehreren DERs, sein. Diese bestimmen mithilfe von Prognosen (Wetterprognosen, Eigenbedarfsprognosen) die handelbare Flexibilität und den Preis zu dem diese angeboten werden kann. Daraufhin registrieren die DERs diese Energiedienstleistung bei der Registry. Möchte ein

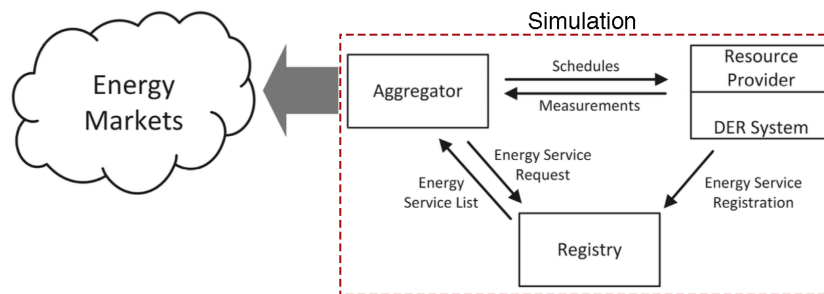

Abbildung 3.1.: Simulierter Bereich des OS4ES (nach Dethlefs u. a. (2016a))

Aggregator eine Energiedienstleistung buchen, so kontaktiert dieser dafür das DER direkt und schließt mit diesem die Buchung ab.

Registry

Die Registry besteht aus zwei verschiedenen Registern: Den White Pages und den Yellow Pages. Das White Pages Register wird für statische Daten der registrierten DERs genutzt, wie etwa ID, den Typ des DER-Systems, der Anschlusspunkt zum Stromnetz usw. Die Yellow Pages enthalten dynamische Informationen. Dynamische Informationen sind die von den DERs angebotenen Energiedienstleistungen (Art der Dienstleistung, Energiemenge usw.). Dabei ist zu erwähnen dass DERs eine Vielzahl von Dienstleistungen anbieten können. Die DERs registrieren sich und ihre Energiedienstleistungen bei der Registry. Die Aggregatoren fragen bei der Registry geeignete Dienstleistungen an. Wählt ein Aggregator eine Dienstleistung aus, teilt die Registry diesem die nötigen Kontaktinformationen des DERs mit, damit dieser den Handel direkt mit dem DER abschließen kann.

Aggregator

Ein Aggregator im OS4ES ist das Koppelelement der DERs zu den Energiemärkten. Durch dessen Funktionalität als Bündelungselement, der von den DERs angebotenen Dienstleistungen, kann der Aggregator die Energiedienstleisungen an den Energiemärkten handeln (Dielman und van der Velden (2003)). Um bestimmte Kontrake zu erfüllen fragt der Aggregator zum Kontrakt passende Energiedienstleistungen bei der Registry (Yellow Pages) an. Diese stellt eine Liste mit verfügbaren Dienstleistungen der DERs mit zugehörigen Daten und Preisen bereit. Aus dieser wählt der Aggregator eine oder mehrere Dienstleistungen aus, die er benötigt um den Kontrakt zu erfüllen. Die Registry stellt dem Aggregator daraufhin

die Kontaktdaten der gewählten DERs (White Pages) zur Verfügung. Der Aggregator kann nun den Handel direkt mit den DERs abschließen.

Nach der Erläuterung der Rollen der Simulation soll im Folgenden nun die Struktur der Komponenten dargestellt werden die diese verkörpern:
Die OS4ES-Registry besitzt eine verteilte Architektur und besteht aus mehreren Logical Registry Entitys (LREs). Ein LRE ist dabei eine Einheit bestehend aus White Pages und Yellow Pages, die die Datenbank für jeweils eine Zone enthält. Es ist jedoch auch möglich dass mehrere LREs innerhalb einer Zone existieren. In diesem Fall wird einer der LREs zur Haupt-LRE erklärt und weitere LRE müssen sich mit dieser abstimmen. Zonen teilen die DERs durch deren Betriebseigenschaften, etwa Art des DER oder Örtlichkeit, auf (Dethlefs u. a. (2015a)). Die Architektur der Registry-Komponenten und deren Zusammenhänge werden in Abbildung 3.2 dargestellt. die Logical Registry Entities decken Zonen des physischen Systems ab. Die Baumstruktur der LREs wird durch Registry Index Service (RIS) aufgelöst. Über den Access Point (AP) erhalten Aggregatoren und DERs Zugang zu den verantwortlichen LREs. Abbildung 3.3 zeigt die organisatorische Architektur der Registry. Die weiß dargestellten LREs decken die physischen Zonen ab, während die schwarz dargestellten Registry Index Services die Baumstruktur auflösen.

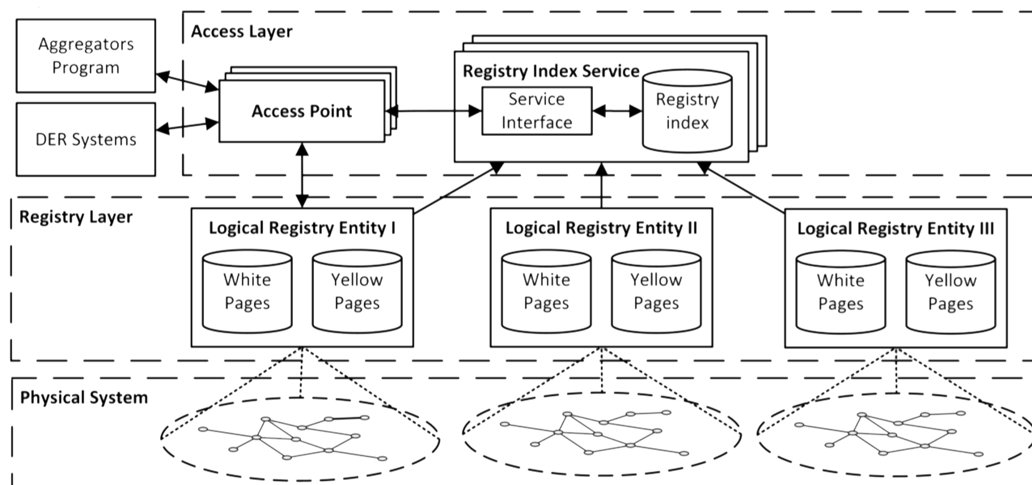

Abbildung 3.2.: Architektur der Registry-Komponenten (aus Dethlefs u. a. (2015b))

Die Struktur von dezentralen Energieressourcen ist höchst variabel und kann daher nicht eindeutig festgelegt werden. Daher wird hier beispielhaft ein DER beschrieben, dass aus einer PV-Anlage mit Batteriesystem besteht. Die physische Architektur der Anlage wird in Abbildung 3.4 dargestellt. Diese zeigt die physischen Komponenten und deren Anschluss an das Stromnetz. Dies ist eine beispielhafte Zusammenstellung der Komponenten. Durch

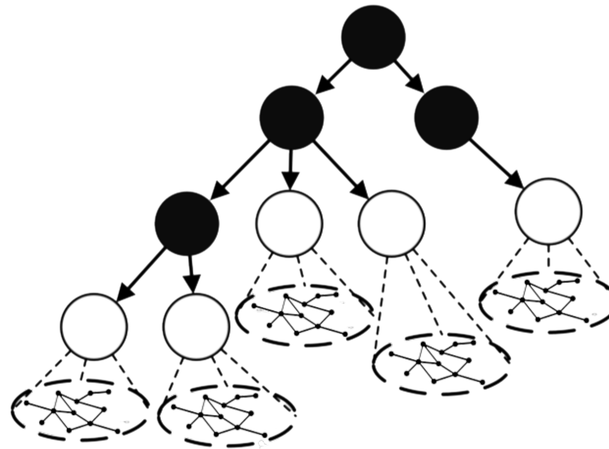

Abbildung 3.3.: Organisatorische Architektur der Registry (aus Dethlefs u. a. (2015b))

Integration der Wärmeproduktion in das PV-System oder andere Modifikationen lässt sich dieses erweitern. Dieses Beispiel zeigt ein PV-System mit Eigenverbrauchsoptimierung. Das bedeutet, dass zum einen versucht wird den Eigenbedarf der angeschlossenen Verbraucher durch die PV-Anlage zu decken und den planbaren Bedarf außerdem so zu organisieren, dass möglichst viel des Eigenbedarfs durch die PV-Anlage versorgt werden kann. Die Energieaufname aus dem Stromnetz wird also möglichst minimiert. Durch die Vermarktung der Flexibilität kann die Aufnahme von Energie aus dem Stromnetz allerdings in einigen Situationen günstiger sein, als die Nutzung von Batterriereserven. Ein Beispielsystem, an dem diese Planung und Vermarktung durchgeführt wird, kann in Behncke (2017) eingesehen werden. Zur Durchführung der Planung im Planungsmodul bestehen Kommunikationsverbindungen innerhalb des Systems zwischen dem Planungsmodul und dem PV-Modul, dem Batteriesystem, den Verbrauchern und dem Trennschalter.

Um nun die Flexibilität in Form von Energeidienstleistungen vermarkten zu können, muss der Zugang zur Handelsplattform hergestellt werden. Abbildung 3.5 zeigt die Kommunikationswege der Anlage. Zunächst besteht eine Verbindung zu einem Wetterdienst, der Prognosen bereitstellt anhand derer die vorraussichtliche Leistung der PV-Module bestimmt wird. Vom PV-Modul selbst werden Momentanwerte geliefert, über welche die Korrektheit der Planungsprognose überprüft wird. An das Batteriesystem sowie die Verbraucher können Fahrpläne übertragen werden und von ihnen aktuelle Informationen zum Zustand an das Planungsmodul gesendet werden. Bei Bedarf kann das Planungsmodul das Gesamtsystem vom Stromnetz trennen (über den Trennschalter) und so veranlassen den Eigenbedarf komplett über die PV-Anlage und das Batteriesystem zu bewältigen ((Quaschning, 2015, S. 250-255)).

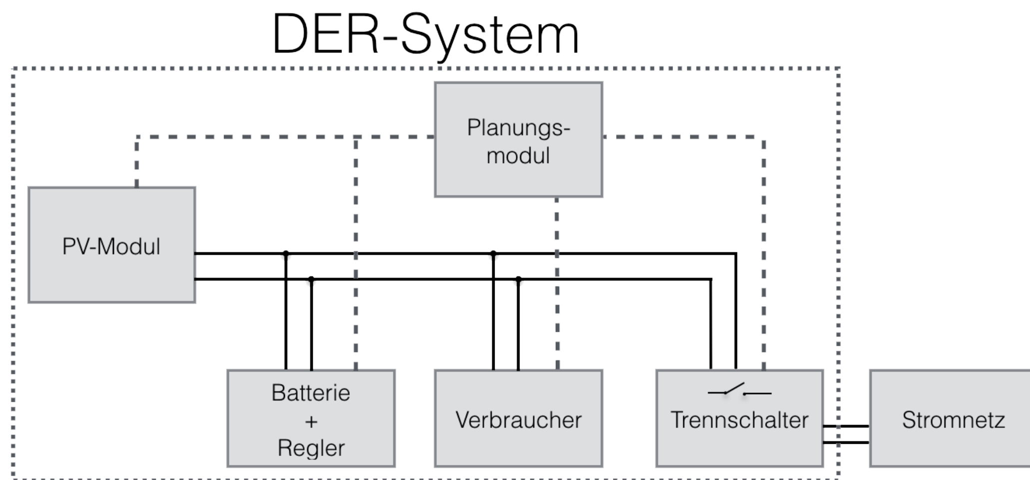

Abbildung 3.4.: Komponenten einer PV-Anlage

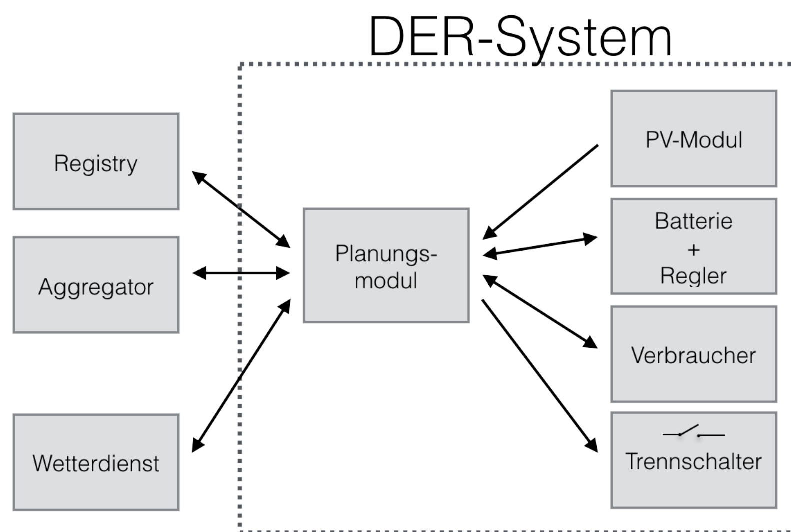

Abbildung 3.5.: Kommunikationswege der PV-Anlage

Die Struktur eines Aggregators ist nicht einheitlich definiert (Dielman und van der Velden (2003)). Daher wird in Abbildung 3.6 ein beispielhafter Programmablaufplan vorgestellt, der die grundlegenden Tätigkeiten eines Aggregators im OS4ES zeigt. Dort wird das in Dethlefs u. a. (2016b) vorgestellte „Aggregation on Demand“-Prinzip angewendet. Nach dem Start des Aggregators reagiert dieser auf auf mögliche handelbare Kontrakte am Strommarkt. Der Aggregator fragt passende Energy Services (ESs) bei der Registry an. Aus den Resultaten wählt dieser die nötige Menge an ES aus, die nötig ist um den Kontrakt zu erfüllen. Dabei wählt der Aggregator die Angebote mit dem optimalen Preis-leistungs-Verhältnis, um selbst das bestmögliche Angebot machen zu können. Sollte das Angebot am Strommarkt ange-

nommen werden, schließt der Aggregator den Handel mit den DERs ab und erfüllt daraufhin den am Strommarkt abgeschlossenen Kontrakt.

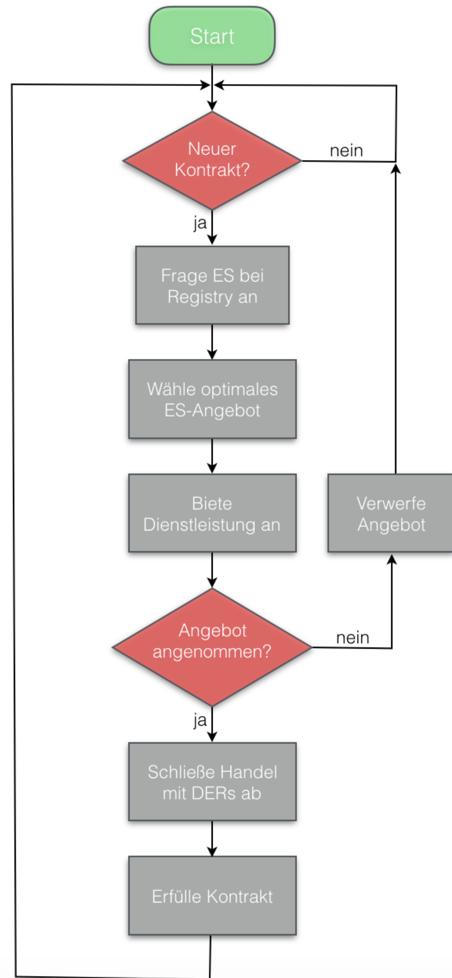

Abbildung 3.6.: Flussdiagramm eines Aggregators

Der Ablauf den gesamten Handels wird in Abbildung 3.8 als Flussdiagramm dargestellt. Hier ist exemplarisch der Handel mit Teilnahme nur eines DER-Systems und eines Aggregators dargestellt. Erkennbar sind die äußeren Abhändigkeiten vom Strommarkt sowie vom Wetterdienst.

Anhand der hier gegebenen Beschreibung lässt sich eine vom OS4ES abstrahierte Struktur des zu simulierenden Systems ableiten. Es handelt sich dabei um einen Handelsplatz mit einer Broker-ähnlichen Registry als zentralem Vermittlungspunkt. Die Klienten kontaktieren die Registry um nach bestimmten Produkten oder Dienstleistungen zu suchen. Diese Produkte werden von den Verkäufern bei der Registry eingetragen und können daraufhin

gekauft werden. Dabei stellt die Registry eine Plattform dar, die zwischen den Handelspartnern vermittelt. Diese abstrakte Struktur wird in Abbildung 3.7 dargestellt.
Durch eine Skalierbarkeit der zentralen Registry können prinzipiell unendlich viele Akteure in einem Handelsplatz partizipieren. Um die Relationen der Rollen zu verdeutlichen werden nun die Kardinalitäten zwischen den Rollen und Produkten aufgezeigt. Ein Verkäufer und Produkt haben eine 1:n Beziehung. An einer Registry können n Verkäufer Produkte listen, es besteht also eine n:1 Beziehung zwischen Verkäufern und Registry. Registry und Aggregatoren besitzen ebenfall eine 1:n Beziehung. Da ein Käufer viele Verkäufer anfragen kann und diese mit eine Vielzahl von Käufern handeln können besteht eine n:m Beziehung.

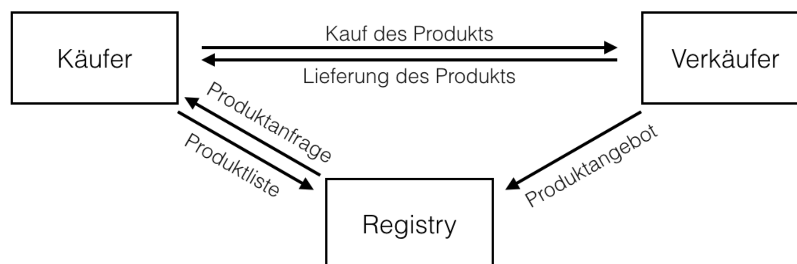

Abbildung 3.7.: Struktur des Handels im OS4ES

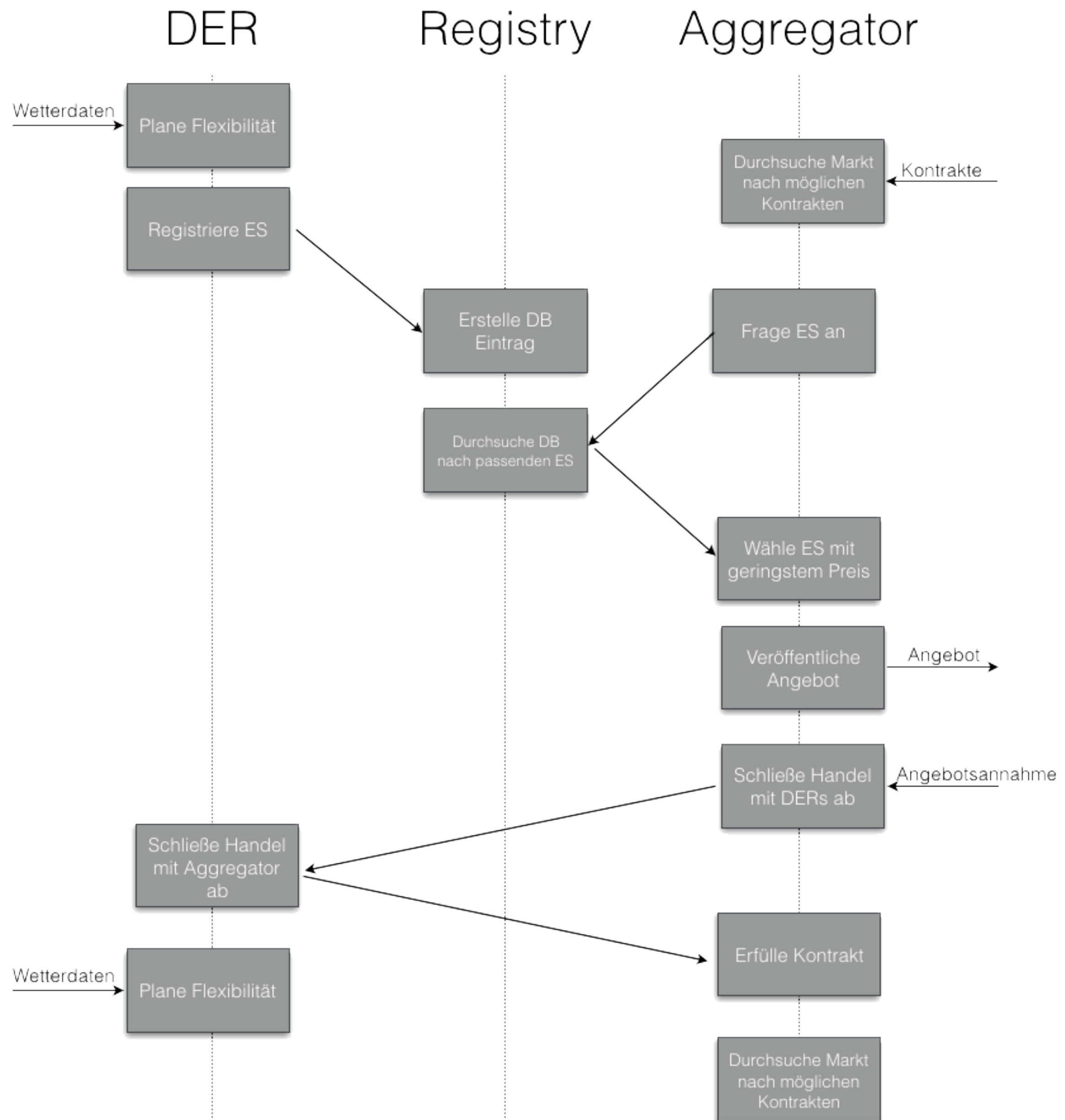

Abbildung 3.8.: Sequenzdiagramm des Handels im OS4ES

3.2. Anforderungen an die Plattform

Der Handel des in der Systembeschreibung dargestellten Systems soll, mithilfe des in dieser Arbeit entwickelten Frameworks, simuliert werden. Die Gesamtsimulation wird dabei durch eine Vielzahl von Einzelmodellen ausgeführt, die die am Markt beteiligten Rollen (siehe Abschnitt 3.1) darstellen. Um Flexibilität bei der verwendeten Hardware zu schaffen, soll das Simulationsframework möglichst unabhängig von der verwendeten Infrastruktur sein und dazu eine abstrahierende Schicht zwischen den physischen Maschinen und den Modellen der Simulation geschaffen werden.

Des Weiteren soll das Simulationsframework für verteilte Simulationen ausgelegt werden. Die Modelle werden dabei als Software-Agenten (siehe Wooldridge (2013)) verstanden, die zusammen ein verteiltes System bilden und die Simulation ausführen. Das Framework soll in der Lage sein Modelle zu unterstützen die in den Programmiersprachen Java oder Python erstellt wurden.
Um die Anforderungen der Modellarchitektur für Entwickler zu erleichtern, sollen möglichst einfache definierte Schnittstellen zwischen dem Framework und den Modellen geschaffen werden. Dies betrifft zum einen die Schnittstellen der simulationsinternen Kommunikation sowie den Schnittstellen zur Parametrisierung, Konfiguration und Management der Modelle. Daher soll eine High-Level-API erstellt werden, die in der Programmiersprache des Modelles (Java, Python) zur Verfügung steht und es den Entwicklern ermöglicht die Schnittelle, durch Implementation der Funktionen in der nativen Programmiersprache, einzubinden.

Das Simulationsszenario soll über eine Front-End-Komponente mit User Interface (UI) konfigurierbar sein. Diese Komponente soll anhand der Szenariokonfiguration in der Lage sein, die Modelle entsprechend zu deployen und zu konfigurieren. Die Ergebnisse der Simulation sollen zunächst von jedem Modell zum Ende des Szenarios an einer zentralen Stelle abgelegt werden, so dass diese vom Nutzer abgerufen werden können.

Das Framework soll ein Zeitmodell unterstützen, das eine variable Zeitbasis umsetzt. Die Modelle sollen sich dabei in einem synchronen Zustand befinden und jeweils diskrete Zeitschritte durchführen, die eine um einen Faktor veränderte Realzeit repräsentieren. Die Synchronität der Komponenten soll durch ein verteiltes Werkzeug des Frameworks sichergestellt und überprüft werden.

Im Anwendungsfall OS4ES soll die Stabilität der Registry unter großer Last überprüft werden. Dazu werden Simulationen mit großen Anzahlen von DERs und Aggregatoren durchgeführt. Um den Ressourcenaufwand dementsprechend planen zu können soll das Framework skalierbar entworfen werden, sodass der Ressourcenbedarf annähernd proportional zum Simulationsaufwand steigt.

Die Anforderungen werden nach dem MoSCoW-Prinzip (vgl. Coley Consulting) priorisiert. Tabelle 3.1 fasst die Anforderungen an die Plattform zusammen und weist ihnen Prioritäten zu.

Tabelle 3.1.: Anforderungen an die Simulationsplattform

Bezeichnung	**Priorität**
Abstraktion der Software von der Hardware	MUST
Verteilung des Simulationssystems	MUST
Unterstützung von Modellen verschiedener Programmiersprachen	SHOULD
Front-End-Komponente	COULD
Zentrale Sammlung der Simulationsergebnisse	SHOULD
Synchrone Simulation in diskretem Zeitmodell	MUST
Skalierbarkeit des Ressourcenbedarfs	SHOULD

3.3. Anforderungen an die Kommunikation

Wie durch die Beschreibung in Abschnitt 3.1 und die Anforderung eines verteilten Systems deutlich wird, sind die Kommunikation zwischen den Modellen, sowie die Kommunikation des Frameworks mit den Modellen, essentiell für ein funktionsfähiges System. Dazu werden nun zunächst die Kommunikationswege innerhalb des Marktes und daraufhin die benötigte Kommunikation des Frameworks mit den Modellen analysiert.

Der Austausch von Nachrichten in diesem Simulationsframework wird über die Netzwerkinfrastruktur und falls nötig auch über das Internet durchgeführt. Die Schnittstellen der Modelle sollen als Web-Services ausgeführt werden und daher das Hypertext Transfer Protocol (HTTP) genutzt werden. Dabei soll die Kommunikation für die Modelle transparent ausgeführt sein.

Anhand von Abbildung 3.8 lassen sich die in Tabelle 3.2 dargestellten Kommunikationswege ableiten. Wie bereits in Abschnitt 3.1 erwähnt sollen keine weiteren Energiemärkte simuliert werden, weswegen die Strommarkt-Kommunikation vernachlässigt werden kann. Die Bereitstellung von Abhängigkeiten wie des Wetterdienstes ist Teil des Szenarios. Ein Wetterdienst zum Beispiel würde wie die DER-Simulationen simulationsinterne Nachrichten austauschen. Da vier der Kommunikationswege den Austausch von Nachrichten zwischen teilnehmenden Modellen der Simulation beschreiben, werden diese gemeinsam mit IntraSim betitelt.

Um den Modellen Steuerungsdaten und Steuerbefehle zukommen lassen zu können wird außerdem eine Kommunikationsstrecke zwischen den Framework und den Modellen benötigt. Für eine saubere Trennung der Kommunikation zwischen den Modellen und den Steu-

Tabelle 3.2.: Kommunikationsstrecken im OS4ES

Teilnehmer 1	Teilnehmer 2	Kurzbezeichnung
Wetterdatenserver	DER-System	Wetterdienst
Aggregator	Energiebörse	Strommarkt
DER-System	Registry	IntraSim
Registry	Aggregator	IntraSim
Aggregator	DER-System	IntraSim

ernachrichten des Frameworks, sollen die Schnittstellen für diese Nachrichten ebenfalls getrennt werden.
Um es den Teilnehmern zu ermöglichen eine Kommunikation mit den entsprechenden Gegenstellen aufzubauen, soll eine Mechanismus eingeführt werden, der die Netzwerkadressen (IPs oder Domain-Namen) mitteilt. Ein Käufer oder Verkäufer muss z.B. wissen, wie die Registry erreicht werden kann.

Mithilfe von HTTP-Statuscodes soll das Framework den Status der getätigten Kommunikation überwachen und gegebenenfalls auf eine gescheiterte Übertragung reagieren.

Tabelle 3.3.: Anforderungen an die Kommunikation

Bezeichnung	Priorität
Schnittstellen über Webservices	MUST
Transparente Kommunikation	SHOULD
Schnittstelle für Steuerbefehle	MUST
Schnittstelle für Nachrichten innerhalb des Systems	MUST
Kommunikation der benötigten Endstellen	MUST

Anhand der beschriebenen Eigenschaften können die in Tabelle 3.3 aufgelisteten Anforderungen an die Kommunikationsmittel abgeleitet werden. Diesen werden ebenfalls Prioritäten zugewiesen.

Es wurden nun die Anforderungen an die Simulationsplattform sowie an die Kommunikation im Simulationsframework erhoben. Die bestehenden Simulationslösungen bieten für diesen Anwendungsfall keine optimalen Voraussetzungen. Daher wird hier der Versuch unternommen ein Simulationsframework zu entwickeln das alle gestellten Anforderungen erfüllt. Nachfolgend soll im nächsten Kapitel das Design und die Architektur dieses Frameworks dargestellt und erläutert werden.

4. Architektur und Design des Frameworks

In diesem Kapitel soll die Architektur eines Frameworks für die Cloud-basierte Simulation von Szenarios auf heterogener Infrastruktur ahand der vorhergehenden Analyse in Kapitel 3 und den daraus folgenden Anforderungen beschrieben werden. Zentraler Aspekt ist dabei die Beschreibung der beteiligten Komponenten und der Kommunikationsinfrastruktur. Dazu wird zunächst die spezifische Architektur des Simulationsframeworks beschrieben und daraufhin die für das Simulationsmanagement zuständigen Komponenten erläutert.

Für die Durchführung einer Simulation muss zunächst ein Szenario bestimmt werden, das Simuliert werden soll. Aus diesem Szenario geht hervor, welche Modelle für die verteilte Simulation in welcher Zusammenstellung benötigt werden. Diese Modelle müssen daraufhin parametriert und gemanaged werden. Nach der Durchführung der Simulation müssen die Simulationsergebnisse gesammelt und bereitgestellt werden.
Der beschriebene und in Abbildung 4.1 dargestellte Ablauf, wird mithilfe von Komponenten des Simulationsframeworks umgesetzt. Es gibt also eine Mensch-Maschine-Schnittstelle über die ein zu simulierendes Szenario bestimmt wird. Anhand des eingestellten Szenarios wird automatisch eine Szenariokonfiguration erstellt. Das Framework startet dann die entsprechenden Modelle, konfiguriert diese und führt die Simulation aus. Wenn die Simulation abgeschlossen ist, werden die Ergebnisse gesammelt und dem Nutzer zur Verfügung gestellt.

Der Simulationsablauf besteht dabei aus vier Phasen: Der Szenario-Eingabe, der Konfiguration der Modelle, der Durchführung der Simulation und dem Sammeln der Ergebnisse. Das Framework stellt Komponenten bereit, mit denen diese vier Phasen realisiert werden können.
Anhand der Analyse wurde die in Abbildung 4.2 dargestellte Umgebung entworfen. Entsprechend der Anforderungen liegt zwischen der physikalischen Schicht und den Anwendungen eine Abstraktionsschicht. Die Architektur des Simulationsframeworks wird nachfolgend in Abschnitt 4.1 und die Komponenten in den Abschnitten 4.2 und 4.3 beschrieben.

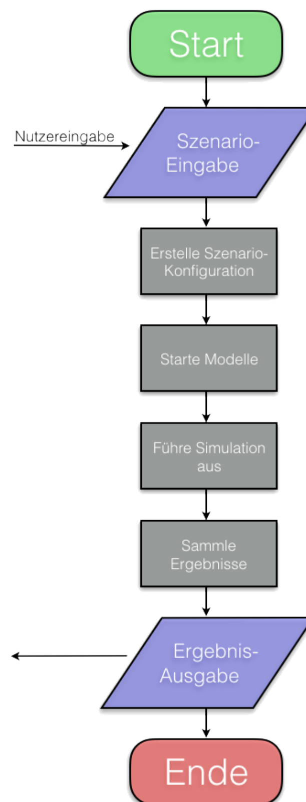

Abbildung 4.1.: Programmablaufplan der Simulation

4.1. Beschreibung der Framework-Architektur

In diesem Abschnitt soll zunächst die Umsetzung der Abstraktion zwischen dem Simulationsframework und der darunterliegenden Infrastruktur erläutert werden. Diese wird mithilfe der in Abschnitt 2.2.3 beschriebenen Docker-Virtualisierungsplattform umgesetzt. Die genaue Anwendung wird im nächsten Abschnitt beschrieben. Nach der Abstraktionsschicht wird dann die Architektur des Frameworks beschrieben, bevor in den folgenden Abschnitten genauere Beschreibungen der beteiligten Komponenten erfolgen.

Um die Framework-Architektur zu verdeutlichen wird diese in zwei Ebenen eingeteilt. Zum einen gibt es die Infrastruktur-Ebene und zum anderen die Framework-Ebene. Die Architektur der Infrastruktur-Ebene wird in Abbildung 4.3 dargestellt. Verfügbare physische Maschinen werden mit dem Docker-Agenten ausgestattet und zu einer Rancher-Infrastruktur hinzugefügt. Die Rancher-Software läuft dabei als Container in der Docker-Umgebung. Rancher dient dabei als Management-Komponente für die beteiligten Hosts und bildet damit eine Cloud-Infrastruktur, die vom Simulationsframework genutzt werden kann. Durch die Funktionalität von Rancher wird eine automatische Lastverteilung je nach Auslastung der

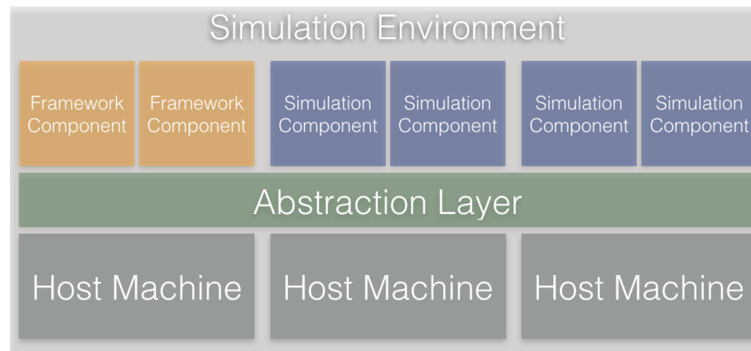

Abbildung 4.2.: Schichtenmodell des Simulationsframeworks

Hosts ermöglicht. Dabei werden die zu startenden Container automatisch so auf die Hosts verteilt, dass keinem Host die Ressourcen ausgehen. Dies ist aufgrund der technischen Limitierung nur bis zur maximalen Auslastung aller Hosts möglich. Die genaue Verwendung der Docker-Plattform und der Rancher Funktionalitäten zum Management der Cloud werden in Abschnitt 4.2.1 beschrieben.

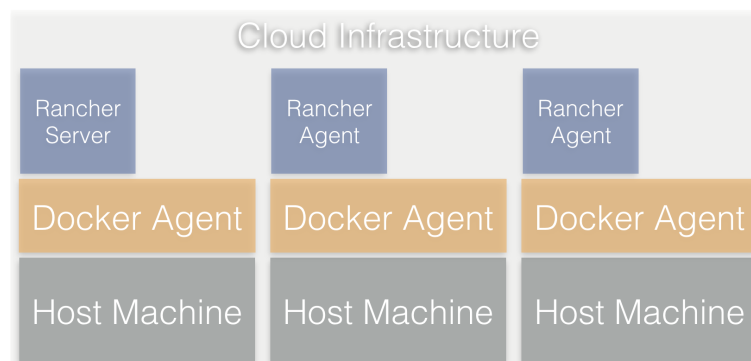

Abbildung 4.3.: Architektur der Infrastruktur-Ebene

Auf dieser Cloud-Infrastruktur die durch Docker von der Hardware abstrahiert wurde, werden dann die Framework-Komponenten als Docker-Container aufgesetzt. Dort setzt die, in Abbildung 4.4 dargestellte, Framework-Ebene auf. Alle im Framework vorhandenen Komponenten werden als Docker-Container ausgeführt und bieten daher eine hohe Portabilität.

Anhand von Abbildung 4.4 sind die zwei Ebenen der durchzuführenden Simulation erkennbar: Die Konfigurationseben und die Simulationsebene. In der Konfigurationsebene wird die Szenario-Konfiguration durchgeführt und die beteiligten Komponenten gestartet. Die Komponenten dieser Ebene laufen also dauerhaft, während die Komponenten der Simulationsebene ad hoc gestartet werden.
Über das Front-End wird die Szenariokonfiguration vorgenommen. Diese Komponente muss

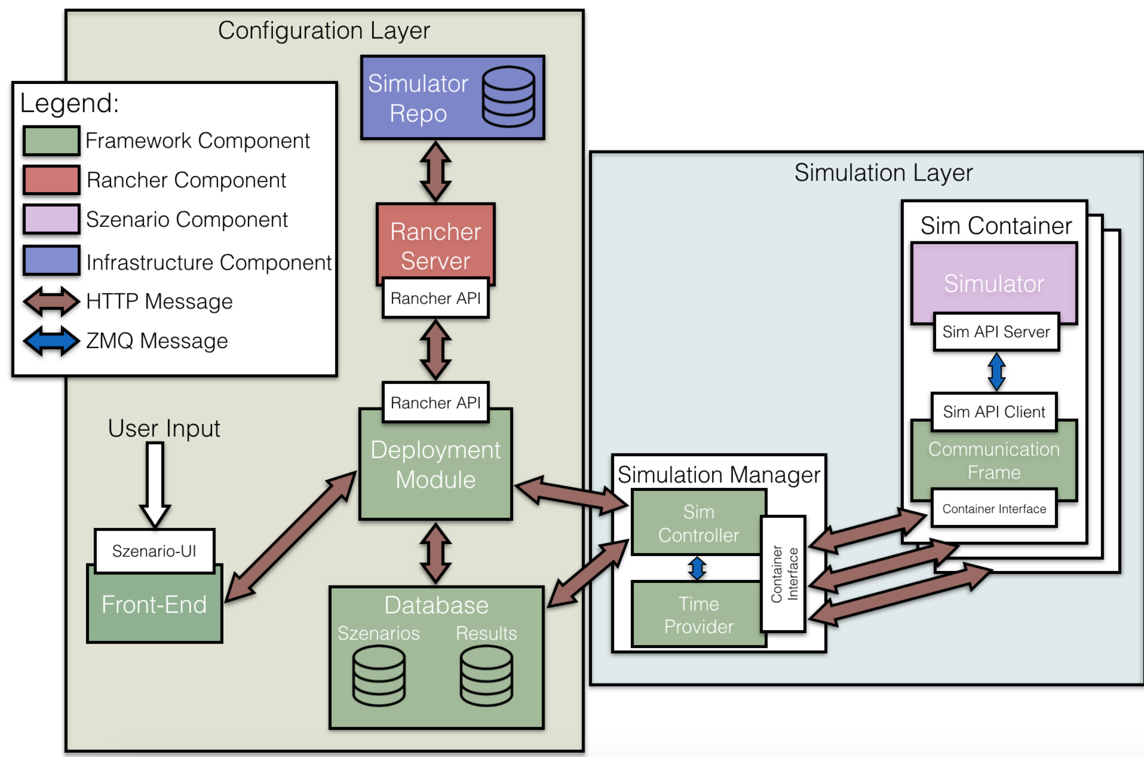

Abbildung 4.4.: Architektur der Framework-Ebene

lokal auf einem Nutzer-Computer ausgeführt werden. Der Nutzer gibt die gewünschte Konfiguration über ein Command Line Interface (CLI) ein. Anhand der angelegten Konfiguration erstellt das Front-End eine Konfigurationsdatei und sendet diese an das Deployment-Modul. Das Deployment-Modul empfängt und liest die Konfigurationsdatei. Die enthaltene Konfiguration wird in der Datenbank abgelegt. Entsprechend der angelegten Konfiguration beauftragt das Deployment-Modul den Rancher-Server die benötigten Komponenten (alle ausgeführt als Docker Container) zu starten. Die Images [1] liegen dabei im Simulatoren-Repository. Auf dieses greift der Docker Server zu und startet die Komponenten.

Zusätzlich zu den benötigten Simulatoren wird ein Simulationsmanager gestartet. Über diesen werden die Modelle konfiguriert und die Durchführung der Simulation gesteuert. Nachdem die Simulation abgeschlossen ist und die Ergebnisse gesammelt wurden beendet das Deployment-Modul die an der Simulation beteiligten Komponenten und stellt dem Nutzer die gewonnen Ergebnisse bereit.

In Abbildung 4.4 wurde eine Farbkodierung der Komponenten und Kommunikationsstrecken eingeführt. Die Farbe der Komponenten gibt dabei an, ob es sich um Komponenten handelt

[1] Docker Container werden aus Docker Images erzeugt

die vom Simulationsframework zur Verfügung gestellt werden, herstellerspezifische Produktlösungen sind, zur Infrastruktur der Simulationsumgebung gehören oder von Ingenieuren und Entwicklern bereitgestellte Simulatoren sind.
Grün dargestellte Elemente werden vom Framework bereitgestellt. Rote Komponenten sind herstellerspezifische Lösungen und blaue Komponenten gehören zur verwendeten Infrastruktur. In lila sind die Modelle dargestellt die mithilfe des Kommunikationsrahmens in das Simulationsframework eingebunden werden.

4.2. Komponenten der Konfigurationsschicht

Nachdem bisher die Gesamtarchitektur des Frameworks beschrieben wurde, sollen in diesem nun die Komponenten beschrieben werden, die in der Konfigurationsschicht (siehe Abbildung 4.4) angesiedelt sind. Dies sind zum einen der Rancher Server, wobei auch die Docker Plattform erläutert wird, das Front-End-Modul sowie das Deployment-Modul. Dabei wird zunächst ein kurzer Überblick über die Funktion der einzelnen Komponenten gegeben, bevor dann eine genaue Erläuterung dieser erfolgt.

4.2.1. Docker-Plattform und Rancher

Um eine Abstraktion der Hard- von der Software zu ermöglichen wird die Docker-Plattform als Lösung zur Anwendungsvirtualisierung verwendet. Durch die Verwendung von Docker wird die in Abbildung 4.2 dargestellte Architektur umgesetzt. Dazu ist es nötig, dass der Docker-Agent auf jeder physischen Maschine installiert werden muss, die an der Simulationsinfrastruktur beteiligt ist. Hierbei ist es unerheblich ob die Infrastruktur homogen oder inhomogen ist. Es können also auch Maschinen unterschiedlicher Bauart und mit unterschiedlichen Betriebssystemen in einer Cloud verwendet werden. Docker Distributionen sind für Microsoft Windows, Apple macOS und Linux Betriebssysteme verfügbar. Um nun die noch spareraten Maschinen in eine Infrastruktur zu vereinen wird die Rancher-Umgebung verwendet.
Auf jeder Host-Maschine (im Folgenden Host genannt) wird dafür der Rancher-Agent installiert. Einer der Hosts erhält dabei den Rancher-Server. Dieser stellt die Verwaltungsschnittstelle zur Verfügung und verwaltet die Hosts.

Angesprochen wird Rancher vom Framework über die Rancher API[2]. Über diese werden vom Simulationsframework Container gestartet und gestoppt. Sollen neue Container gestar-

[2] https://docs.rancher.com/rancher/v1.5/en/api/v2-beta/

tet werden, geschieht dies indem Rancher aus einem im Repository abgelegten Image einen neuen Container erstellt. Die benötigten Images der Module und Simulatoren müssen daher zum Zeitpunkt der Simulationsdurchführung in dem Repository vorhanden sein.

4.2.2. Front-End

Um dem Nutzer eine Möglichkeit zu bieten eine Simulation zu konfigurieren und auszuführen, muss eine Schnittstelle geschaffen werden, die diese Funktionalität bietet. Dies wird hier mithilfe einer Front-End-Komponente ermöglicht. Diese soll im Folgenden beschrieben werden.

Das Front-End ist die System/Nutzer Schnittstelle des Simulationsframeworks. Über dieses wird die Szenariakonfiguration in das System eingegeben. Das Front-End wird lokal auf einer Anwendermaschine ausgeführt. Dabei wird die Konsole zur Dateneingabe verwendet.
Der grundlegende Programmablauf wird in Abbildung 4.5 dargestellt. Das Front-End nimmt Nutzereingaben entgegen und prüft ob diese Konfigurationsformationen enthalten. Handelt es sich um Konfigurationsinformationen werden diese in einem Konfigurationsarray abgelegt. Sind keine Konfigurationsinformationen enthalten wird geprüft ob es sich um den Startbefehl der Simulation handelt. Ist die Szenariokonfiguration abgeschlossen und der Startbefehl gegeben, wird aus dem Array, das die Szenariokonfiguration enthält, eine Konfigurationsdatei. Diese wird über HTTP an das Deployment-Modul übergeben. Ist die Simulation abgeschloßen, sendet das Deployment-Modul die Simulationsergebnisse. Das Front-End gibt diese dann als Konsolenausgabe aus.
Sollte innerhalb der Simulation ein Fehler auftreten der durch Simulatoren verursacht wurde, so wird von diesen eine Fehlernachricht mitgegeben die am Front-End-Modul ausgegeben wird.
Die Formatierung der Konfigurationsdatei, sowie das Front-End-UI werden ausführlich in den Abschnitten 4.4.1 und 4.4.2 beschrieben.

4.2.3. Deployment-Modul

Um eine Umsetzung eines geplanten Szenarios in eine ausführbare Simulation zu ermöglichen, wird eine Komponente benötigt, die die Fähigkeit besitzt Nutzereingaben in eine solche zu übersetzen.
Das Deployment-Modul ist zuständig für die Umsetzung der Informationen einer Szenariokonfiguration in ein ausführbares Szenario mit allen beteiligten Komponenten und Modulen. Es nimmt die Szenariokonfiguration vom Front-End entgegen und setzt das enthaltene Szenario um. Damit ist das Deployment-Modul, als Verbindungsglied zwischen Infrastruktur- und Framework-Ebene, der zentrale Dreh- und Angelpunkt des Simulationsframeworks.

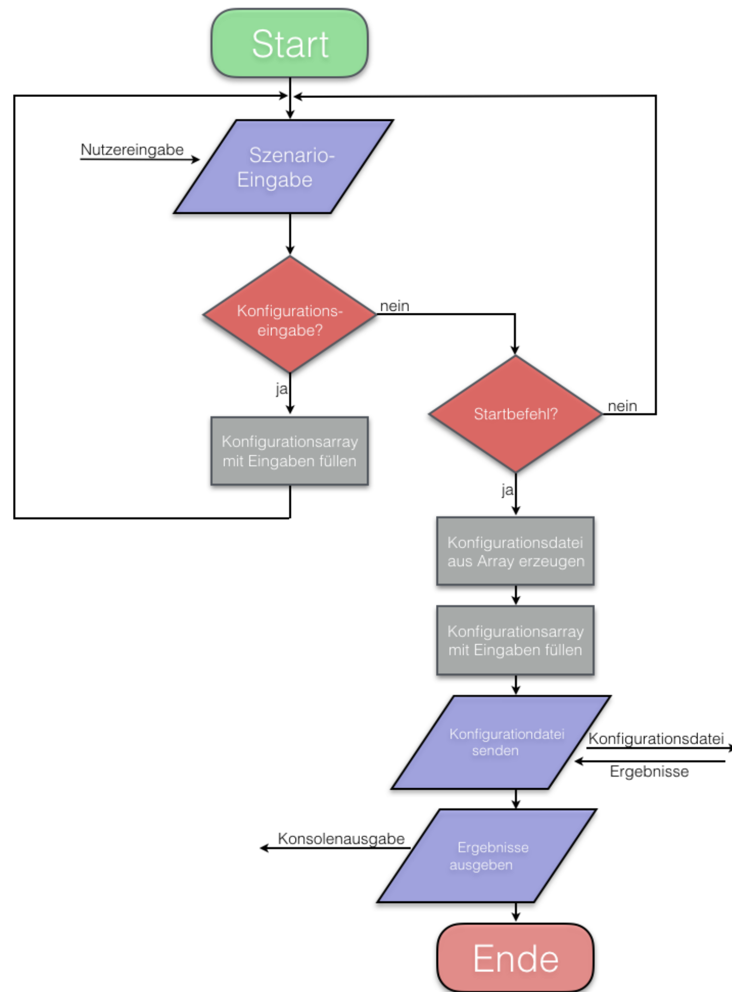

Abbildung 4.5.: Funktionslogik der Front-End Komponente

Zur Veranschaulichung der Tätigkeiten und Zusammenhänge des Deployment-Moduls innerhalb des Simulationsframeworks werden diese in Abbildung 4.6 dargestellt.
Die Einzelschritte der in der Abbildung dargestellten Vorgänge werden nachfolgend beschrieben. Die beschriebenen Tätigkeiten sind in der Abbildung, sowie im Text mit Nummern versehen.

Soll eine Simulation durchgeführt werden, muss eine Front-End-Instanz gestartet werden. Wenn diese eine initiale Verbindung zum Deployment-Modul aufbaut (1), fragt das Deployment-Modul eine aktuelle Liste der verfügbaren Modelle beim Rancher Server an (2) und sendet diese an das Front-End (3), um dem Nutzer diese zur Verfügung zu stellen.

Sendet das Front-End-Modul die im vorigen Abschnitt erwähnte Konfigurationsdatei (4), wird die Datei eingelesen und daraus alle notwendigen Informationen extrahiert, die benötigt wer-

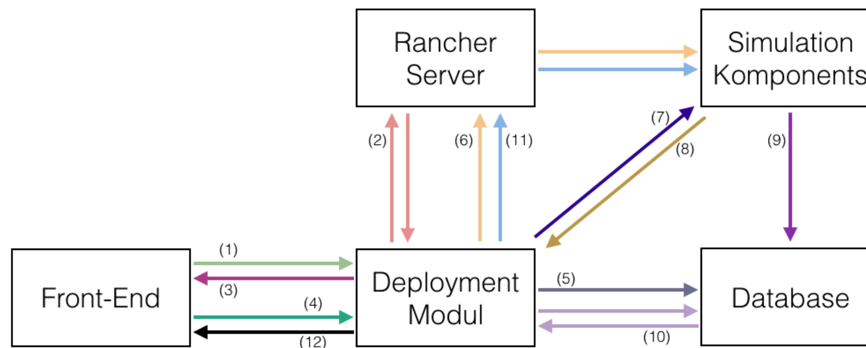

Abbildung 4.6.: Funktionslogik der Front-End Komponente

den um das beschriebene Szenario umsetzen zu können. Die Szenariokonfiguration wird für spätere Nutzung im Datenbankmodul abgelegt (5).
Im nächsten Schritt startet das Deployment-Modul die benötigten Simulationskomponenten über den Rancher Server (6). An den im folgenden Abschnitt beschriebenen Simulationsmanager sendet das Deployment-Modul die Konfigurationsdaten des Szenarios (Die zugehörigen Adressen der Registry und anderen Abhängigkeiten) (7).

Ist eine Simulation ausgeführt worden, wird das Deployment-Modul vom Simulationsmanager informiert (8). Dieser legt die Ergebnisdaten in der Datenbank ab (9). Dieses ruft daraufhin die Ergebnisse aus der Datenbank ab (10), beendet die Simulationskomponenten über den Rancher Server (11) und sendet die Simulationsergebnisse an das Front-End-Modul (12).

4.3. Komponenten der Simulationsschicht

Die bis zu diesem Punkt beschriebenen Komponenten des Simulationsframeworks gehörten zur Konfigurationsschicht des Frameworks und daher zu den dauerhaft laufenden Komponenten. Die folgenden Komponenten gehören zur Simulationsschicht und werden nur zur Durchführung einer Simulation gestartet. Wenn eine Simulation durchgeführt wurde und die Ergebnisse gesammelt, werden die Komponenten wieder gestoppt.
In der Simulationsschicht liegt der Simulationsmanager, der für die Konfiguration der Modelle, die Synchronität der Modelle (und Bereitstellung der globalen Zeit), den Start und Stopp der Simulation und das Sammeln der Ergebnisse zuständig ist. Neben dem Simulationsmanager liegen außerdem die Simulationscontainer in der Simulationsschicht. Diese enthalten neben den Simulatoren noch einen Kommunikationsrahmen der eine transparente Kommunikationsinfrastruktur für die Simulatoren bereitstellt.

4.3.1. Simulationsmanager

Um einen geregelten Ablauf und gemeinsames Simulationsergebnis einer verteilten Simulation zu ermöglichen ist es notwendig, dass eine verwaltende Komponente in der Lage ist den Simulatoren Anweisungen zu geben, Konfigurationsinformationen bereitzustellen und das Gesamtsystem zu kontrollieren. Diese Aufgaben erfüllt in diesem Simulationsframework der Simulationsmanager.

Der Simulationsmanager besteht wiederum aus zwei Subkomponenten: Dem Simulationscontroller und dem Time Provider. Der Simulationscontroller ist zuständig für den Ablauf und die Konfiguration der Simulation und ihrer Komponenten. Der Time Provider ist zuständig dafür die Synchronität der Simulationskomponenten sicherzustellen und die Simulationszeit bereitzustellen.

Simulationscontroller

In Kapitel 4.2.3 wurden einige Aufgaben erwähnt, die vom Simulationsmanager ausgeführt werden. Diese Aufgaben sind innerhalb des Simulationsmanagers dem Simulationscontroller zugewiesen. Um einen gewissen Überblick zu erhalten mit welchen Komponenten des Frameworks bzw. der Simulation der Simulationscontroller interagiert wird die Architektur des Simulationsmanagers inklusive der Kommunikationswege in Abbildung 4.7 dargestellt. Die Aufgaben des Simulationscontrollers bestehen aus der Konfiguration der Simulatoren, des Managements des Simulationsablaufes, sowie der Sammlung der Simulationsergebnisse und Speicherung dieser in der Datenbank.

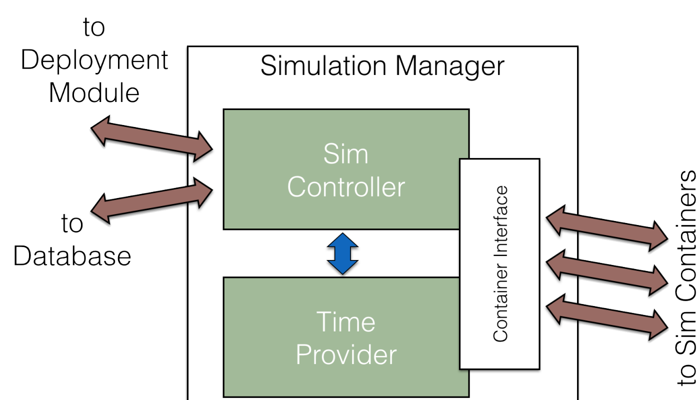

Abbildung 4.7.: Architektur und Kommunikationswege des Simulationsmanagers

Wird eine Simulation beauftragt, startet das Deployment-Modul die benötigten Komponenten über den Rancher Server, darunter auch den Simulationsmanager. Sobald dieser gestartet ist, fragt der Simulationscontroller die Konfigurationsdaten für die anstehende Simulation beim Deployment-Modul an. Dieses antwortet mit den Konfigurationsdaten. Nun hat der Simulationscontroller alle Informationen die notwendig sind um die Simulation auszuführen. Er sendet daraufhin die Adressen bzw. Domain Names der Registry, sowie der Abhängigkeiten, an die Simulatoren. Dabei wird einem Verkäufersimulator immer die zu seiner Zone passende Registry-Adresse übermittelt. Zusätzlich wird die im folgenden Abschnitt näher beschriebene Timebase sowie der Simulationszeitraum übertragen. An das Zeitmodul sendet der Simulationscontroller, nachdem alle Simulatoren auf die Konfigurationsdaten mit „Ready“ geantwortet haben, den Simulationszeitraum und die Timebase zusammen mit einem Startbefehl.

Meldet das Zeitmodul dass alle Zeitschritte erfolgreich durchlaufen sind, sendet der Simulationscontroller einen Result-Request an alle Simulatoren, welche daraufhin mit ihren Ergebnissen antworten. Sollte die Simulation nicht erfolgreich durchlaufen worden sein (Gründe hierfür werden in Abschnitt 4.3.1 aufgezeigt) so sendet der Controller ebenfalls einen Result-Request, meldet allerdings die fehlerhafte Ausführung der Simulation an das Deployment Modul. Bei erfolgreicher Simulationsdurchführung sendet der Simulationscontroller dementsprechend eine Erfolgsnachricht an das Deployment-Modul.
Die von den Simulatoren eingesammelten Ergebnisse werden nachfolgend, für das Deployment-Modul abrufbar, in der Datenbank abgelegt.

Im nächsten Abschnitt wird die zweite zum Simulationsmanager gehörende Komponente beschrieben: Das Zeitmodul.

Zeitmodul

In diesem Simulationsframework ist ein diskretes Zeitmodell, mit fixer Größe der Zeitschritte, vorgesehen. Dabei kann allerdings die zeitliche Auflösung der Simulationsergebnisse durch Wahl der Zeitbasis angepasst werden. Die Wahl der Zeitbasis gibt hier an um welchen Zeitraum die Uhr pro Schritt (Step) voranschreitet. Das Zeitmodul lässt die Uhr dabei so schnell voran schreiten, wie es die beteiligten Simulatoren zulassen, um eine kurze Simulationslaufzeit zu ermöglichen. Da sich die Simulatoren in einem synchronen Zustand befinden sollen (alle Simulatoren befindet sich am selben Zeitpunkt der Simulationsuhr) bestimmt der Simulator mit der längsten Bearbeitungsdauer die Schrittgeschwindigkeit.
Es besteht die Möglichkeit dass die Bearbeitungszeiten von Simulatoren zwischen verschiedenen Schritten variieren, die Zeitschritte der Simulationsuhr sind daher auf einer realen Zeitachse nicht äquidistant. Um dies zu verdeutlichen stellt Abbildung 4.8 exemplarisch die

Bearbeitungszeiten vom Simulatoren über die Simulationsschritte dar. Hier ist erkennbar dass der Simulator, der in einem Schritt die längste Bearbeitungsdauer besitzt, die Schrittdauer bestimmt. Die Simulationsuhr schreitet dabei allerdings kontinuierlich in derselben Schrittgröße voran.

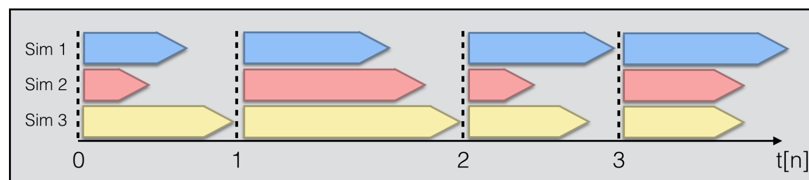

Abbildung 4.8.: Diskrete Zeitschritte der Simulation

Der Zusammenhang zwischen den Simulationsschritten n und der Simulationsuhr $t_s[n]$ wird durch folgende Gleichung gegeben, wobei t_{Start} ermöglicht, durch die Addition einer Startzeitkonstanten, eine Abbildung auf einen Realzeitraum durchzuführen.

$$t_s[n] = t_{Start} + (n \cdot timebase)$$

Die Variable $timebase$ entspricht dabei der gewählten Zeitbasis im Szenario. Sie gibt an um welche Zeit die Simulationsuhr pro Simulationsschritt voranschreitet. Die Ausgabe dieser Funktion wird in UNIX-Zeit angegeben, einem Zeitformat dass die Zeit in Sekunden angibt, die seit dem „Epoch“, 00:00:00 Uhr am 01.01.1970, vergangen sind. Mehr Informationen dazu können unter (IEEE und The Open Group, A.4.16) eingesehen werden. Die Zeitbasis ist daher als Zeit in Sekunden definiert.

Ein solcher Synchronisationsmechanismus wird als Barrier-Synchronisation bezeichnet. Dabei können Prozesse nur bis zu einer festgelegten Barriere laufen und stoppen dort, bis jeder beteiligte Prozess die Barriere erreicht hat. Daraufhin darf die Barriere kollektiv überschritten werden. Dieser Mechanismus wird in (Mellor-Crummey und Scott, 1991, S. 11 ff) unter der Bezeichnung der zentralisierten Barriere beschrieben. In diesem Simulationsframework werden die Barrieren durch die Simulationsschritte dargestellt. Haben alle Simulatoren die Arbeit des aktuellen Zeitschrittes durchgeführt hebt das Zeitmodul die Barriere auf und lässt die Simulatoren in den nächsten Simulationsschritt übergehen.

Dieser Ablauf, dargestellt in Abbildung 4.9, wird vom Zeitmodul umgesetzt indem dieses einen Step-Befehl an die Simulatoren sendet. Diese antworten mit einer Step-Finished-Nachricht. Haben alle Simulatoren den Schritt beendet (und die Step-Finished-Nachricht gesendet), wird der nächste Step-Befehl gesendet. Alle Nachrichten die während der Bearbeitungszeit eines Schrittes bei den Simulatoren eintreffen erhalten denselben Zeitstempel und werden als parallel eingetroffen betrachtet.

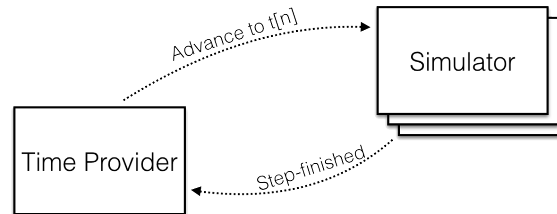

Abbildung 4.9.: Stepper-Funktion des Zeitmoduls

Da der Ausfall eines Simulators nicht auszuschließen ist, muss ein Deadlock vermieden werden der auftreten würde wenn sich nicht alle Simulatoren mit einer Step-Finished-Nachricht melden. Dies wird hier sichergestellt, indem eine Alive?-Nachricht an den betroffenen Simulator gesendet wird, nachdem ein Timeout-Zeitintervall verstrichen ist. Antwortet dieser mit einer Alive!-Nachricht, startet das Timeout-Intervall erneut. Sollte dieses erneut verstreichen wird der Simulator aus dem Szenario ausgeschlossen. Handelt es sich bei dem nicht reagierenden Simulator um eine Registry-Einheit, wird die Simulation als fehlerhaft beendet, da ein Ausfall dieser einen Ausfall von mehreren Käufer- und Verkäufersimulatoren zur Folge hätte.

Direkt vor dem Beginn einer Simulation erhält das Zeitmodul einer Nachricht vom Simulationscontroller, die Informationen über des Simulationszeitraum und die Timebase enthält. Mit dieser wird der Simulationsstart initiiert. Das Zeitmodul sendet dann den Step-Befehl zu Zeitschritt 0. Damit beginnt die Simulation.
Nachdem alle Zeitschritte durchlaufen sind sendet das Zeitmodul eine Sim-Finished-Nachricht an den Simulationscontroller um zu signalisieren, dass die Simulation erfolgreich durchlaufen wurde.

4.3.2. Kommunikationsrahmen

Um Simulatoren in eine verteilte Simulation einzubinden müssen diese standardisierte Schnittstellen zur Verfügung stellen. Um die Simulatoren einfacher gestalten zu können und Entwicklern eine leicht implementierbare API zur Verfügung stellen zu können wurde der Kommunikationsrahmen entwickelt. Dieser stellt das Bindeglied zwischen den Simulatoren und der Kommunikationsinfrastruktur dar, indem er an der Simulator-gerichteten Seite eine leicht zu implementierende API zur Verfügung stellt und auf der Framework-gerichteten Seite die nötigen Webservices und Schnittstellen besitzt um eine effiziente und verlässliche Kommunikation zu ermöglichen.

Simulatoren in diesem Simulationsframework werden, wie alle Komponenten, in einem Docker-Container betrieben. Für Simulatoren wird dabei eine Containervorlage bereitgestellt, der Simulationscontainer. Die Architektur eines Simulationscontainers wird in Abbildung 4.10 dargestellt. Der Simulator und der Kommunikationsrahmen kommunizieren über die Sim-API, die in Abschnitt 4.4.3 beschrieben wird. Um Simulationsentwicklern eine API in einer Mehrzahl von Programmiersprachen anbieten zu können, wird die Verbindung zwischen Kommunikationsrahmen und Simulator per standardisierten ZMQ[3] Messages ausgeführt. So kann die Simulator-Seite der API in mehreren Language-Bindings bereitgestellt werden.
Der Kommunikationsrahmen bietet Framework-seitig Webservices an, über die die Kommunikation zwischen den Simulatoren und die Simulationssteuerung erfolgt. Über verschiedene URLs auf dem Docker Container lassen sich verschiedene Endpunkte ansprechen.

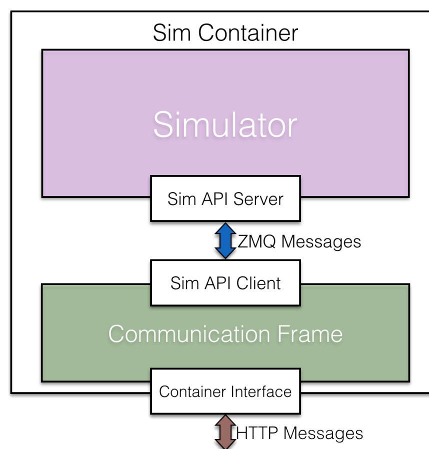

Abbildung 4.10.: Stepper-Funktion des Zeitmoduls

4.4. Beschreibung der Kommunikationsschnittstellen

Nachdem in den vorigen Abschnitten die Komponenten Module des Simulationsframeworks erläutert wurden, sollen hier nun die Schnittstellen beschrieben werden, die diese Komponenten zur Kommunikation nutzen und bereitstellen. Ebenfalls sollen die Datenmodelle der übertragenen Informationen dargestellt werden. Durch eine passende Auslegung der Schnittstellen soll die Implementation der Komponenten modular möglich sein und direkte Kompatibilität sicherstellen.

[3] http://zeromq.org

4.4.1. Datenmodelle

In diesem Abschnitt sollen die wichtigsten, im Simulationsframework für Datenübertragungen genutzten Datenmodelle beschrieben und erläutert werden. Es werden die verwendeten Nachrichten beschrieben und die Datenfelder dieser Nachrichten aufgelistet. Die beschriebenen Nachrichten beinhalten die Simulatoren-Liste, die Konfigurationsnachrichten des Front-Ends, des Deployment-Moduls und des Simulationscontainers. Außerdem werden die Nachrichten für Step-Befehle und IntraSim-Kommunikation erläutert. Eine klare Definition der verwendeten Nachrichten ist notwendig um eine saubere Implementation der Framework-Komponenten zu ermöglichen. Daher werden neben der Definition der Nachrichtentypen jeweils auch Beispiele dieser dargestellt, um den Inhalt der Felder zu verdeutlichen.

Bei jeder Kommunikationsstrecke zwischen verschiedenen Komponenten dieses Frameworks wird das JSON-Datenformat verwendet.
JavaScript Object Notation (JSON) ist ein Datenformat in Textform für den Datenaustausch zwischen Anwendungen. Die Formatierung ist menschenlesbar und daher von diesen interpretierbar. Das JSON-Datenformat ist unabhängig von Programmiersprachen und kann, wenn ein Parser verfügbar ist, in jeder Programmiersprache verwendet werden. (Bray (2014))
Aufgrund dieser Tatsachen und der großen Verbreitung als Datenformat für Webservices wird JSON für das hier entworfene Simulationsframework verwendet.
Die Beispiele der verwendeten Nachrichtentypen sind in JSON-Notation aufgeführt und veranschaulichen auch die Lesbarkeit des Datenformats. Wobei die Daten innerhalb der Module als Datenobjekte behandelt werden müssen und daher auch Datentypen zur Beschreibung des erwarteten Inhalts der Felder angegeben werden.

Simulatoren-Liste

Die Simulatoren-Liste ist ein Nachrichtenformat das zwischen dem Front-End-Modul und dem Deployment-Modul ausgetauscht wird. Sie beinhaltet Informationen über für eine Simulation verfügbare Simulatoren bzw. deren Images.

Damit der Nutzer eine korrekte Szenario-Eingabe über das Front-End-UI vornehmen kann ist es notwendig, dass dieser darüber informiert wird aus welchen Simulatoren das Szenario zusammengestellt werden kann. Daher stellt das Deployment-Modul eine Liste aller verfügbaren Simulatoren-Images zusammen und überträgt diese an das Front-End-Modul. Die Images werden dabei unterteilt in Registries, Käufer, Verkäufer und Abhängigkeiten.

Tabelle 4.1 listet die Datenfelder der Simulatoren-Liste auf. Die aufgeführte ID ist eine Containerweit eindeutige Identifikationsnummer zum Zwecke der Rückverfolgbarkeit im Fehlerfall. Auf diese folgt ein Array aus Registry-Images. Diese werden mit Namen, Szenario-Typ und Datum der letzten Änderung angegeben. Dieser Szenario-Typ gibt an für welche Marktumgebung diese Images erstellt wurden. Hier sollten von Entwicklern eindeutige Szenario-Namen zur Identifikation vergeben werden. Im Falle des OS4ES-Projekts werden könnte „OS4ES" als Szenario-Typ gewählt werden. Dies würde dann sicherstellen dass alle Images mit Typ OS4ES miteinander kompatibel sind und in einem Szenario verwendet werden können.
Die Arrays für Käufer, Verkäufer und Abhängigkeiten sind analog zu dem der Registries aufgebaut. Die Namen der Docker-Images sind eindeutig und werden nachfolgend zur Auswahl der Simulatoren verwendet.

Tabelle 4.1.: Datenfelder der Simulatorenliste

Name	**Inhalt**	**Datentyp**
id	ID	String
registries	Array der Registry-Images	Array
name	Name des Images	String
type	Szenariotyp	String
updated	Datum der letzten Aktualisierung	String
buyers	Array der Käufer-Images	Array
name	Name des Images	String
type	Szenariotyp	String
updated	Datum der letzten Aktualisierung	String
sellers	Array der Verkäufer-Images	Array
name	Name des Images	String
type	Szenariotyp	String
updated	Datum der letzten Aktualisierung	String
depenencies	Array der Dependency-Images	Array
name	Name des Images	String
type	Szenariotyp	String
updated	Datum der letzten Aktualisierung	String

In Listing 4.1 wird eine Beispielnachricht des Typs Simulatoren-Liste dargestellt. Hier gibt es Registry-Images für zwei verschiedene Szenario-Typen: „OS4ES" und „hotels". Diese Beispielszenarios werden weiterhin in den Beispielen genutzt und sollen daher kurz vorgestellt werden. das „OS4ES"-Szenario reproduziert den Handel im OS4ES. Es gibt Registry-Images, sowie Aggregatoren- und DER-Simulatoren. Das „Hotels"-Szenario stellt eine Preisvergleichsplattform dar. Es gibt Registry-Images, dort können Preise für Hotel-

zimmer von Anbieter-Simulatoren hinterlegt werden. Kunden-Images simulieren Käufer, die nach günstigen Hotelzimmern suchen.
Anhand der Szenario-Typen wird auch die Namenskonvention von Docker-Images deutlich. Ein Image-Name besteht aus dem eigentlichen Namen des Images, sowie einem so genannten Tag. Dieses Tag gibt an um welche Version des Images es sich handelt. So können mehrere Versionen des Images „DER_PV_model2“ vorhanden sein. Diese unterscheiden sich dann durch das Tag. Anzumerken ist, dass Szenarios möglich sind, die keine Abhängigkeiten benötigen. Daher kann dieses Feld auch ungenutzt bleiben.

```
1  {
2    "id": "f8944b67-12a3-69fg-c458",
3    "registries": [{
4      "name": "OS4ES_Reg:v2",
5      "type": "OS4ES",
6      "updated": "23.03.2017" },
7    {
8     "name": "Hotel_Reg:v0.1",
9     "type": "hotels",
10    "updated": "01.04.2017" }],
11   "buyers": [{
12     "name": "Aggegator_new:v3",
13     "type": "OS4ES",
14     "updated": "30.03.2017" },
15   {
16    "name": "Vergleichstest:v4",
17    "type": "hotels",
18    "updated": "30.06.2016" }],
19   "sellers": [{
20     "name": "DER_PV_model2:v1",
21     "type": "OS4ES",
22     "updated": "02.01.2016" },
23   {
24     "name": "hotelprice:v2",
25     "type": "hotels",
26     "updated": "20.03.2017" }],
27   "dependencies": [{
28     "name": "weather_service:v2.3",
29     "type": "OS4ES",
30     "updated": "08.08.2015" }],
31   }
```

```
32 }
```

Listing 4.1: Beispiel einer Simulatorenliste

Front-End-Konfigurationsnachricht

Front-End-Konfigurationsnachrichten werden vom Front-End-Modul an das Deployment-Modul übertragen wenn eine Szenario-Konfiguration abgeschlossen ist und eine Simulation gestartet werden soll. Die Struktur dieser Nachricht und die verwendeten Datenfelder werden hier beschrieben.

Anhand einer Front-End-Konfigurationsnachricht wird ein Szenario soweit beschrieben, dass daraus eine Simulation ausgeführt werden kann. Zusätzlich zu den verwendeten Simulatoren kann noch ein Simulationszeitraum angegeben werden. Diese Möglichkeit besteht, da einige mögliche Modelle historische Daten zur Simulation nutzen und daher ein bestimmter Zeitraum für alle Simulatoren festgelegt werden muss. Die in der Front-End-Konfigurationsnachricht vorhandenen Datenfelder werden in Tabelle 4.2 dargestellt.

Tabelle 4.2.: Datenfelder der Front-End-Konfigurationsnachricht

Name	Inhalt	Datentyp
id	ID	String
SimID	Identifikationsnummer der Simulation	String
timeframe	Simulationszeitraum	Array
start	Startzeitpunkt	String
end	Endzeitpunkt	String
timebase	Zeitbasis	int
simulators	Simulatoren im Szenario	Array
regtype	Registry-Typ	String
buyers	Käufersimulationen	Array
type	Typ der Simulation	String
sellers	Verkäufersimulationen	Array
type	Typ der Simulation	String
zone	zugehörige Zone	int
dependencies	Dependency-Simulationen	Array
type	Typ der Simulation	String
user	zugehörige Nutzergruppe	String

Das erste Datenfeld ist wieder die im vorigen Abschnitt eingeführte ID. Darauf folgt das Feld SimID. Dieses gibt einen Universally Unique Identifier (UUID) an, über den die auszuführen-

de Simulation, von allen Komponenten identifiziert werden kann. Da der Simulationscontroller die Ergebnisse der Simulation nach deren Ausführung in der Datenbank ablegt, kann das Deployment-Modul die zugehörigen Daten über die SimID identifizieren.
Der zu simulierende Zeitraum wird über drei Datenfelder definiert: Die Startzeit, den Endzeitpunkt und die Schrittgröße. Darauf folgen die Angaben zu den Käufersimulatoren. Diese werden in einem Array aus Strings aufgeführt und mit dem Image-Namen referenziert. Verkäufersimulatoren benötigen zusätzlich zur Angabe des Image-Namens noch die Information zur zugehörigen Zone. Zonen wurden in Abschnitt 3.1 eingeführt.
Abhängigkeiten benötigen zur eindeutigen Zuordnung zwei Felder: Den Image-Namen und die Nutzergruppe.

```
{
  "id": "12a34b67-f894-1a23-c458",
  "SimID": "49ff28a0-56b5-47fa-8bac-b0e7a69c6209",
  "timeframe": {
    "start": "1492115125",
    "end": "1492215742",
    "timebase": "10" },
  "simulators": {
  "regtype": "OS4ES_Reg:v2",
  "buyers": [{
    "type": "aggregator:latest" }],
  "sellers": [{
    "type": "DER-PV:latest",
    "zone": "1" }],
    "dependencies": [{
      "type": "weatherservice:v1",
      "user": "sellers" }]
  }
}
```

Listing 4.2: Beispiel einer Front-End-Konfigurationsnachricht

Um die Zusammenhänge deutlich zu machen stellt Listing 4.2 eine beispielhafte Front-End-Konfigurationsnachricht dar. Hier geht aus der Angabe des Zeitraumes hervor dass eine Zeitspanne von 100617 s simuliert werden soll, was 10062 Simulationsschritten bei der Schrittlänge von 10 s beträgt.
Das geplante Szenario stellt einen OS4ES-Marktplatz dar, daher wird das passende Registry-Image verwendet. Es nehmen ein Käufer- und ein Verkäufersimulator teil. Zusätzlich wird ein Wettersimulator gestartet dessen Nutzergruppe die Verkäufer sind. Diese benötigen also Zusatzinformationen vom Wetterdienst um ihre Simulationsergebnisse zu erstellen.

Im nächsten Abschnitt wird die Deployment-Modul-Konfigurationsnachricht beschrieben bei der die Konfigurationsinformationen dann in die Simulationsebene des Frameworks übertreten.

Deployment-Modul-Konfigurationsnachricht

In Kapitel 4.3.1 wurde der Zusammenhang erwähnt, in dem die Deployment-Modul-Konfigurationsnachricht zum Einsatz kommt. Diese transportiert die Szenario-Konfiguration zwischen dem Deployment-Modul in der Konfigurationsebene und dem Simulationsmanager (Simulationscontroller) in der Simulationsebene. Daher liegt das Augenmerk der Datenfelder dieser Nachricht nicht mehr bei den Typen der Beteiligten Komponenten, sondern auf den zugehörigen Netzwerkadresse, über diese die Komponenten erreichbar sind. Der Simulatiosmanager muss die Adressen aller beteiligten Simulatoren kennen, um diesen die nötigen Konfigurationsdaten zukommen zu lassen.

Tabelle 4.3.: Datenfelder der Deployment-Modul-Konfigurationsnachricht

Name	Inhalt	Datentyp
id	ID	String
SimID	Identifikationsnummer der Simulation	String
timeframe	Simulationszeitraum	Array
start	Startzeitpunkt	String
end	Endzeitpunkt	String
timebase	Zeitbasis	int
registry	Adresse der Registry	String
buyers	Array der Käufer-Container-Adressen	String-Array
sellers	Array der Verkäufer-Container	Array
address	Adresse der Verkäufer-Container	String
zone	zugehörige Zone	String
dependencies	Array der Dependencies	Array
address	Adresse der Dependency-Container	String
user	Nutzergruppe	String

Das in Kapitel 4.2.3 beschriebene Deployment-Modul startet die notwendigen Komponenten, darunter auch den Simulationsmanager. Dieser muss nun mit allen nötigen Informationen versorgt werden, um die gestarteten Komponenten erreichen und Konfigurationsinformationen zustellen zu können.
Tabelle 4.3 zeigt dazu die in der Deployment-Modul-Konfigurationsnachricht vorhanden Datenfelder. Wie bei den vorigen Nachrichtentypen gibt es auch hier als erstes die ID und

die SimID.
Daraufhin folgen die Angaben zum Simulationszeitraum. Diese werden im selben Format übergeben wie bei der Front-End-Konfigurationsnachricht.
Das nächste Datenfeld enthält die Adresse des Registry Containers. Die Adressen der Käufersimulationen werden als Array von Strings übergeben. Im nächsten Feld, der Angabe der Verkäufersimulator-Adressen, wird eine zusätzliche Angabe der Zonenzugehörigkeit benötigt. Diese wird daher ebenfalls angegeben. Zuletzt werden noch die Netzwerkadressen der Abhängigkeiten angegeben, inklusive der zugehörigen Nutzergruppe.

```
1  {
2    "id": "44d97b67-18a4-1c25-958c",
3    "SimID": "49ff28a0-56b5-47fa-8bac-b0e7a69c6209",
4    "timeframe": {
5      "start": "1492115125",
6      "end": "1492115742",
7      "timebase": "10" },
8    "registry": "OS4ES_Registry"
9    "simulators": {
10    "buyers": ["Aggregator_type1_1", "Aggregator_type1_2",
11     "Aggregator_type2_1", "Aggregator_type3_1"],
12    "sellers": [{"name": "DER_PV_1", "zone": 1},
13     {"name": "DER_PV_2", "zone": 1},
14     {"name": "DER_PV_3", "zone": 1},
15     {"name": "DER_WIND_1", "zone": 2}],
16    "dependencies": [{
17     "address": "Weather_sim_1",
18     "user": "sellers" }]
19   }
20 }
```

Listing 4.3: Beispiel einer Deployment-Modul-Konfigurationsnachricht

Eine solche Deployment-Modul-Konfigurationsnachricht wird in Listing 4.3 dargestellt. Erkennbar ist dort, dass die Containernamen über die Rancher-Plattform als Domain-Namen aufgelöst werden können und somit als Netzwerkadressen dienen. Außerdem wird für die Registry bis zu diesem Punkt nur eine Adresse angegeben, obwohl zwei Zonen im Szenario gebildet werden sollen. Dies wird aufgelöst, indem die Adresse der Registry ein Suffix bekommt das den Zonen zugeteilt wird. Diese Auflösung der Registry-Adresse in Zonen wird im nächsten Abschnitt der Beschreibung der Container-Konfigurationsnachricht erläutert und dargestellt.

Container-Konfigurationsnachricht

Nachdem in den vorigen Kapiteln die Szenario-Konfigurationsnachrichten der Management-Komponenten dargestellt wurden, soll nun die Container-Konfigurationsnachricht erläutert werden, mit deren Hilfe die Simulationscontainer ihre Konfigurationsdaten erhalten.

Container-Konfigurationsnachrichten werden vom Simulationsmanager (Simulationscontroller) an die Simulationscontainer versendet und enthalten alle nötigen Informationen, die die Simulatoren benötigen um die Simulation gemeinsam durchführen zu können. Da mithilfe dieser Nachricht jeweils nur eine Instanz konfiguriert wird, fällt der Umfang der Nachricht deutlich geringer aus, als bei den vorigen Nachrichten.

Tabelle 4.4.: Datenfelder der Container-Konfigurationsnachricht

Name	**Inhalt**	**Datentyp**
id	ID	String
SimID	Identifikationsnummer der Simulation	String
timeframe	Simulationszeitraum	Array
start	Startzeitpunkt	String
end	Endzeitpunkt	String
timebase	Zeitbasis	int
registry	Adresse der Registry	String
dependencies	Adresse der Dependency	String-Array

Wie schon in den anderen Konfigurationsnachrichten gibt es die Datenfelder ID, SimID und Simulationszeitraum (dargestellt in Tabelle 4.4). Außerdem gibt es ein Feld mit der Netzwerkadresse der Registry und eines mit der Adresse einer Abhängigkeit, etwa eines Wetterdienstes. Über die Angabe der Nutzergruppe für Abhängigkeitssimulatoren am Front-End-UI (siehe Kapitel 4.2.2) wurde zugeordnet welcher Gruppe von Simulatoren den Dienst der Abhängigkeit benötigen. Daher wird nun, falls Abhängigkeiten vorhanden sind, die Adresse dieser mitgegeben.

```
1 {
2   "id": "44d958c7-1a48-1c25-18a4",
3   "SimID": "49ff28a0-56b5-47fa-8bac-b0e7a69c6209",
4   "timeframe": {
5     "start": "1492115125",
6     "end": "1492115742",
7     "timebase": "10" },
8   "registry": "OS4ES_Registry/zone1"
```

```
9     "dependencies": "Weather_sim_1"
10  }
```

Listing 4.4: Beispiel einer Container-Konfigurationsnachricht

Eine Container-Konfigurationsnachricht könnte aussehen wie in Listing 4.4 dargestellt. Hier wird die Adresse der Registry mit „OS4ES_Registry/zone1“ angegeben. Die physikalische Adresse der Registry lautet „OS4ES_Registry“. Über die Zusatzangabe „/zone1“ wird der Simulationscontainer mit seinen Diensten der Zone 1 zugeteilt.
In diesem Falle gibt es eine Wettersimulation als Abhängigkeit. Sollte keine Abhängigkeit vorhanden sein, bleibt dieses Feld leer. So wird sichergestellt dass den Simulationscontainern als Endpunkte bekannt sind, die sie für die Durchführung der Simulation benötigen. Den Containern der Abhängigkeiten und der Registry werden dieselben Nachrichten zugestellt, diese können aber alle Informationen, abgesehen vom Simulationszeitraum, ignorieren, da diese nicht benötigt werden.

Mit diesen drei Konfigurationsnachrichtentypen kann die Szenario-Konfiguration vorgenommen werden. Nun werden noch die Nachrichtentypen beschrieben die währen der Laufzeit der Simulation auftreten: Der Step-Befehlsnachricht und der IntraSim-Nachricht.

Step-Befehlsnachricht

Diese Nachricht wird vom Zeitmodul an Simulationscontainer gesendet. Sie dient dazu den Simulationscontainern die korrekte Simulationszeit mitzuteilen.
Die Step-Befehlsnachricht fordert Simulationscontainer auf zur nächsten Barriere (dem nächsten Simulationsschritt) voranzuschreiten. Über diese wird die Synchronität der Simulationskomponenten sichergestellt und das diskrete Zeitmodell umgesetzt. Ein Simulationscontainer befindet sich so lange an einem Zeitpunkt t_n bis eine Step-Befehlsnachricht eintrifft und diesen auffordert zu t_{n+1} voranzuschreiten.

Die Datenfelder der Step-Befehlsnachricht werden in Tabelle 4.5 dargestellt. Das Feld Nachrichtentyp gibt an, um welche art Nachricht des Zeitmoduls es sich handelt. Im Prinzip lassen sich vier Nachrichtentypen aus dieser Nachricht ableiten. Zum einen der Step-Befehl (advanceTo) zum anderen die, aus Abschnitt 4.3.1 bekannte, Alive?-Nachricht (alive) und eine Stop-Nachricht (kill) sowie die Sim-Finished-Nachricht (sim-finished). Für den Step-Befehl stellt Listing 4.5 eine Beispielnachricht dar.

Tabelle 4.5.: Datenfelder der Step-Befehlsnachricht

Name	Inhalt	Datentyp
id	ID	String
type	Nachrichtentyp	String
current	momentaner Status	Array
time	aktuelle Zeit	String
timestep	aktueller Simulationsschritt	int
advance	angestrebter Status	Array
newTime	neue Zeit	String
newStep	neuer Simulationsschritt	int

```
{
  "id": "58c71c25-1a48-44d9-06c3",
  "type": advanceTo
  "current": {
    "time": "1492115125",
    "timestep": 1},
  "advance": {
    "newTime": "1492115135",
    "newStep": 2}
}
```

Listing 4.5: Beispiel einer Step-Befehlsnachricht

Neben dem Nachrichtentyp werden noch der momentane sowie der einzunehmende Simulationsschritt angegeben. Der momentane Simulationsschritt wird als Referenz angegeben, sodass die Simulatoren ihre lokale Uhr abgleichen können. Die Felder des angestrebten Status geben an welcher Zeitschritt eingenommen werden soll.

Für den Typ Alive?-Nachricht sind die Felder des aktuellen und angestrebten Simulationsschrittes von besonderer Bedeutung. Darüber kann der betroffene Simulator seinen aktuellen Status überprüfen und gegebenenfalls korrigieren. Die Stop-Nachricht hat die Aufgabe Simulatoren, die von der Simulation aufgrund von Nichteinhaltung der Zeitschritte oder Blockade der Simulation ausgeschlossen werden sollen, zu stoppen. Ein Simulator der eine Stopp-Nachricht erhält wird also von der Simulation ausgeschlossen. In einem solchen Fall wird das in Abschnitt 4.3.1 erläuterte Vorgehen angewendet.

Nachdem nun die wichtigsten Konfigurations- und Steuernachrichten des Simulationsframeworks beschrieben wurden, soll nun noch die IntraSim-Nachricht beschrieben werden, über

die die Kommunikation zwischen den Simulationscontainern, also innerhalb des Szenarios, abgewickelt wird.

IntraSim-Nachricht

Werden innerhalb des Simulationsszenarios zum Beispiel Informationen von anderen Simulatoren benötigt oder soll eine Dienstleistung bei der Registry registriert werden dann wird dafür von den Simulatoren die IntraSim-Nachricht verwendet.
Die IntraSim-Nachricht ist ein generischer Nachrichtentyp der es zulässt dass darüber Informationen übertragen werden, bei denen sich die Endpunkte über das Datenformat einigen müssen. Es ist dabei die Aufgabe der Simulationsentwickler Nachrichtentypen zu erstellen, die in der Lage sind die benötigten Informationen zu transportieren, und diese in allen relevanten Modulen zu implementieren. Dieser Ansatz wurde gewählt, da die zu übertragenen Informationen höchst heterogener Natur sind. Je nach Szenario-Typ müssen eventuell komplett andere Daten übertragen werden.

Tabelle 4.6.: Datenfelder der IntraSim-Nachricht

Name	**Inhalt**	**Datentyp**
id	ID	String
type	Nachrichtentyp	String
payload	Entität	Datenobjekt

Damit stellt die IntraSim-Nachricht prinzipiell eher einen Nachrichtenrahmen dar als eine Nachricht. Tabelle 4.6 stellt die Datenfelder der IntraSim-Nachricht dar. Dabei ist nur das Feld ID fest definiert. Dieses erfüllt dieselben Anforderungen wie bei den bereits beschriebenen Nachrichtentypen. Das Feld Nachrichtentyp erfüllt den Zweck den Kommunikationspartnern mitzuteilen welchen Datentyp sie zu erwarten haben. Zur Verdeutlichung wird in Listing 4.6 eine mögliche IntraSim-Nachricht dargestellt.

```
1 {
2   "id": "123e4567-e89b-12d3-a456",
3   "type": "weather_forecast",
4   "payload": {
5     "weather_station": "Fuhlbuettel",
6     "sunshine": "0.5",
7     "windspeed": "8.4",
8     "sun_forecast": [ "0.3", "0.3", "0.0", "0.1" ],
```

```
9        "wind_forecast": [ "8.3", "8.2", "8.0", "8.7" ]
10     }
11  }
```

Listing 4.6: Beispiel einer IntraSim-Nachricht

In diesem Beispiel wird der Nachrichtentyp als „weather_forecast“ angegeben. Damit wird innerhalb der Nachricht ein Datenobjekt übertragen bei dem der Empfänger weiß wie dieses zu lesen ist. Dieses wird von den Simulationsentwicklern festgelegt und implementiert.
Das darauffolgende Datenfeld Entität beinhaltet das zu übertragene Datenobjekt, in diesem Falle das Weather_forecast-Objekt.

Mit den hier beschriebenen Nachrichtentypen wurde nur eine Auswahl der im Simulationsframework genutzten Typen dargelegt. Weitere Nachrichtentypen können in Anhang A eingesehen werden. Im Folgenden werden nun die Schnittstellen des Frameworks beschrieben, an denen der Nutzer bzw. Simulationsentwickler ansetzen.

4.4.2. Front-End-UI

Nachdem die Programmlogik des Front-End-Moduls bereits in Abschnitt 4.2.2 beschrieben wurde, soll hier nun das Front-End-UI beschrieben werden, über das vom Nutzer Szenario-Konfigurationen und Simulationsbefehle ausgeführt werden können.
Über das Front-End-UI wird die Szenario-Konfiguration vorgenommen und der Startbefehl für die Simulation gegeben. Die Nutzerschnittstelle ist als CLI ausgeführt. Wie bereits in Kapitel 4.2.2 erwähnt, wird sobald das Front-End gestartet wird die Simulatoren-Liste vom Deployment-Modul angefragt. Daraufhin steht dem Nutzer das Front-End-UI zur Verfügung.
Nachfolgend werden die verfügbaren Befehle der Schnittstelle sowie deren Funktion dargestellt.

list

Über den List-Befehl lassen sich Elemente Anzeigen die für die Szenario-Konfiguration verfügbar sind. der List-Befehl wird immer mit einer Zusatzinformation gegeben. Die möglichen Optionen sind `all`, `registries`, `buyers`, `sellers` und `dependencies`. Die Folgende Eingabe gibt eine Liste aller Verfügbaren Simulatoren aus.

```
list all
```

show

Der Show-Befehl gibt den aktuellen Stand der Szenario-Konfiguration aus. Diese benötigt keine zusätzlichen Angaben.

config

Die Eingabe der Szenario-Konfiguration erfolgt mithilfe des Config-Befehls. Um die Befehle bei der Szenario-Konfiguration möglichst kurz zu halten, wurde dafür eine extra Eingabeebene erstellt. Über die Eingabe von `config` wird die Konfigurationsebene geöffnet. In dieser Konfigurationsebene gibt es einen speziellen Satz an Befehlen für die Szenario-Konfiguration. Die `add`, `remove` und `setTime` Befehle werden nachfolgend erläutert. Die Konfigurationsebene kann mit `end` verlassen werden.

add

Dieser Eingabebefehl dient dazu Simulatoren zur Szenario-Konfiguration hinzuzufügen. Hinter dem `add` wird der Simulator-Typ angegeben, dann die zugehörige Zone und schließlich die Anzahl der gewünschten Simulatoren dieses Typs in dieser Zone. Dabei wird die Simulatorengruppe (`buyer`, `seller`, `registry`, `dependency`) vor dem Simulator-Typen angegeben. Je nach Simulatorengruppe folgen darauf unterschiedliche Attribute. Bei einer `registry` sind keine weiteren Eingaben nötig. Eine Eingabe könnte aussehen wie folgt:

```
add registries:"Hotel_Reg:v0.1"
```

Die Simulatorengruppe `buyers` benötigt zusätzlich noch die Anzahl der gewünschten Simulatoren dieses Typs. Daher könnte eine Eingabe zum Beispiel aussehen wie folgt:

```
add buyers:"Aggegator_new:v3" quantity:"4"
```

Soll ein `sellers` Simulator hinzugefügt werden so sind zusätzlich Angaben zur zugehörigen Zone sowie zur Anzahl der Simulatoren in dieser Zone nötig, also beispielsweise:

```
add sellers:"DER_PV_model2:v1"  zone:"1" quantity
   :"10"
```

Zur Eingabe der Simulatoren in die Szenariokonfiguration muss zunächst `config` eingegeben und bestätigt werden. Damit ist die Konfigurationsebene ausgewählt. Nun können die Simulatoren hinzugefügt werden. Ein beispielhafter Ablauf wird in Listing 4.7 dargestellt.

```
config
config: add registries:"Hotel_Reg:v0.1"
config: add buyers:"Aggegator_new:v3" quantity:"4"
config: add sellers:"DER_model2:v1" zone:"1" quantity:"10"
config: end
```

Listing 4.7: Beispiel des Hinzufügens von Simulatoren

remove

Mit dem Remove-Befehl können innerhalb der `config`-Ebene Einträge aus der Szenario-Konfiguration entfernt werden. Die Eingabe erfolgt ähnlich zum `add`-Befehl. Zunächst wird `remove` angegeben gefolgt von `type` und dem Simulator-Typ. Am Ende wird noch `quantity` mit der gewünschten Anzahl angegeben. So können auch mehrere Simulatoren mit einem Befehl entfernt werden. Eine Eingabe könnte aussehen wie folgt:

```
remove type:"DER_PV_model2:v1" quantity:"2"
```

setTime

Der Befehl `setTime` dient dazu den Simulationszeitraum festzulegen. Dieser wird wie in Kapitel 4.4.1 beschrieben über drei Angaben definiert: Den Startzeitpunkt, den Endzeitpunkt sowie der Zeitbasis. Diese werden mithilfe des `setTime`-Befehls festgelegt. Nach `setTime` wird mit `start` die Startzeit angegeben, dann mit `stop` die Endzeit und mit `timebase` die Zeitbasis. Dabei wird die Zeit, wie auch in den Nachrichtentypen, in ms angegeben. So kann eine Zeiteingabe wie folgt aussehen:

```
setTime start:"1492115125" stop:"1492115742"
   timebase:"1000"
```

checkAndRun

Mit diesem Befehl wird die Simulationsausführung beauftragt. Jedoch wird die Simulation mit Eingabe des Befehls nicht sofort gestartet. Zunächst wird die Szenario-Konfiguration ausgegeben, um diese noch einmal prüfen zu können. Daraufhin kann die Simulation durch Eingabe von `run` gestartet werden.
Durch diese Bestätigung verpackt das Front-End die eingegebene Szenario-Konfiguration in einer Front-End-Konfigurationsnachricht. Durch Eingabe von `end` kann der `checkAndRun` -Modus verlassen werden.

results

Ist die Ausführung einer Simulation beendet, sendet das Deployment-Modul die Simulationsergebnisse an das Front-End. Damit sind die Ergebnisse für den Nutzer über das UI abrufbar. Über die Ergebnis-Ebene können mithilfe der `list`-, `show`-Befehle Ergebnisse einzelner Simulatoren eingesehen werden. Die dafür benötigte Syntax wird nachfolgend dargestellt.

Wie schon bei der Konfigurationsebene, lässt sich die Ergebnis-Ebene durch Eingabe des jeweiligen Befehls, in diesem Fall `results`, erreichen.

list

Der `list`-Befehl bewirkt innerhalt der Ergebnis-Ebene eine Auflistung der Simulatoren für die Ergebnisse verfügbar sind, also für die Simulatoren deren Ergebnisse vom Deployment-Modul übergeben wurden. Im Regelfall sind dies alle zuvor in der Szenario-Konfiguration definierten Teilnehmer. Vor den Einträgen der Simulatoren ist eine Nummerierung dargestellt, die eine Auswahl der anzuzeigenden Ergebnisse vereinfacht. Listing 4.8 stellt dies beispielhaft dar.

```
results
results: list
results: 1 : OS4ES_Reg:v2
results: 2 : aggregator:latest_1
results: 3 : aggregator:latest_2
results: 4 : DER-PV:latest_1
results: 5 : DER-PV:latest_2
```

```
results: 6 : DER-Wind:v1
results: 7 : weatherservice:v1
```

Listing 4.8: Beispiel des Hinzufügens von Simulatoren

show

Nachdem mit `list` die Liste der verfügbaren Ergebnisse dargestellt wird, kann mit dem `show`-Befehl einer der Einträge zum Anzeigen ausgewählt werden.
Folgende Syntax wird verwendet um die Ergebnisse eines ausgewählten Simulators anzuzeigen:

```
show:"5"
```

Bei Eingabe des obigen Befehls würden die Simulationsergebnisse des Simulators Nummer 5 auf der Liste aus Listing 4.8 ausgegeben werden.

Nachdem an dieser Stelle das Front-End-UI beschrieben wurde, soll nun die zweite Schnittstelle erläutert werden an der Entwickler ansetzen. Die Sim-API ist die Schnittstelle zwischen dem Kommunikationsrahmen und den Simulatoren, die zusammen den Simulationscontainer bilden.

4.4.3. Sim-API

Die Sim-API ist die Schnittstelle zwischen dem in Kapitel 4.3.2 beschriebenen Kommunikationsrahmen und dem Simulator. Über diese findet die sowohl Kommunikation der Simulatoren innerhalb des Simulationsframeworks statt, als auch die Kommunikation des Framework mit den Simulatoren.

Um Simulatoren in einer Vielzahl von Programmiersprachen unterstützen zu können wird eine High-Level-API verwendet, für die Implementation in verschiedenen Programmiersprachen erstellt werden können. Bei dieser High-Level-API stellt das Simulationsframework Softwarebibliotheken zur Verfügung, mit deren Hilfe Simulationsentwickler direkt in der gewohnten Programmiersprache arbeiten können. Es ist ebenfalls möglich die Nutzung dieser High-Level-API zu überspringen und direkt an der ZMQ-Übertragungsstrecke zwischen Kommunikationsrahmen und Simulator anzugreifen. In diesem Falle müssen Simulationsentwickler dann allerdings das Handling der ZMQ-Messages organisieren und alle benötigten Funktionen eigenhändig implementieren.

Der Simulator benötigt über diese Sim-API die Methoden um Nachrichten selbstständig versenden zu können und um Nachrichten zu empfangen. Dazu werden zwei verschiedene Ansätze verwendet. Zum einen werden Methoden innerhalb der Sim-API-Bibliothek implementiert die vom Simulator genutzt werden können und zum Anderen Abstrakte Methoden die vom Simulationsentwickler implementiert werden müssen. Die Auswahl des Ansatzes ist dabei abhängig vom Kommunikationsschema das verwendet wird.

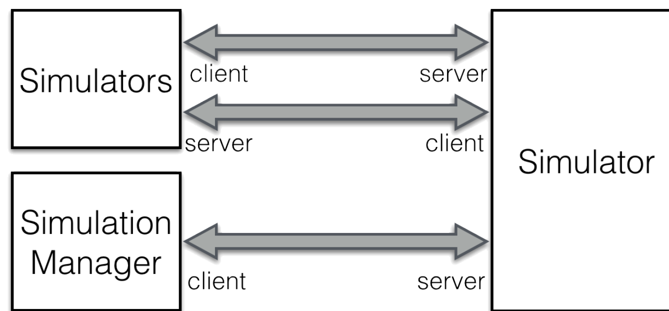

Abbildung 4.11.: Diskrete Zeitschritte der Simulation

Abbildung 4.11 zeigt die Kommunikationsmuster im Zusammenhang mit den Simulatoren. Bei jeder Datenübertragung wird ein Request-Reply-Schema verwendet. Dabei agiert die Endstelle die die Kommunikation initiiert als Client. Die auf die Nachricht reagierende Endstelle als Server. Damit sendet der Client einen Request an den Server, dieser antwortet mit der Reply. Dieses Kommunikationsschema wird verwendet, um auch bei monodirektionaler Kommunikation eine Empfangsbestätigung erhalten zu können. Der Simulationsmanager sendet als Client Daten oder Anfragen an den Simulator. Dieser antwortet mit einer Empfangsbestätigung oder den Angeforderten Daten.
Bei der Kommunikation zwischen den Simulatoren (IntraSim) ist es möglich dass beide Seiten die Kommunikation initiieren. Daher ist es notwendig, dass die High-Level-API einen Client- und einen Serverendpunkt für IntraSim-Kommunikation implementiert.

Nachdem nun die Kommunikationsstrecken erläutert wurden die von der Sim-API zur Verfügung gestellt werden, sollen nun die Methoden und ihre Funktion beschrieben werden die vom Simulationsentwickler genutzt werden können bzw. die von den Entwicklern implementiert werden müssen.
Zunächst wird die sendIntraSim-Methode beschrieben die von Entwicklern genutzt werden kann um Nachrichten zu versenden. Daraufhin werden die zu implementierenden Handler-Methoden beschrieben. Handler-Methoden sind in diesem Fall Methoden die dafür zuständig sind bei Übergabe von Nachrichten an diese eine bestimmte Aktion auszulösen und so benötigte Funktionen zu implementieren um den Anforderung den Simulationsframeworks zu entsprechen.

sendIntraSim

Die `sendIntraSim`-Methode ist vorgesehen um Nachrichten innerhalb des Simulations-Szenarios versenden zu können. Da, wie zuvor beschrieben, immer ein Request-Reply-Verfahren genutzt wird, spricht diese Methode die Client-Komponente im Kommunikationsrahmen an. Die Übergabeparameter sind dabei zum einen die Zieladresse `address` und die zu übertragenden Daten `Payload`.

```
1  method sendIntraSim(adress of recipient, payload data) {
2
3    convert payload data to JSON
4
5    send payload-JSON to recipient address
6
7    receive response-JSON from recipient
8
9    convert received JSON to data object
10
11   return received data
12 }
```

Listing 4.9: Beispiel des Hinzufügens von Simulatoren

Um die Funktionalität der Methode zu verdeutlichen, wird diese in Listing 4.9 dargestellt. Die Empfängeradresse wird als String angegeben und die zu übertragenden Daten als IntraSim-Objekt. Beim IntraSim-Objekt handelt es sich dabei um die Implementation der IntraSim-Nachricht. Der Rückgabewert dieser Funktion ist dementsprechend ebenfalls ein IntraSim-Objekt.

Die zu implementierenden Methoden werden von der Sim-API-Bibliothek als abstrakte Methoden deklariert und geben somit vor welche Methoden implementiert werden müssen und welchen Rückgabewert diese besitzen sollen. Bei diesen Methoden handelt es sich um Handler-Methoden für die verschiedenen Nachrichtentypen die den Simulator erreichen. Dies betrifft die Server-Endstellen der Kommunikationswege. Bei diesen Methoden stellt eine außenliegene Komponente einen Request und der Simulator muss als Server darauf reagieren. Daher ist es notwendig dass der Simulator Methoden bereitstellt die eintreffende Nachrichten korrekt verarbeiten und benötigte Daten, oder auch nur Empfangsbestätigungen zurücksendet. Eine Statusprüfung der Übertragung wird in jedem Falle durchgeführt, unabhängig davon ob eine Methode einen Rückgabewert besitzt oder nicht. Diese Statuskontrolle wird über den Rahmen erledigt, den das Framework um die Nachrichten legt. Bei erfolgreicher Übertragung der Konfigurationsnachricht an den Simulator erhält der Sender

der Nachricht den HTTP-Statuscode 200 als Antwort, was einem „OK“ entspricht. Bei nicht erfolgreicher Übertragung wird der Statuscode 404 (NOT FOUND) übergeben. Diese Methode ermöglicht eine zuverlässige Aussage über den Erfolg einer Datenübertragung über standardisierte Methoden.

Nachfolgend werden die von den Simulationsentwicklern zu implementierenden Methoden beschrieben.

handleConfig

Die erste Nachricht die ein Simulator vom Simulationsmanager erhält ist die Container-Konfigurationsnachricht (siehe Kapitel 4.4.1). Diese wird vom Kommunikationsrahmen entgegengenommen und über die `handleConfig`-Methode abgewickelt. Diese Methode besitzt als Rückgabewert eine Config-Response-Nachricht (siehe Anhang A.1) die einen Status-String enthält, der die Werte `failure` oder `success` annehmen kann. Im Regelfall antwortet der Simulator mit `success` wenn die Konfigurationsnachricht ausgelesen und der Simulator mit den enthaltenen Werten parametriert wurde. Sollte eine `failure` so wird im Nachrichtenfeld Fehlernachricht angegeben welcher Fehler aufgetreten ist. Dieser wird am Front-End ausgegeben um den Nutzer über den aufgetretenen Fehler zu informieren.

handleStep

Die Synchronisation der Simulatoren erledigt das Zeitmodul, wie in Kapitel 4.3.1 beschrieben, durch als Zeitschritte umgesetzte Barrieren. Die Barrieren werden mittels der Step-Befehlsnachricht gesetzt und aufgehoben. Die `handleStep`-Methode ist dafür zuständig diesen Mechanismus am Simulator umzusetzen. Mittels dieser Methode werden Step-Befehlsnachrichten übertragen, auf die der Simulator nach erfolgreicher Erledigung eines Zeitschrittes mit einer Step-Response-Nachricht (siehe Anhang A.2) antwortet. Der Simulator soll durch diese Methode Step-Befehlsnachrichten vom Typ `advanceTo` entgegennehmen und sein System entsprechend dem geforderten Zeitschritt anpassen. Erfolgt keine Antwort auf diese Nachricht wird an diese Methode eine `alive?`-Nachricht gesendet. Auch eine mögliche `kill`-Nachricht erreicht diese Handler-Methode. Sollte tatsächlich ein Fehler auftreten wird dieser vom Simulationsmanager per Fehlernachricht (siehe A.5) über das Deployment-Modul an das Front-End übergeben. Dort wird die Fehlernachricht ausgegeben. Nach Beendigung des letzten Zeitschrittes sendet der Simulationsmanager noch eine Sim-Finished-Nachricht.

```
1 method handleStep(Step-Command-message) {
2
3   handle message according to included message type
```

```
4
5    if (message type is "advanceTo") {
6
7    return Step-Response of type "step-finished"
8
9    } else if (message type is "alive?") {
10
11   return Step-Response of type "alive!"
12
13   } else if (message type is "kill") {
14
15   stop simulator
16
17   } else if (message type is "sim-finished") {
18
19   stop simulation and gather results
20
21   }
```

Listing 4.10: Beispiel des Hinzufügens von Simulatoren

Listing 4.10 stellt hier ein stark vereinfachtes Verhalten dar, welches von der `handleStep`-Methode erwartet wird. Wie hier dargestellt dient die Step-Response-Nachricht als Antwortnachricht für Step-Befehlsnachrichten für die Nachrichtentypen `advanceTo` und `alive?`.

handleIntraSim

Nachdem zuvor die Handler-Methoden für Nachrichten zwischen Framework-Komponenten und Simulatoren beschrieben wurden, soll hier nun die Methode für Nachrichten innerhalb des Simulationsszenarios dargestellt werden.
Die `handleIntraSim`-Methode stellt das Gegenstück zur `sendIntraSim`-Methode dar. Abbildung 4.12 zeigt diesen Zusammenhang. Sie ist die korrespondierende Endstelle dieser Kommunikationsstrecke und stellt damit die Serverseitige Endstelle der Request-Reply-Architektur dar. Sendet ein Simulator eine IntraSim-Nachricht über die `sendIntraSim`-Methode an einen anderen Simulator, so wird die enthaltene Nachricht an dessen `handleIntraSim`-Methode übergeben.

Die durch diese Methode ausgelöste Reaktion ist nicht fest definiert, da sie vom Inhalt der Nachricht und der Art des Simulationsszenarios abhängt. Es ist damit die Aufgabe der Simulationsentwickler den Inhalt der IntraSim-Nachricht korrekt zu verarbeiten und darauf

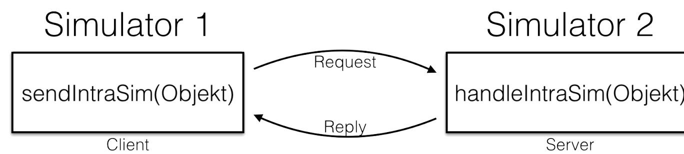

Abbildung 4.12.: Interaktion von sendIntraSim() und handleIntraSim()

zu reagieren. Der Rückgabewert dieser Funktion ist ebenfalls eine IntraSim-Nachricht. Der Inhalt der Antwortnachricht ist ebenfalls nicht definiert und es obliegt den Simulationsentwicklern diese zu standardisieren. Dieses Simulationswerkzeug stellt dabei nur die Nachrichtenart zur Verfügung mit der Kommunikation innerhalb der Simulation ermöglicht wird.

handleResult

Die vorigen Methoden werden in der Konfigurations- und Durchführungsphase der Simulation eingesetzt. Die hier erläuterte Methode wird eingesetzt nachdem eine Simulation durchgeführt wurde. Es ist notwendig die Simulationsergebnisse, die sich durch die verteilte Architektur des Simulationsframeworks nicht an einem Ort befinden, zu sammeln und gemeinsam abzuspeichern.

Die `handleResult`-Methode wird vom Simulationsmanager verwendet um die Ergebnisse von den Simulatoren einzusammeln. Dabei ist das Format der Ergebnisdaten nicht fest definiert. Es ist lediglich notwendig, dass diese auch im JSON-Format vorliegen. Der Rückgabewert der Funktion ist eine Result-Nachricht (siehe Anhang A.4), deren Payload ist ein String-Array, in welches die JSON-Daten Eintragsweise abgelegt werden. Simulationsentwickler sind verantwortlich dafür die Ergebnisse in einem String-Array bereitzustellen und bei Eintreffen einer Result-Request-Nachricht (siehe Anhang A.3) als Rückgabewert zu übergeben. Der Simulationsmanager sammelt die Daten und legt diese ebenfalls Zeilenweise in einer Tabelle ab, sodass diese vom Deployment-Modul für die Darstellung am Front-End-Modul abgerufen werden können.

Mit Abschluss der Beschreibung der Sim-API ist die Architektur des Simulationsframeworks vollständig beschrieben. Es wurde zunächst ein Überblick der Framework-Architektur in der Intrastruktur- und Framework-Ebene gegeben. Die Framework-Ebene wurde daraufhin begrifflich nach Einsatzbereich der Komponenten in zwei Ebenen, die Konfigurations- und Simulationsebene, unterteilt. Diese Komponenten wurden dann eingehend erläutert. Nach den Komponenten wurden dann die Kommunikationsschnittstellen und Datenmodelle beschrieben, die in diesem Simulationsframework Anwendung finden.
Entwickler die Simulatoren für das hier vorgestellte Framework entwickeln wollen müssen dabei beachten, dass diese die Sim-API implementieren und die beschrieben Datenmodelle

an den erläuterten Schnittstellen unterstützen müssen.

in diesem Kapitel wurde damit zunächst das Design des entworfenen Simulationsframeworks vorgestellt und weiterhin die Architektur und die Schnittstellen des Frameworks erläutert. Erst wurde die Architektur des Gesamtsystems beschrieben und dann auf die einzelnen Komponenten der Konfigurations- und Simulationsebene eingegangen. Daraufhin wurden die wichtigsten Nachrichtentypen der Kommunikationsstrecken erläutert. Außerdem wurden das Front-End-UI und die Sim-API Beschrieben.
Im nächsten Kapitel soll nun die in diesem Kapitel vorgestellte Architektur anhand der in Kapitel 3 gestellten Anforderungen validiert werden.

5. Validierung und Bewertung

In Kapitel 3 wurde das zu simulierende System analysiert und daraus Anforderung an das zu entwerfende Simulationsframework gestellt. In diesem Kapitel soll das in Kapitel 4 entwickelte Framework nun anhand der Anforderungen validiert werden. Dazu wird das Gesamtframework betrachtet und auf die Anforderungen hingehend untersucht. Zusätzlich wird das Simulationsframework gegen die in Kapitel 2.3 beschriebenen bestehenden Simulationsplattformen, über die Funktionsweise und Funktionalitäten abgegrenzt.

5.1. Anforderungen

Um das Simulationsframework zu validieren werden zunächst die in Kapitel 3 aufgestellten Anforderungen dargestellt. Es werden dann die Anforderungen an die Simulationsplattform und daraufhin die Anforderungen an die Kommunikation validiert.

5.1.1. Simulationsplattform

In diesem Abschnitt soll das entwickelte Simulationsframework gegenüber den Anforderungen an die Simulationsplattform validiert werden. Tabelle 5.1 zeigt die Anforderungen an die Simulationsplattform. Die Überprüfung der Anforderungen erfolgt hier von der höchsten Priorität (MUST) zu der niedrigsten Priorität (COULD).

Must

Zunächst wurde gefordert, dass eine Abstraktion der Software von der Hardware durchgeführt wird. Diese Anforderung wird durch den Einsatz der Docker-Umgebung erfüllt. Die Anwendungsvirtualisierung mit Containern isoliert diese von der ausführenden Hardware

Tabelle 5.1.: Anforderungen an die Simulationsplattform

Bezeichnung	Priorität
Abstraktion der Software von der Hardware	MUST
Verteilung des Simulationssystems	MUST
Unterstützung von Modellen verschiedener Programmiersprachen	SHOULD
Front-End-Komponente	COULD
Zentrale Sammlung der Simulationsergebnisse	SHOULD
Synchrone Simulation in diskretem Zeitmodell	MUST
Skalierbarkeit des Ressourcenbedarfs	SHOULD

und stellt eine virtuelle Infrastruktur bereit. Ferner wird durch die Nutzung der Rancher-Umgebung eine Cloud-Infrastruktur erzeugt, die eine Gruppe von physischen Maschinen zu einer virtuellen Infrastruktur vereint.

Des Weiteren wurde gefordert dass das Simulationssystem als verteiltes System ausgelegt wird. Da die Komponenten und Simulatoren des Frameworks alle in Containern isoliert sind und über Webservices kommunizieren besitzen diese keinen gemeinsamen Speicher und kommunizieren über Nachrichten. Zusätzlich ist für den Anwender über das Front-End nicht ersichtlich dass es sich um mehr als eine Maschine handelt, die die Simulation ausführt. Damit handelt es sich nach Tanenbaum und van Steen (2007) bei dem hier entwickelten Simulationsframework um ein verteiltes System.

Die nächste Anforderung mit höchster Priorität ist die synchrone Simulation über ein diskretes Zeitmodell. Da jede Maschine eines verteilten Systems eine eigene Systemzeit besitzt ist eine totale Synchronität nicht erreichbar, denn die Übertragungsdauer der Taktsignale benötigt Zeit. Durch die Übertragungsverzögerung können Systemuhren in einem verteilen System, dass über eine nicht deterministisch ausgelegte Infrastruktur kommuniziert, nicht in totaler Synchronität gehalten werden. Die Übertragungsdauer in Computernetzwerken kann näherungsweise Vorausgesagt werden, wenn entsprechende Messungen vorausgehen. Durch die variable Routenfindung jedoch können auch starke Abweichungen der Übertragungsdauer auftreten.
Daher wurde in diesem Simulationsframework die Barrier-Simulation nach Mellor-Crummey und Scott (1991) verwendet. Bei dieser wird jede Simulator-Instanz durch Barrieren in einem synchronen Zustand gehalten, bis die Barriere von der Kontrollinstanz (Zeitmodul) aufgehoben wird. Bis zur Aufhebung der nächsten Barriere befinden sich alle Simulatoren daraufhin synchron im nächsten diskreten Zustand. Da die Simulationsuhr pro Barriere um ein definiertes Zeitintervall voranschreitet besitzt die Simulation ein diskretes Zeitmodell mit definierter Schrittgröße, bei dem sich alle Simulatoren in einem synchronen Zustand befinden.

Damit wurde sind alle Anforderungen mit höchster Priorität (MUST) erfolgreich eingehalten worden.

Should

Der nächste Prioritätsstufe (SHOULD) wurden drei Anforderungen zugewiesen. Zum einen der Unterstützung von Modellen bzw. Simulatoren die in unterschiedlichen Programmiersprachen erstellt wurden. Zum anderen sollen die Simulationsergebnisse der verteilten Simulatoren an einer zentralen Stelle gesammelt werden. Ebenfalls soll der Ressourcenbedarf skalierbar sein.

Eine Unterstützung von Modellen die in verschiedenen Programmiersprachen geschrieben wurden wird umgesetzt, indem Simulationscontainer mit dem Kommunikationsrahmen und dem Simulator ausgestattet werden. Dazu müssen noch die nötigen Komponenten eingespielt werden, die zur Ausführung des jeweiligen Simulators benötigt werden. Ein Simulator zum Beispiel der in Python erstellt wurde benötigt noch eine Python-Laufzeitumgebung innerhalb des Containers um ausgeführt werden zu können.
Wenn Simulationsentwickler auf die Nutzung der High-Level-API verzichten kann prinzipiell jede Programmiersprache unterstützt werden für die es eine ZMQ-Bibliothek gibt. Die High-Level-API muss zunächst für die gewünschten Programmiersprachen implementiert werden um diese zu unterstützen. Damit ist eine Programmiersprachenunabhängigkeit theoretisch gegeben. Durch Implementation der High-Level-API in den Programmiersprachen können diese auch nativ unterstützt werden.

Zusätzlich wurde gefordert dass die Simulationsergebnisse an einem zentralen Punkt gesammelt werden. Dies ist notwendig, um dem Nutzer die gesammelten Ergebnisse zur Verfügung stellen zu können. Da sich die Simulatoren in diesem Framework innerhalb ihrer Simulationscontainer befinden besitzen diese keinen gemeinsamen Speicher auf dem die Simulationsergebnisse abgelegt werden können. So befinden sich die Ergebnisse auf dem Speicher des jeweiligen Simulationscontainers und müssen an einer zentralen Stelle gesammelt werden. Dies geschieht durch den Simulationsmanager der zum Ende einer Simulation die Result-Request-Nachricht an alle beteiligten Simulatoren sendet. Daraufhin legt der Simulationsmanager die Ergebnisse in der Datenbank ab. Damit wird sichergestellt, dass die Simulationsergebnisse zentral abrufbar sind und für den Nutzer zur Einsicht vorliegen.

Die letzte Anforderung der Prioritätsstufe „SHOULD“ ist die Skalierbarkeit der Ressourcenbedarfs. Die Skalierbarkeit kann, ohne bestehende Implementation des Frameworks, nur theoretisch betrachtet werden. Die Skalierbarkeit der zu simulierenden Szenarios hängt von den implementierten Modellen ab. Das Simulationsframework besitzt mit dem Simulationsmanager eine mögliche Engstelle. Da der Simulationsmanager die Simulationsuhr steuert, muss dieser in jedem Simulationsschritt Nachrichten mit jedem Simulator austauschen. Bei

einer steigenden Größe des Szenarios wird der Simulationsmanager zwangsläufig an seine Grenzen kommen. Diese Grenze wird durch die Anzahl der Nachrichten bestimmt die von der Implementation pro Sekunde verarbeiten kann. Um die Auslastungsgrenze des Simulationsmanagers zu bestimmen müsste die Implementation dieser Komponente Belastungs- bzw. Stresstests unterzogen werden.
Damit ist eine theoretische Skalierbarkeit bis zu einer Grenze gegeben, jedoch kann die Anforderung nicht generell als erfüllt gewertet werden.

Could

Die letzte vorhandene Prioritätsstufe ist „COULD“ und besitzt eine Anforderung. Es sollte eine Front-End-Komponente existieren, über die die auszuführenden Simulationen konfigurier- und steuerbar sind.
In Kapitel 4.2.2 wird eine Front-End-Komponente eingeführt über die sich die Szenarios konfigurieren lassen, Simulationen gestartet werden können und Ergebnisse angezeigt werden können. Außerdem werden Meldungen zu möglichen auftretenden Fehlern dort angezeigt.
Damit kann auch diese Anforderung als erfüllt gewertet werden.

Damit ist die Validierung der Anforderungen an die Simulationsplattform abgeschlossen. Nachfolgend sollen nun noch die Anforderungen an die Kommunikation validiert werden.

5.1.2. Kommunikation

Nachdem zuvor die Validierung des Simulationsframeworks anhand der Anforderungen an die Simulationsplattform durchgeführt wurde, soll hier die Validierung gegenüber den Anforderungen an die Kommunikation ausgeführt werden. Die an die Kommunikation gestellten Anforderungen werden in Tabelle 5.2 dargestellt. Diese werden ebenfalls von der höchsten zur niedrigsten Priorität abgearbeitet. Mit der höchsten Prioritätsstufe (MUST) sind vier Anforderungen vorhanden. Mit der nächsten Prioritätsstufe (SHOULD) ist eine Anforderung vorhanden.

Must

Zunächst wurde gefordert, dass die Kommunikationsschnittstellen über Webservices ausgeführt werden. Über Webservices wird in dem hier entworfenen Simulationsframework ei-

Tabelle 5.2.: Anforderungen an die Kommunikation

Bezeichnung	Priorität
Schnittstellen über Webservices	MUST
Transparente Kommunikation	SHOULD
Schnittstelle für Steuerbefehle	MUST
Schnittstelle für Nachrichten innerhalb des Systems	MUST
Kommunikation der benötigten Endstellen	MUST

ne Peer-to-Peer-Kommunikation aufgebaut. So kann die Einführung eines zentralen Nachrichtenservers umgangen werden. Die Verwendung von Webservices als Kommunikationsschnittstelle wurde in Kapitel 4 beschrieben. Somit sind die Kommunikationsschnittstellen der Simulatoren als Webservices ausgelegt. Zusätzlich sind die Schnittstellen zwischen den Framework-Komponenten ebenfalls als Webservices ausgelegt worden. Damit kann die Anforderung der Umsetzung der Komponentenschnittstellen als erfüllt bezeichnet werden.

Eine weitere elementare Anforderung an die Kommunikationsschnittstellen ist die Erstellung einer spezifischen Schnittstelle für Steuerbefehle. Diese Steuerbefehle werden von Framework-Komponenten an die Simulatoren gesendet. In Kapitel 4.4.3 wird die Sim-API beschrieben die definierte Schnittstellen für Steuerungsnachrichten vom Typ Container-Konfigurationsnachricht, Step-Befehlsnachricht und Result-Request-Nachricht. Damit besitzt der Kommunikationsrahmen Simulator-seitig definierte Schnittstellen für Steuerungsbefehle des Simulationsframeworks und erfüllt die Anforderung damit.

Anhand des selben Kapitels kann auch die folgende Anforderung an die Kommunikationsfunktionalität des Frameworks validiert werden. Gefordert wurde eine Schnittstelle für Nachrichten, die von Simulatoren untereinander ausgetauscht werden. Über diese Schnittstelle werden von Simulatoren zur Durchführung der Simulation benötigte Informationen ausgetauscht. Wie in Kapitel 4.4.3 beschrieben werden zwei Schnittstellen (Client- und Serverseitig) benötigt um bidirektionale Kommunikation zwischen den Simulatoren zu ermöglichen. Die IntraSim-Nachricht wird bei dieser Art von Kommunikation verwendet. Anhand der gestellten Anforderung wurde also eine Schnittstelle für Nachrichten innerhalb des Simulationssystems definiert.

Damit Simulatoren die für die Simulation benötigten Kommunikationsendstellen kennen wurde gefordert dass diese Endstellen mitgeteilt werden. Diese Anforderung wird mithilfe der, in Kapitel 4.4.1 eingeführten, Container-Konfigurationsnachricht erfüllt. Anhand der Architektur der Szenarios, die in diesem Framework ausgeführt werden können, lässt sich ermitteln welche Endpunkte den Simulatoren bekannt sein müssen um die Simulation korrekt ausführen zu können. Anhand der in der Front-End-Komponente vorgenom-

menen Szenario-Konfiguration werden die benötigten Endstellen mithilfe der Container-Konfigurationsnachricht an die Simulatoren übermittelt.

Should

Somit gelten alle hoch priorisierten Anforderungen an die Kommunikation innerhalb des Simulationsframeworks als erfüllt. Im Folgenden wird nun die Validierung der letzten Anforderung durchgeführt. Diese besitzt die Prioritätsstufe „SHOULD" und fordert eine für die Simulatoren transparente Kommunikationsinfrastruktur.
Durch die Einführung der Sim-API mit deren Mitteln die Kommunikation innerhalb des Simulationsframeworks abgewickelt wird entsteht eine Entkoppelung der Kommunikationsinfrastruktur und den Simulatoren. Simulatoren nutzen implementierte Methoden bzw. implementieren abstrakte Methoden um die Kommunikation zu verwirklichen. Über den Kommunikationsrahmen wird die Umsetzung der Methodenaufrufe auf die Kommunikation über Webservices vorgenommen (siehe Kapitel 4.3.2 und 4.4.3). Die Kommunikation stellt sich daher für die Simulatoren als transparent dar und erfüllt somit die gestellte Anforderung.

Tabelle 5.3.: Validierung der Frameworkarchitektur

Anforderung	**Einhaltung**
Abstraktion der Software von der Hardware	Ja
Verteilung des Simulationssystems	Ja
Modelle verschiedener Programmiersprachen	Ja
Front-End-Komponente	Ja
Zentrale Sammlung der Simulationsergebnisse	Ja
Synchrone Simulation in diskretem Zeitmodell	Ja
Skalierbarkeit des Ressourcenbedarfs	Bed. ja
Schnittstellen über Webservices	Ja
Transparente Kommunikation	Ja
Schnittstelle für Steuerbefehle	Ja
Schnittstelle für Nachrichten innerhalb des Systems	Ja
Kommunikation der benötigten Endstellen	Ja

Damit erfüllt das in Kapitel 4 entworfene und spezifizierte Simulationsframework alle in Kapitel 3 gestellten Anforderungen. Diese werden zur Übersicht mit den Ergebnissen der Validierung dieses Abschnittes in Tabelle 5.3 dargestellt. Es bleibt jedoch durch Tests zu überprüfen in wie fern die Skalierbarkeit des Ressourcenbedarfs eingeschränkt wird. Ebenso ist die Unterstützung von Modellen in verschiedenen Programmiersprachen abhängig von der

Implementation dieses Frameworks. Theoretisch stellt lediglich die Verfügbarkeit von ZMQ-Bindings für spezifische Programmiersprachen eine Grenze dar.

5.2. Abgrenzung

Nachdem zuvor das hier entworfene Simulationsframework und die gestellten Anforderungen an dieses gegenübergestellt wurden, soll nachfolgend die Abgrenzung der verwandten Produkte anhand der identifizierten Anforderungen vorgenommen werden. Es wird geprüft, inwiefern die bestehenden Simulationsplattformen aus Kapitel 2.3 den Anforderungen entsprechen.

Tabelle 5.4.: Gegenüberstellung Anforderungen/verwandte Produkte

Anforderung	**Mosaik**	**Jadex**	**GridSpice**
Abstraktion der SW von der HW	Nein	Teilw.	Ja
Verteilung des Simulationssystems	Ja	Ja	Ja
Modelle verschiedener Programmiersprachen	Ja	Nein	Ja
Front-End-Komponente	Ja	Ja	Ja
Zentrale Sammlung der Simulationsergebnisse	Ja	Ja	Ja
Synchrone Simulation in diskretem Zeitmodell	Ja	Ja	Ja
Skalierbarkeit des Ressourcenbedarfs	Bed. ja	Ja	Ja
Schnittstellen über Webservices	Nein	Teilw.	k.A.
Transparente Kommunikation	Teilw.	Ja	k.A.
Schnittstelle für Steuerbefehle	Ja	Bed. ja	Ja
Schnittstelle für Nachrichten innerhalb des Systems	Nein	Bed. ja	Ja
Kommunikation der benötigten Endstellen	Nein	Bed. ja	k.A.

Dazu listet Tabelle 5.4 die Anforderungen an das Simulationsframework auf und stellt dabei gegenüber ob die Simulationsplattformen die Anforderungen erfüllen. Es ist hier erkennbar, dass keine der Simulationsplattformen die Anforderungen uneingeschränkt erfüllt. Mosaik bietet keine Abstraktion der Simulationsumgebung von der Hardware-Infrastruktur, nutzt keine Webservices für die Kommunikation, keine Schnittstelle für Nachrichten innerhalb des Systems und keine Kommunikation der benötigten Endstellen an die Simulatoren. Nachrichten zwischen den Modellen werden dort immer über die Mosaik-Applikation vermittelt, daher findet keine Peer-to-Peer-Kommunikation statt. Die Anforderung an die Skalierbarkeit wird wie in dem hier entwickelten Framework auch nur bedingt erfüllt, da ein Engpass durch die Nachrichtenvermittlung entsteht. Die Skalierbarkeit ist also bedingt durch die Durchsatzgrenze der Vermittlungsstelle.

Eine Transparente Kommunikation ist bei Mosaik nur gegeben, wenn ein Simulator in Python implementiert wurde. Für Python wird eine High-Level-API-Bibliothek angeboten. Andere Programmiersprachen müssen die Kommunikationsendstellen erst implementieren. Die Anforderung an eine Transparente Kommunikation ist also nur teilweise erfüllt.
Daher bietet die Mosaik-Plattform keine Option für die Nutzung in dem in Kapitel 3.1 beschriebenen Anwendungsfall.

Jadex bietet ebenfalls keine Abstraktion von der Hardware-Infrastruktur im Sinne einer Virtualisierung an. Es ist allerdings möglich Verteilungstranzparenz herzustellen. Außerdem können nur in Java implementierte Agenten verwendet werden und es werden keine Schnittstellen über Webservices ausgeführt. Es ist prinzipiell möglich Webservices zur Kommunikation zwischen Simulatoren einzurichten. Dabei wird aber nicht die Framework-interne Kommunikation über Webservices abgewickelt. Daher wird die in Kapitel 3 gestellte Anforderung nur teilweise erfüllt. Einige Schnittstellen für Steuerbefehle können durch Erweiterungs-Frameworks bereitgestellt werden. Ebenso die Schnittstellen für Nachrichten innerhalb des Systems und die Kommunikation der benötigten Endstellen. Dies führt zu einer bedingten Erfüllung dieser Anforderungen.
Durch die hier beschriebenen Einschränkungen ist Jadex ebenfalls keine passende Lösung für den Anwendungsfall.

GridSpice erfüllt prinzipiell alle Anforderungen zu denen Informationen verfügbar waren. Zu der Kommunikation zwischen den Simulatoren, der Transparenz der Kommunikation sowie der Kommunikation der benötigten Endstellen konnten keine spezifischen Informationen gefunden werden.
Jedoch gibt es bei GridSpice weitere Einschränkungen. Das GridSpice-Framework ist auf die Nutzung mit AWS ausgelegt. Das bedeutet einen dauerhaften Kostenfaktor für die Zeit der Nutzung des Simulationsframeworks. Außerdem basieren die Simulationen bei GridSpice immer auf Simulationen des Stromnetzes. Auf diesem lassen sich dann weitere Elemente aufsetzen. Daher wird eine Simulation des Anwendungsfalls aus Kapitel 3.1 aufwändiger und Ressourcenintensiver als notwendig, da die Simulation des Stromnetzes zur Generierung der benötigten Simulationsergebnisse nicht notwendig ist.
Daher ist auch diese Simulationslösung kein passendes Produkt für diese Simulation.

Zusammenfassend ist also keines der vorgestellten Produkte in allen Punkten konform zu den in Kapitel 3 gestellten Anforderungen.

5.3. Fazit

Damit wird deutlich dass das hier entworfene Simulationsframework die gestellten Anforderungen erfüllt und zusätzlich eine Ausrichtung auf eine Cloud-Infrastruktur besitzt. Durch diese Ausrichtung werden aktuelle Trends ausgenutzt und die stetig wachsende Anzahl an Cloud-Service-Anbietern bietet Optionen zur Nutzung mit diesem Framework. Lediglich die Anforderung der Skalierbarkeit der Ressourcen ist mit Einschränkungen erfüllt worden. Durch eine technische Obergrenze der Datenverarbeitung in der ØMQ-Komponente im Zeitmodul, besitzt auch die Skalierbarkeit des Simulationsframeworks ein Limit. Anhand der im vorigen Abschnitt dargestellten Abgrenzung gegenüber verwandten Produkten wird deutlich dass die drei vorgestellten Simulationslösungen keine passende Option für den Anwendungsfall dieser Arbeit darstellen, was die Notwendigkeit dieses Simulationsframeworks weiterhin stützt.

In diesem Kapitel wurde die in Kapitel 4 vorgestellte Architektur und Funktionalität des Simulationsframeworks den, in Kapitel 3 aufgestellten, Anforderungen gegenübergestellt und validiert. Außerdem wurde das Framework gegenüber verwandten Lösungen abgegrenzt, indem diese mit den Anforderungen dieser Arbeit gegenübergestellt wurden. Abschließend wurde ein Fazit aus den gewonnenen Ergebnissen gezogen.

6. Prototypische Implementierung des Kommunikationsrahmens

Im vorangegangenen Kapitel 4 wurde die Architektur des Simulationsframeworks und die Kommunikationsschnittstellen beschrieben. In diesem Kapitel soll nun die prototypische Implementierung des Kommunikationsrahmens beschrieben werden. Dazu werden zunächst die eingesetzten Produkte in Abschnitt 6.1 beschrieben sowie die Gründe für deren Verwendung. Anschließend wird in Abschnitt 6.2 die im Prototypen umgesetzte Architektur spezifiziert und beschrieben.

6.1. Eingesetzte Werkzeuge

In diesem Abschnitt werden die zur Implementierung des Kommunikationsrahmens verwendeten Werkzeuge beschrieben und begründet warum diese eingesetzt wurden. Die beschriebenen Werkzeuge sind das Spring Framework und ZeroMQ.

Die Implementierung des Prototyps ist in Java umgesetzt. Dementsprechend sind die verwendeten Produkte für Java ausgelegte Lösungen oder unterstützen eine Vielzahl von Programmiersprachen.
Das Spring Framework ist ein open source Framework für die Java Plattform. (Pivotal Software (2017b)) Mit der Spring Boot Erweiterung stellt Spring Programmtemplates bereit, die zum Beispiel ein einfaches Aufsetzen von Web-Services ermöglichen. Dabei sind die Programme bereits vorkonfiguriert, beinhalten alle nötigen Bibliotheken und können leicht an vorliegende Bedürfnisse adaptiert werden (Pivotal Software (2017a)).

Da Spring Boot mehrere Funktionen erfüllt, die bei der Implementierung benötigt werden wird es dabei eingesetzt. Über das Spring Framework wird das Einbinden von Abhängigkeiten, also zum Ausführen des Programmes benötigten Bibliotheken, durchgeführt. Außerdem bietet Spring Boot eine sehr komfortable Lösung zur Erstellung von Web Applikationen an. Durch die Verwendung von Annotiations in Spring Boot lassen sich Funktionen von Klassen deklarieren. Eingehende Anfragen bei einem Web-Service werden beispielsweise durch die

„@RestController“-Annotation direkt an die annotierte Klasse übergeben, in der dann die verschiedenen Anfragen den passenden Methoden zugeordnet werden.

Zusätzlich wird die ZeroMQ eingesetzt. ZeroMQ, auch ZMQ oder ØMQ, ist eine Bibliothek für verteilte Nachrichtenübertragung über Sockets. ZeroMQ stellt asynchrone Nachrichtenübertragung zur Verfügung bei der jedoch, nicht wie bei anderen Middleware-Lösungen, kein Broker benötigt wird. Bei der Verwendung von ØMQ ist es notwendig eines der zur Verfügung gestellten Kommunikationsmuster zu verwenden. Dabei werden zahlreiche Kommunikationsmuster, unter Anderem Request-Reply, Push-Pull, unterstützt. (ØMQ (2014a)) ZeroMQ ist in einer vielzahl von Sprachbindungen verfügbar (siehe ØMQ (2014b)), darunter auch Python, C und Java.
ZeroMQ bietet eine leichtgewichtige Implementation von Kommunikationsstrecken an und bietet komfortable Funktionen wie Multicast-Nachrichten, kombinierte Kommunikationsmuster und Datenverschlüsselung auf dem Übertragungsweg (ØMQ (2014a)).

Es wird in dem hier entwickelten Prototypen eingesetzt, da eine lose Kopplung zwischen dem Kommunikationsrahmen und der High-Level-API erreicht werden soll. So ist eine Adaption verschiedener Sprachbindungen der High-Level-API über standardisierte Schnittstellen problemlos möglich. Da sich die Kommunikationsstrecke innerhalb eines Docker-Containers befindet ist eine Implementation dieser mithilfe der ØMQ eine leichtgewichtige Lösung mit hoher Zuverlässigkeit.

Nachdem nun die eingesetzten Werkzeuge vorgestellt wurden und ihr Einsatz kurz begründet wurde, soll nun im nächsten Abschnitt die Architektur der Implementation des Kommunikationsrahmen dargestellt werden.

6.2. Architektur des Prototypen

In diesem Kapitel wird die Software-Architektur des implementierten Prototypen des Kommunikationsrahmen beschrieben. Dazu wird zunächst ein Überblick über die zu implementierende Struktur und die beteiligten Module gegeben. Darauf wird dann die Umsetzung beschrieben und mithilfe geeigneten Abbildungen dargestellt.

Zur Erstellung einer prototypischen Implementation des Kommunikationsrahmens müssen drei Komponenten innerhalb des Rahmens erstellt werden. Der Webservice, ein Handler für die eintreffenden Nachrichten und eine ZeroMQ-Komponente, die die Schnittstelle zum Simulator bereitstellt. Die ZeroMQ-Komponente ist dabei geteilt in eine Server- und eine Client-Komponente um eine bilateral initiierte Kommunikation zu ermöglichen. Wie bereits

erwähnt stellt Spring Boot eine Lösung für Web-Applikationen bereit, die in diesem Prototypen auch verwendet wird. Die genaue Verwendung der Komponenten wird noch weiter erläutert.

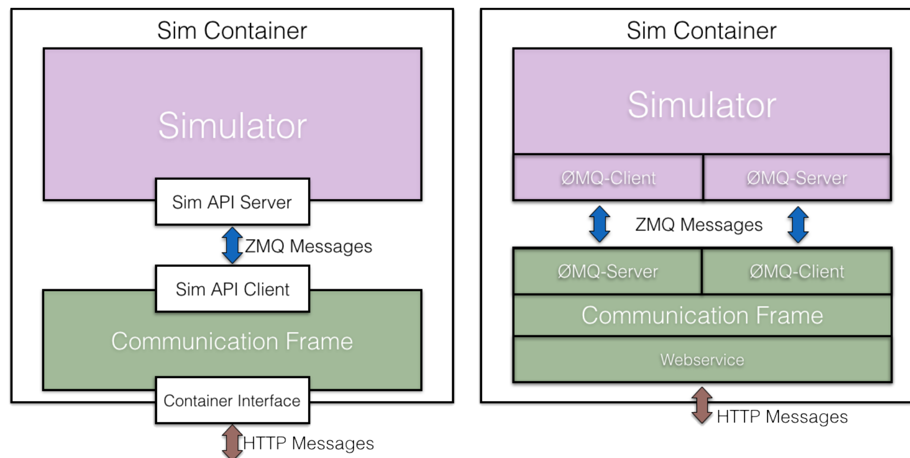

Abbildung 6.1.: Spezifikation der Protoyp-Architektur

Abbildung 6.1 stellt die Darstellung des Simulationscontainers aus Kapitel 4.3.2 gegenüber der drei Komponenten-Architektur die hier beschrieben wird. Dabei wird dargestellt dass das Container Interface über den Webservice realisiert wird und die Sim-API durch ZMQ-Services.
Von außerhalb eintreffende Anfragen werden vom Web-Service entgegengenommen und anhand von URL-Mappings zugeordnet. Anhand der Spezifikation aus Kapitel 4.4.3 sind vier URL-Pfade notwendig um die Anfragen korrekt einzuordnen. Die verwendeten Pfade werden in Tabelle 6.1 aufgelistet.

Tabelle 6.1.: Liste der umgesetzten URL-Pfade

Pfad	**zugehöriger Nachrichtentyp**
/config	Container-Knfigurationsnachricht
/sync	Step-Befehlsnachricht
/intraSim	IntraSim-Nachrichten
/result	Result-Request-Nachrichten

Diese Pfade werden im Webservice des Kommunikationsrahmens umgesetzt. Auf der Simulator-seitigen Schnittstelle werden diese Pfade über verschiedene Kommunikationsstrecken umgesetzt. Jeder Nachrichtentyp erhält dabei eine eigene Portnummer. Die Portnummer wird dann am Kommunikationsrahmen und an der High-Level-API festgelegt. Damit wird sichergestellt, dass alle Komponenten austauschbar bzw. erweiterbar sind und

trotzdem alle Schnittstellen kennen. Die festgelegten Portnummern werden in Tabelle 6.2 dargestellt.

Tabelle 6.2.: Liste der umgesetzten Port-Mappings

Portnummer	zugehöriger Nachrichtentyp
5500	Container-Knfigurationsnachricht
5510	Step-Befehlsnachricht
5520	IntraSim-Nachrichten (eingehend)
5530	Result-Request-Nachrichten
5540	IntraSim-Nachrichten (ausgehend)

Mithilfe des Spring-Boot-starter-web Pakets werden alle nötigen Komponenten für den Web-Service bereitgestellt. Dieses beinhaltet unter Anderem das Apache Tomcat Paket, welches die Webservices in Java ermöglicht. Für den Transport der Nachrichten zur High-Level-API werden die Rahmen-Datenobjekte aus dem OS4ES-Projekt verwendet. Diese sind generische Datentypen, die neben den übertragenen Daten, noch den Namen des Datentypen übertragen, sodass diese ohne zusätzliche Informationen in die Korrekten Datenobjekte gewandelt werden können. Außerdem wird die open source Applikation ByteArray verwendet. Mit dieser können Java Objekte in Byte-Arrays überführt werden und zurück. Dies ist notwendig, da Nachrichten, die über ZeroMQ übertragen werden immer als Byte-Array vorliegen müssen.
Nachdem hier eine kurze Übersicht über die Architektur gegeben wurde und erste Spezifikation festgelegt wurden, sollen in den nächsten Abschnitten die implementierten Klassen beschrieben werden. Die beschriebenen Klassen sind der RestController, der Handler, das Kommunikationsmodul der High-Level-API und der ApiHandler.

6.2.1. RestController

In diesem Abschnitt soll die RestController-Klasse des Prototyps beschrieben werden. Diese ist dafür verantwortlich Anfragen die am Webservice eintreffen entgegenzunehmen und entsprechend der Programmlogik weiterzuverarbeiten.
Der Name RestController (Representational State Transfer (REST)) ist etwas irreführend, da das implementierte Interface genaugenommen nicht RESTful ist. Um ein RESTful Interface anzubieten muss dieses nach Balasubramanian u. a. (2013) die HTTP-Methoden (GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, CONNECT, TRACE) nach ihrer Definition nutzen um Ressourcen abzufragen bzw. zu modifizieren. Diese Funktionalität wird hier jedoch nicht genutzt.

Dieser Name wurde allerdings gewählt, da mit der Spring Annotation „@RestController“ die Klasse festgelegt wird die Anfragen am Webservice entgegen nimmt. Die HTTP-Methoden werden dort ebenfalls mitgegeben, sie werden in diesem Falle jedoch nicht konsistent verwendet.

```
@RequestMapping(value = "/IntraSim", method = RequestMethod.POST)
public ResponseEntity<IntraSimObject> intraSim(@RequestBody IntraSimObject content) {
    final ReqRep.Request<IntraSimObject> request = DerRequest.reqrep(IntraSimObject.class).entity(content).build();
    final ReqRep.Response<IntraSimObject> response = Handler.handleIS(request);

    if (response.success()) {
        LOG.info("/IntraSim - message processed - id: " + content.getId());
        return new ResponseEntity<IntraSimObject>(response.getEntity(), HttpStatus.OK);
    }
    else {
        LOG.info("/IntraSim - message could not be processed - id: "+content.getId()+" error: "+response.getMessage());
        return new ResponseEntity<IntraSimObject>(response.getEntity(), HttpStatus.NOT_FOUND);
    }
}
```

Abbildung 6.2.: IntraSim-Methode des DerControllers

Der RestController besitzt die „@RestController“-Annotation um als REST-Controller zu fungieren. Diese Klasse besitzt fünf Methoden. Eine für jeden Nachrichtentyp bzw. Pfad (siehe Tabelle 6.1) plus eine für den URL-Pfad „Info“. Über diesen wird ein String mit Informationen über den Kommunikationsrahmen zurückgegeben. Diese Informationen sind etwa Versionsnummer der Applikation und Applikationstyp (hier etwa: „Communication frame of a simulation container, version: 1.0“).

In Abbildung 6.2 wird der Quellcode der IntraSim-Methode exemplarisch für die Controller-Methoden dargestellt. Über die „@RequestMapping“-Annotation wird zugewiesen bei welchem Anfragepfad die Anfragen an den Webservice an diese Methode geleitet werden. „@RequestBody“ weißt den Nachrichteninhalt der Anfrage als Parameter der Methode zu, in diesem Fall ein IntraSimObject, zu. Im nächsten Schritt wird das Objekt in ein DerRequest-Objekt verpackt. Dieses Objekt ist das im vorigen Abschnitt beschriebene Rahmen-Datenobjekt aus dem OS4ES-Projekt. Über die build-Methode der DerRequest-Klasse wird ein eine Request-Instanz erzeugt, deren Felder Class und entity mit den Informationen aus der Anfrage (Typ ist IntraSim.class und entity ist der Nachrichteninhalt) gefüllt werden.
Daraufhin wird die handleIS-Methode der Handler-Klasse aufgerufen. Diese gibt die Antwort auf die eingetroffene Nachricht zurück. Über die das ENUM ResponseType der Der-Response (das korrespondierende OS4ES-Rahmenobjekt für Antworten) wird überprüft ob die Übertragung zum Simulator erfolgreich war. Je nach ResponseType wird daraufhin der Inhalt der Antwortnachricht, entweder mit dem HttpStatus OK oder NOT_FOUND, an den Anfragesteller übergeben.

6.2.2. Handler

Nachdem zuvor der RestController erläutert wurde, soll hier nun der Handler beschrieben werden, dessen Handler-Methoden vom RestController aufgerufen werden um die eintreffenden Anfragen zu verarbeiten. Entsprechend der Methodenaufrufe des RestControllers besitzt der Handler vier Methoden. Dabei wird je eine Methode für jeden Nachrichtentyp verwendet. Die Handler-Methoden nehmen die Anfragen in Form eines DerRequest-Request-Objekts entgegen und übertragen diese Zum Simulator über die High-Level-API. Die Antwort wird als Rückgabewert der Handler-Methoden als DerRequest-Response zurückgegeben.

```
public static ReqRep.Response<IntraSimObject> handleIS(ReqRep.Request<IntraSimObject> request) {
    LOG.info("Sending IntraSim request");
    byte[] byteRequest = null;
    ReqRep.Response<IntraSimObject> response = null;

    try {
        byteRequest = ByteArray.toByteArray(request);
    } catch (IOException e) {
        e.printStackTrace();
        LOG.error("handleIS - ByteArray.toByteArray exception: " + e);
        return new ReqRep.Response<IntraSimObject>(ResponseType.FAILURE, e.toString());
    }

    ClientZMQ clientCMD = ClientZMQDeployer.getClient(IntraSim_PORT);
    byte[] byteResponse = clientCMD.sendMsg(byteRequest);

    if (byteResponse == null) {
        return new ReqRep.Response<IntraSimObject>(ResponseType.FAILURE, "currently not available");
    }

    try {
        response = ByteArray.toSpecObj(byteResponse, ReqRep.Response.class);
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
        LOG.error("handleIS - ByteArray.toSpecObj exception: " + e);
        return new ReqRep.Response<IntraSimObject>(ResponseType.FAILURE, e.toString());
    }
    return response;
}
```

Abbildung 6.3.: handleIS-Methode des Handlers

Die bereits im vorigen Abschnitt erwähnte handleIS-Methode wird in Abbildung 6.3 exemplarisch für die Handler-Methoden abgebildet. Der Handler hat nun die Aufgabe die Anfrage in ein Byte-Array zu wandeln, dieses per ØMQ an die High-Level-API im Simulator zu übertragen, die Antwort entgegenzunehmen, diese wieder in ein Java-Objekt zu wandeln und dann an den RestController zu übergeben. Dabei wird immer ein Error-Handling, also eine Fehlerüberwachung durchgeführt.

Die Konvertierung vom Java-Objekt zum Byte-Array wird über die toByteArray-Methode des ByteArray ausgeführt. Daraufhin wird eine Clienten-Instanz der ØMQ erzeugt über den

ClientZMQDeplolyer. Der ClientZMQDeployer wurde hier eingeführt um immer die korrekte Instanz des ØMQ-Klienten zu erhalten. Diese binden sich fest an einen Port, sodass dieser nicht mehr von weiteren Instanzen genutzt werden kann. Über den ØMQ-Klienten wird dann die Anfrage an den Simulator übertragen. Der Rückgabewert der sendMsg-Methode enthält dann die Antwort auf die gestellte Anfrage.
Die Antwort wird dann von einem Byte-Array zum entsprechenden Java-Objekt konvertiert und an den RestController übergeben. Sollte noch im Kommunikationsrahmen ein Fehler bei der Verarbeitung auftreten so gibt der Handler die Antwort mit dem ResponseType „Failure“ zurück.

6.2.3. Kommunikationsmodul (High-Level-API)

Zwischen dem in Kapitel 6.2.2 beschriebenen Handler und dem Kommunikationsmodul liegt die erwähnte ØMQ-Übertragungsstrecke zur Entkopplung der Komponenten. Das Kommunikationsmodul ist eine Klasse innerhalb der High-Level-API-Bibliothek die den Simulationsentwicklern zur Verfügung gestellt wird um die Webservice-Schnittstellen des Simulationsframeworks nutzen zu können. In dieser werden die benötigen ØMQ-Klienten- und Serverinstanzen erzeugt und eintreffende Anfragen an den im folgenden Abschnitt 6.2.4 beschriebenen ApiHandler weitergeleitet.

Um die High-Level-API nutzen zu können ist es notwendig dass Simulationsentwickler eine Instanz des Kommunikationsmoduls erzeugen. der Konstruktor sieht dabei vor das eine Instanz des ApiHandlers mitgegeben wird. Bei Instanziierung dieser Klasse werden die ØMQ-Klienten und -Server über den ClientZMQDeployer und ServerZMQDeployer gestartet und auf die benötigten Ports gebunden.

```
while (!Thread.currentThread().isInterrupted()) {
    byte[] request = Socket.recv(0);
    LOG.info("message received");

    /**
     * Passes the arriving request to the handler implemented by the simulation developer
     */

    if (zmqport.equals("5500")) {
        ReqRep.Request<IntraSimObject> rrreq = converter.convertToReqIS(request);
        IntraSimResponse = new ReqRep.Response<IntraSimObject>(ResponseType.SUCCESS, handler.handleIS(rrreq.getEntity()));

        try {
                response = ByteArray.toByteArray(cmdResponse);
            } catch (IOException e) {
                e.printStackTrace();
                LOG.error("An error occured while using ByteArray.toByteArray(): " + e);
            }
        LOG.info("Sending IntraSim response");
    }
}
```

Abbildung 6.4.: ZMQ-Controller der High-Level-API

Anfragen treffen innerhalb der Server-Instanzen der ZMQ ein. Diese Routine wird in Abbildung 6.4 dargestellt. Hier wird eine eintreffende Anfrage zunächst von einem Byte-Array in ein DerRequest-Request-Objekt mit IntraSimObject als Payload-Objekt gewandelt über die convertToReqIS-Methode des ObjectConverters. Daraufhin wird der Payload (das IntraSimObject) dieser Anfrage an die handleIS-Methode des ApiHandlers übergeben. Der Rückgabewert wird in eine DerRequest-Response verpackt und an den ZMQ-Server zurückgegeben.

6.2.4. ApiHandler (High-Level-API)

Bis zu diesem Punkt wurden implementierte Klassen beschrieben, die zum Handling der eintreffenden Nachrichten notwendig sind. Der ApiHandler allerdings ist eine abstrakte Java Klasse, in der die benötigten Handler-Methoden deklariert werden. Für die Implementation dieser Handler-Methoden sind die Simulationsentwickler verantwortlich. Über die Implementation wird ermöglicht, dass andere Komponenten „Remote Procedure Calls“, also die Fernausführung von Methoden, ausführen können.

Die zu Implementierenden Methoden des ApiHandlers sind handleConfig, handleStep, handleIntraSim und handleResult. Diese Methoden wurden in Kapitel 4.4.3 beschrieben.

Nachdem nun die Implementierten Klassen und deren Methoden erläutert wurde wird in Abbildung 6.5 ein Sequenzdiagramm dargestellt dass den Aufruf nach dem Eintreffen einer Anfrage am Webservice veranschaulicht. Es werden alle aufgerufenen Methoden und die zugehörigen Klassen dargestellt. Die Trennung des Kommunikationsrahmens und des Simulators ist mit einer gestrichelten Linie gekennzeichnet.

Damit wurde in diesem Kapitel eine prototypische Implementation des in Kapitel 4.3.2 beschriebenen Kommunikationsrahmens angefertigt. Die Umsetzung in Java wurde mithilfe des Spring Frameworks und der ØMQ-Bibliothek implementiert. Außerdem wurde die Applikation mit ihren Klassen und deren Methoden beschrieben und mit den in Kapitel 4.3.2 beschriebenen Funktionen in Verbindung gebracht. Anschließend wurde das Funktionsprinzip anhand eines Sequenzdiagramms dargestellt. Nachfolgend werden nun die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick gegeben.

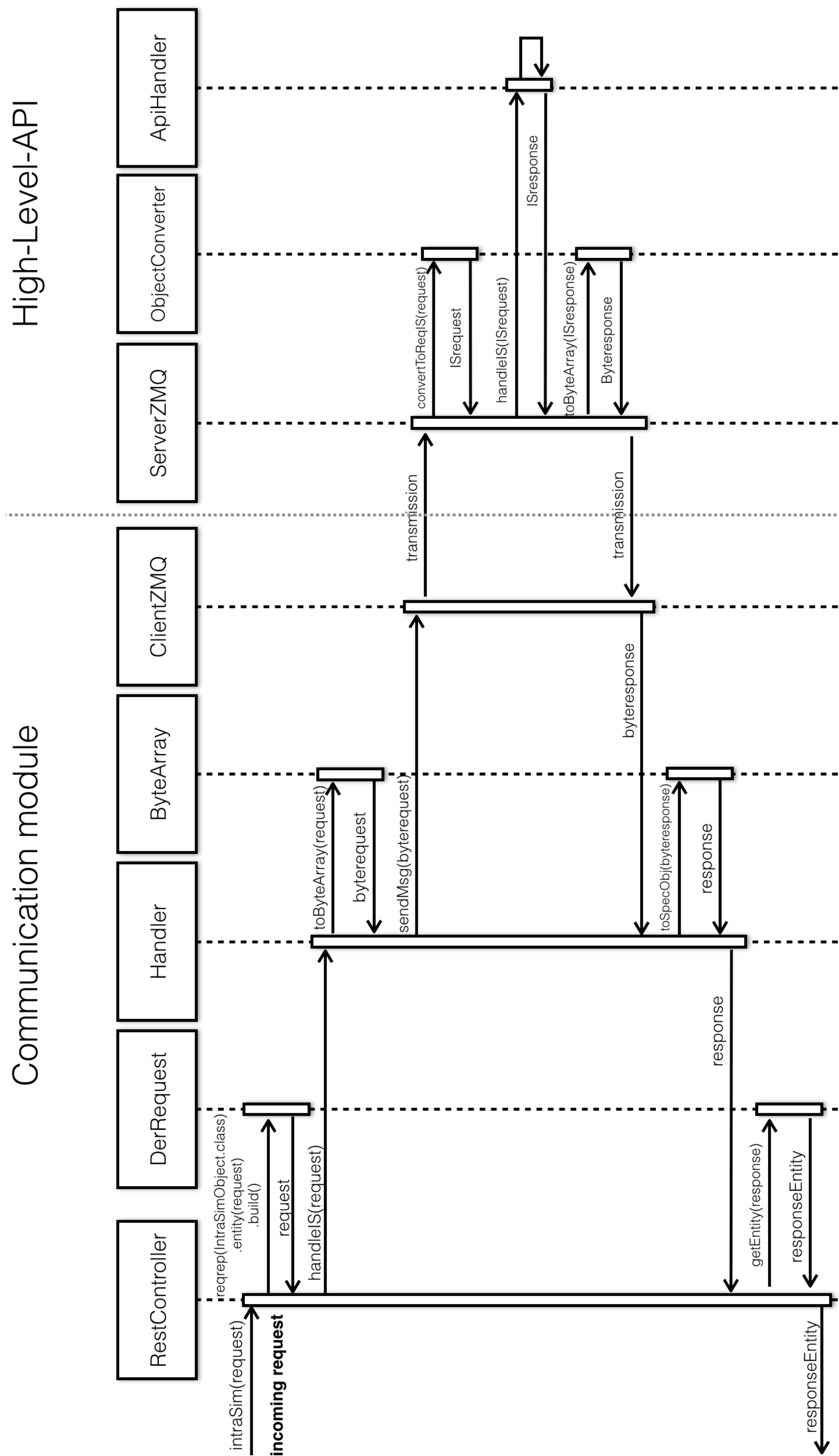

Abbildung 6.5.: Sequenzdiagramm einer Anfrage am Webservice

7. Zusammenfassung und Ausblick

Nachdem in den letzten Kapiteln die Entwicklung und der Entwurf eines Simulationsframeworks sowie die Implementierung eines Prototypen beschrieben wurde, sollen in diesem Kapitel die gewonnenen Ergebnisse zusammengefast werden. Diese werden außerdem mit den in Kapitel 1.2 formulierten Zielen in Bezug gebracht. Außerdem wird ein Ausblick über zukünftige Tätigkeiten und Forschungsthemen gegeben, mit denen das Produkt dieser Arbeit weiterentwickelt werden könnte.

7.1. Zusammenfassung

Die Simulation ist ein wichtiges Werkzeug in der Entwicklung neuer Technologien und Produkte. Durch ihren Einsatz können die Ansätze validiert und getestet werden. So kann die Funktionalität und Einsatzfähigkeit von Produkten überprüft werden, bevor diese einem Labortest unterzogen und dann in Feldtests eingesetzt werden.
Ein solches Szenario stellt sich auch für den Energy-Service-Marktplatz dar, der im OS4ES-Projekt entwickelt wurde (siehe Kapitel 2.1.4). Die dort entwickelten Komponenten sollen durch Simulation ihre Robustheit und Einsatzfähigkeit beweisen, bevor physische Systeme zum Test verwendet werden.

In dieser Arbeit wurde ein Cloud-basiertes Simulationsframework entwickelt, das auf heterogenen Infrastrukturen verwendet werden kann, um Szenarios zu simulieren. Das Framework eignet sich für Szenarios, die einen Verzeichnisdienst (Directory Service) der verfügbaren Endstellen implementieren. Im OS4ES-Projekt wird dieser Verzeichnisdienst von der Registry (siehe Kapitel 3.1) verkörpert. Daher lassen sich mit diesem Framework Simulationen des Handels über das OS4ES ausführen und auswerten. Die Anwendbarkeit beschränkt sich dabei nicht auf den OS4ES-Marktplatz. Es wurde eine allgemeingültiges Simulationsframework entwickelt, das nur geringe Anforderungen an die zu simulierenden Szenarios stellt, um ein möglichst breites Nutzungsspektrum zu ermöglichen.

Für die Umsetzung dieser Aufgabe wurden Framework-Komponenten entworfen, die in sich ein verteiltes System bilden und auf einer Cloud-Infrastruktur betrieben werden. Diese Cloud-Infrastruktur wird durch die Verwendung von Docker als Virtualisierungslösung und Rancher

als Plattformverwaltungslösung umgesetzt. Mithilfe dieser Werkzeuge wird jede Komponente in einem Container isoliert und eine virtuelle Infrastruktur zur Verfügung gestellt. Rancher bindet die von einzelnen Maschinen bereitgestellten Ressourcen in eine Cloud-Infrastruktur und verteilt Container darauf entsprechend der bereitgestellten Ressourcen.
Die entwickelten Framework-Komponenten bieten dem Nutzer die Möglichkeit eine Szenario-Konfiguration über eine Front-End-Komponente vorzunehmen. Dazu wählt der Nutzer die gewünschten Modelle und fügt sie der Szenario-Konfiguration hinzu. Ist diese abgeschlossen kann die Simulation von der Front-End-Komponente aus gestartet werden. Dabei nimmt die zentrale Komponente des Simulationsframeworks, das Deployment-Modul, die Konfigurationsdaten entgegen und startet, die in Simulationscontainern gekapselten, Simulator-Instanzen, auf der Cloud-Infrastruktur.
Die Simulationscontainer enthalten neben den Simulatoren noch einen Kommunikationsrahmen, der Kommunikationsschnittstellen in Form von Webservices zur Verfügung stellt. Gemäß der Zielsetzung aus Kapitel 1.2 sind alle Kommunikationsschnittstellen innerhalb des Simulationsframeworks als Webservices umgesetzt worden.

Zur Ausführung der Simulation wurde eine Schnittstelle entwickelt, die von Simulatoren implementiert werden muss, um eine Kompatibilität zum Simulationsframework zu gewährleisten. Über diese „Sim-API“ genannte Schnittstelle werden Steuerbefehle an Simulatoren übertragen und Kommunikationswege innerhalb des Simulationsszenarios realisiert. Simulatorseitig wurde dazu eine High-Level-API erdacht, die eine Implementation in der jeweiligen Programmiersprache des Simulators benötigt. Über diese kann durch einen einfachen Methodenaufruf eine für den Simulator transparente Kommunikationsschnittstelle angesprochen werden. Um eine Kommunikation innerhalb des Szenarios zu nutzen, werden den Simulatoren die benötigten Endstellen bei der Konfiguration mitgeteilt.

Eine Besonderheit des hier entwickelten Simulationsframeworks ist, dass die Simulationscontainer erst gestartet werden, wenn eine Simulation ausgeführt werden soll. Nach Beendigung dieser werden diese Simulationscontainer wieder beendet. Somit gibt es nur eine geringe Anzahl von Applikationen, die in der Cloud-Infrastruktur dauerhaft betrieben werden. Dies ist vor allem ressourcensparend.

Während der Ausführung einer Simulation wird vom Zeitmodul, einer Komponente des Simulationsmanagers, ein synchroner Zustand der Simulatoren erzeugt und die Simulationszeit gesteuert. Der Simulationsmanager ist die zentrale Komponente bei der Simulationsausführung. Diese sendet alle nötig Steuer- und Konfigurationsdaten an die Simulatoren und ist dafür zuständig, die Simulationsergebnisse in der Datenbank abzulegen, um diese dem Nutzer über das Front-End-Modul zur Verfügung stellen zu können.

Damit wurde ein Design für ein Simulationsframework bereitgestellt, das die gestellten Anforderungen erfüllt und durch die Cloud-basierte Architektur eine dynamische Lastverteilung

ermöglicht. Durch diese Lastverteilung können heterogene Infrastrukturen verwendet werden, um die Cloud aufzusetzen. Im Gegensatz zu anderen Simulationsframeworks die auf Cloud-Infrastrukturen basieren, wird in dieser Arbeit eine freie Lösung verwendet, die auch auf einer eigenen Infrastruktur aufgesetzt werden kann.
Da auch innerhalb des Simulationsframeworks (zwischen Framework-Komponenten) standardisierte Nachrichtentypen verwendet werden, besteht eine lose Kopplung zwischen diesen, was eine Modularität in Hinsicht auf eine folgende Implementation bereitstellt.

Anhand einer prototypischen Implementation des Kommunikationsrahmens wurde ein Ansatz zur Umsetzung des vorgestellten Simulationsframeworks durch eine Software-Implementierung vorgenommen. Anhand der Definition und Beschreibungen aus Kapitel 4 lässt sich eine Implementierung der Framework-Komponenten durchführen. Dabei wurde eine mögliche Implementation des Kommunikationsrahmens und dessen Funktionsweise vorgestellt und visualisiert.

Mit dem in dieser Arbeit entworfenen Simulationsframework wurde eine leichtgewichtige Lösung für Simulationen vorgestellt, die eine Szenario-Architektur mit einem zentralen Verzeichnisdienst (Directory Service) als Vermittlungsstelle benötigen. Durch die festgelegte Szenario-Architektur können Simulationen mit äußert geringem Konfigurationsaufwand durchgeführt werden. Der Entwurf als verteiltes System bietet außerdem die Möglichkeit, die Cloud-Infrastruktur dynamisch an den Simulationsaufwand anzupassen. Sollte die Leistungsfähigkeit der Cloud nicht ausreichen, kann durch Einbindung weiterer Maschinen weitere Leistung zur Verfügung gestellt werden. Ebenso begünstigt die verteilte Architektur Szenarios mit großen Anzahlen von Simulatoren, da keine einzelne Maschine alle Simulatoren betreiben muss.

7.2. Ausblick

In dieser Arbeit wurde der Entwurf eines Cloud-basierten Simulationsframeworks für heterogene Infrastrukturen vorgestellt. Der Entwurf bietet durch den modularen Aufbau das Potential zur Weiterentwicklung einzelner Komponenten. So kann die Unterstützung für weitere Zeitmodelle entworfen werden um, durch eine eventuell eventbasierte Ausführung, eine noch zeiteffizientere Durchführung von Simulationen zu ermöglichen.
Weiterhin bietet das Simulationsmanagement einige Ansätze zur Optimierung. Im momentanen Entwurf ist nicht vorgesehen eine Simulation bei Ausfall von Simulatoren dynamisch zu rekonfigurieren oder Ersatzsimulatoren zu starten. Auch eine Backup-Instanz von Simulator-Typen wäre möglich, die im Falle einer Fehlfunktion direkt mit den Parametern der fehlerhaften Simulatoren parametriert werden kann.
Des Weiteren ist eine Erweiterung des Frameworks um weitere Komponenten denkbar. Eine

Front-End-GUI, also eine grafische Benutzeroberfläche über die Szenario-Konfigurationen vorgenommen und Simulationsergebnisse dargestellt werden können, würde Nutzern den Einsatz des Simulationsframeworks erleichtern. Bei Entwicklung einer solchen grafischen Oberfläche wäre ebenfalls eine Darstellung des Simulationsfortschritts hilfreich sowie eine Vorschau der bisher gewonnenen Simulationsergebnisse.
Da die Simulationsergebnisse aktuell nur als Text dargestellt werden, ist eine Aufarbeitung, die eine grafische Darstellung der Ergebnisse ermöglicht, sinnvoll. Eine solche Funktionalität könnte auch im Front-End-GUI eingebunden werden. Naheliegend wäre eine Web-basierte Lösung, sodass diese über einen Browser nutzbar ist und eine lokale Ausführung eines Front-End-Moduls überflüssig macht. Ein Vorteil dieser Lösung wäre auch, dass ein Fernzugriff auf das Simulationsframework vereinfacht würde.
Zusätzlich ist die Einbindung einer automatisierten Simulator-Image-Erstellung denkbar. Über diese könnten Simulatoren automatisiert in den Simulationscontainer eingebunden und in das Simulator-Repository (siehe Abbildung 4.4) hochgeladen werden.

Um eine Nutzung des Simulationsframework zu ermöglichen ist es notwendig eine prototypische Implementation aller Framework-Komponenten durchzuführen, um das entworfene Simulationsframework auf Funktionalität und etwaige Probleme zu untersuchen. Anhand einer solchen Implementierung könnten Optimierungspotentiale des Ablaufs und der Funktionalitäten analysiert und daraufhin umgesetzt werden.
Zusätzlich ist es für die Anwendung des Simulationsframeworks unabdingbar, dass Simulatoren entwickelt werden, die kompatibel zu dem hier entworfenen Framework sind. Dazu müssen diese die in Abschnitt 4 beschriebene Sim-API implementieren und die definierten Datenmodelle unterstützen.

Abschließend muss festgehalten werden, dass die Entwicklung eines Simulationsframeworks, wie es hier vorgestellt wurde, ein iterativer Prozess ist. Daher kann der hier dargestellte Entwurf als Grundstein einer solchen Entwicklung angesehen werden, die durch wiederholte Optimierungsprozesse zu einem ausgereiften Produkt führt.
Durch das hier entworfene Simulationsframework bietet sich die Möglichkeit das OS4ES-Marktplatzmodell zu testen. Damit kann in groß angelegten Szenarios die Robustheit der OS4ES-Registry überprüft werden, indem eine große Anzahl von DER- und Aggregator-Simulatoren an dieser betrieben werden. In Hinsicht auf die Marktintegration von dezentralen Energieressourcen kann dieses Simulationsframework weitgehend zur Weiterentwicklung und Optimierung des OS4ES und ähnlichen Ansätzen beitragen.
Eine Integration von dezentralen Energieressourcen führt in Zukunft dazu, dass selbst Endverbraucher ihre Flexibilität am OS4ES anbieten können. Auch dies ließe sich in dem hier vorgestellten Simulationsframework einbinden und simulieren. Das entwickelte Framework bietet eine skalierbare, lose-gekoppelte und erweiterbare Lösung, um fortschrittliche Anwendungen aus Wissenschaft und Forschung zu simulieren und validieren.

Literaturverzeichnis

[Amazon Web Services 2017] AMAZON WEB SERVICES, Inc: *AWS: Entdecken Sie Unsere Produkte*. 2017. – URL https://aws.amazon.com/de/?nc2=h_lg

[Anderson u. a. 2014] ANDERSON, Kyle ; DU, Jimmy ; NARAYAN, Amit ; GAMAL, Abbas E.: *GridSpice: A Distributed Simulation Platform for the Smart Grid*. 2014. – URL http://ieeexplore.ieee.org/document/6846273/

[Balasubramanian u. a. 2013] BALASUBRAMANIAN, Raj ; PAUTASSO, Cesare ; CARLYLE, Benjamin ; ERL, Thomas: *SOA with REST: Principles, Patterns Constraints for Building Enterprise Solutions with REST*. 1. Auflage. Prentice Hall, 2013. – ISBN 978-0-13-701251-0

[Behncke 2017] BEHNCKE, Lars O.: *Kontinuierliche Intraday-Fahrplananpassung eines PV-Batteriesystems zur Vermarktung seiner Flexibilität*. 2017

[BMWi 2017] BMWI: *Erneuerbare Energien*. 2017. – URL https://www.bmwi.de/Redaktion/DE/Dossier/erneuerbare-energien.html. – Accessed: 28.04.2017

[Braubach und Pokahr 2011] BRAUBACH, Lars ; POKAHR, Alexander: *Jadex Active Components Framework - BDI Agents for Disaster Rescue Coordination*. 2011. – URL https://vsis-www.informatik.uni-hamburg.de/getDoc.php/publications/441/jadex.pdf

[Braubach und Pokahr 2012] BRAUBACH, Lars ; POKAHR, Alexander: *The Jadex Project: Simulation*. 2012. – URL https://vsis-www.informatik.uni-hamburg.de/getDoc.php/publications/462/jadex_b.pdf

[Braubach u. a. 2004] BRAUBACH, Lars ; POKAHR, Alexander ; LAMERSDORF, Winfried: *Jadex: A Short Overview*. 2004. – URL https://vsis-www.informatik.uni-hamburg.de/getDoc.php/publications/212/jadex_node.pdf

[Braubach u. a. 2006] BRAUBACH, Lars ; POKAHR, Alexander ; LAMERSDORF, Winfried ; KREMPELS, Karl-Heinz ; WOELK, Peer-Oliver: *A Generic Time Management Service for Distributed Multi-Agent Systems*. 2006. – URL

https://vsis-www.informatik.uni-hamburg.de/getDoc.php/publications/272/timeservice_journal_revised.pdf

[Bray 2014] BRAY, Timothy W.: *RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format*. 2014

[Bundesnetzagentur 2011] BUNDESNETZAGENTUR: *Smart Grid und Smar Market*. 2011

[Bundesregierung 2017] BUNDESREGIERUNG: *Energiewende im Überblick*. 2017. – URL https://www.bundesregierung.de/Content/DE/StatischeSeiten/Breg/Energiekonzept/0-Buehne/ma\T1\ssnahmen-im-ueberblick.html. – Accessed: 26.04.2017

[Coley Consulting] COLEY CONSULTING: *MoSCoW Prioritisation*. – URL http://www.coleyconsulting.co.uk/moscow.htm. – Accessed: 24.03.2017

[Dethlefs 2014] DETHLEFS, Tim: *Ein verbraucherorientiertes Energiesystem für Smart Grids*. 2014. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2014/2618/pdf/MA_Dethlefs.pdf

[Dethlefs u. a. 2016a] DETHLEFS, Tim ; BRUNNER, Christoph ; PREISLER, Thomas ; RENKE, Oliver ; RENZ, Wolfgang ; SCHRÖDER, Andrea: Energy Service Description for Capabilities of Dristributed Energy Resources. In: *Energy Informatics*. Springer, 2016

[Dethlefs u. a. 2015a] DETHLEFS, Tim ; PREISLER, Thomas ; RENKE, Oliver ; RENZ, Wolfgang ; LANG, Andreas ; PAWILS, Andrea ; MORILLON, Andre ; PESCHKA, Anne-Kathrin ; WAGNER, Robert: *D4.1: The OS4ES Security and Privacy Concept and the Distributed DER Registry System Architecture*. 2015. – URL http://www.os4es.eu/.cm4all/iproc.php/Downloads/D4-1_final.pdf?cdp=a

[Dethlefs u. a. 2015b] DETHLEFS, Tim ; PREISLER, Thomas ; RENZ, Wolfgang: *An Architecture for a distributed Smart Grid Registry System*. 2015

[Dethlefs u. a. 2016b] DETHLEFS, Tim ; PREISLER, Thomas ; RENZ, Wolfgang: *Towards Adaptive Virtual Power Plants for Smart Markets*. 2016. – URL https://www.vde-verlag.de/proceedings-en/454308037.html

[Dielman und van der Velden 2003] DIELMAN, K. ; VELDEN, A. van der: *Virtual power plants (VPP) - a new perspective for energy generation?* 2003. – URL http://ieeexplore.ieee.org/document/1438108/

[DOCKER a] DOCKER: *Docker Documentation*. – URL https://docs.docker.com. – Accessed: 12.02.2017

[DOCKER b] DOCKER: *Docker Overview*. – URL https://docs.docker.com/engine/understanding-docker/. – Accessed: 10.02.2017

[DOCKER c] DOCKER: *What is Docker?*. – URL https://www.docker.com/what-docker. – Accessed: 11.02.2017

[EPEX SPOT] EPEX SPOT: *Produkte Intraday-Auktion*. – URL http://www.epexspot.com/de/produkte/intradayauction/deutschland. – Accessed: 24.02.2017

[Gellings 1985] GELLINGS, Clark W.: The concept of demand-side management for electric utilities. In: *Proceedings of the IEEE* 73 (1985), Nr. 10, S. 1468–1470

[IEC 60038 2009] IEC 60038: *CENELEC-Normspannungen*. 2009

[IEEE und The Open Group] IEEE ; THE OPEN GROUP: *The Open Group Base Specifications Issue 7*. – URL http://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_xbd_chap04.html#tag_21_04_16. – Accessed: 11.04.2017

[Kent und Seo 2005] KENT, S. ; SEO, K.: *RFC 4301 - Security Architecture for the Internet Protocol*. 2005. – URL https://tools.ietf.org/html/rfc4301

[Konstantin 2013] KONSTANTIN, Panos: *Praxisbuch Energiewirtschaft*. 3. Auflage. Springer, 2013. – ISBN 978-3-642-37264-3

[Lannoye u. a. 2012] LANNOYE, Eamonn ; FLYNN, Damian ; O'MALLEY, Mark: *IEEE: Evaluation of Power System Flexibility*. 2012

[Mandl 2014] MANDL, Peter: *Grundkurs Betriebssysteme*. Springer, 2014. – ISBN 978-3-658-06217-0

[Mellor-Crummey und Scott 1991] MELLOR-CRUMMEY, John M. ; SCOTT, Michael L.: Algorithms for Scalable Synchronization on Shared Memory Multiprocessors. In: *ACM Transactions on Computer Systems* 9 (1991), Nr. 1, S. 21–65

[Next 2016] NEXT: *Was ist Einspeisemanagement?* 2016. – URL https://www.next-kraftwerke.de/wissen/direktvermarktung/einspeisemanagement. – Accessed: 02.05.2017

[OFFIS 2012] OFFIS: *mosaik - Modulare Simulation aktiver Komponenten im Smart Grid*. 2012. – URL https://www.offis.de/offis/projekt/mosaik.html. – Accessed: 14.04.2017

[Pivotal Software 2017a] PIVOTAL SOFTWARE: *Spring Boot*. 2017. – URL https://projects.spring.io/spring-boot/. – Accessed: 20.04.2017

[Pivotal Software 2017b] PIVOTAL SOFTWARE: *Spring Documentation*. 2017. – URL https://spring.io/docs. – Accessed: 20.04.2017

[Pokahr u. a. 2005] POKAHR, Alexander ; BRAUBACH, Lars ; LAMERSDORF, Winfried: *Jadex: a BDI reasoning engine*. 2005. – URL https://vsis-www.informatik.uni-hamburg.de/getDoc.php/publications/250/promasbook_jadex.pdf

[Quaschning 2015] QUASCHNING, Volker: *Regenerative Energiesysteme*. 9. Auflage. Hanser, 2015. – ISBN 978-3-446-44267-2

[RANCHER] RANCHER: *Quick Start Guide*. – URL http://docs.rancher.com/rancher/v1.2/en/quick-start-guide/. – Accessed: 14.02.2017

[Rohjans u. a. 2013] ROHJANS, S. ; LEHNHOFF, S. ; SCHÜTTE, S. ; SCHERFKE, S. ; HUSSAIN, S.: *Mosaik - A modular Platform for the Evaluation of Agent-Based Smart Grid Control*. 2013. – URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6695486

[Scherfke und Schütte 2012] SCHERFKE, Stefan ; SCHÜTTE, Steffen: *Mosaik - Architecture Whitepaper*. 2012. – URL http://mosaik.offis.de/downloads/mosaik_architecture_2012.pdf

[Schröder u. a. 2015] SCHRÖDER, Andrea ; ZAHNER, Martin ; DETHLEFS, Tim ; RENZ, Wolfgang: OS4ES - Offenes System für Energiedienstleistungen. In: *VDE*, 2015

[Schütte u. a. 2012] SCHÜTTE, Steffen ; SCHERFKE, Stefan ; SONNENSCHEIN, Michael: *Mosaik - Smart Grid Simulation API - Toward a Semantic Based Standard for Interchanging Smart Grid Simulations*. 2012. – URL http://mosaik.offis.de/downloads/mosaik_SimAPI_SmartGreens2012.pdf

[Tanenbaum und van Steen 2007] TANENBAUM, Andrew S. ; VAN STEEN, Maarten: *Verteilte Systeme*. 2. Auflage. Pearson, 2007. – ISBN 978-3-8273-7293-2

[UCTE 2004] UCTE: *Operation Handbook*. UCTE, 2004

[Wooldridge 2013] WOOLDRIDGE, Michael J.: *An Introduction to MultiAgent Systems*. 2. Auflage. Wiley, 2013. – ISBN 978-0-470-51946-2

[ØMQ 2014a] ØMQ: *ZeroMQ Feature List*. 2014. – URL http://zeromq.org/docs:features. – Accessed: 18.04.2017

[ØMQ 2014b] ØMQ: *ØMQ Language Bindings*. 2014. – URL http://zeromq.org/bindings:_start. – Accessed: 18.04.2017

A. Weitere Nachrichtentypen

A.1. Definition der Config-Response-Nachricht

Tabelle A.1.: Datenfelder der Config-Response-Nachricht

Name	Inhalt	Datentyp
id	ID	String
status	Konfigurationsstatus	String
message	Fehlernachricht	String

```
{
  "id": "b2f3f6f8-bdb0-4827-8a06",
  "status": "failure",
  "message": "Model could not be configured. Error: java.lang.Error: UnknownError"
}
```

Listing A.1: Beispiel einer Config-Response-Nachricht

A.2. Definition der Step-Response-Nachricht

Tabelle A.2.: Datenfelder der Step-Response-Nachricht

Name	Inhalt	Datentyp
id	ID	String
type	Nachrichtentyp	String
time	Simulatorzeit	Array
time	aktuelle Zeit	String
timestep	aktueller Simulationsschritt	int

```
1 {
2   "id": "bc77a81a-6f79-478e-a08b",
3   "type": "step-finished",
4   "time": {
5     "time": "1492763075950",
6     "timestep": "124",
7   }
8 }
```

Listing A.2: Beispiel einer Step-Response-Nachricht

A.3. Definition der Result-Request-Nachricht

Tabelle A.3.: Datenfelder der Result-Request-Nachricht

Name	Inhalt	Datentyp
id	ID	String
type	Nachrichtentyp	String

```
1 {
2   "id": "123e4567-e89b-12d3-a456",
3   "type": "result-request",
4   }
5 }
```

Listing A.3: Beispiel einer Result-Request-Nachricht

A.4. Definition der Result-Nachricht

Tabelle A.4.: Datenfelder der Result-Nachricht

Name	**Inhalt**	**Datentyp**
id	ID	String
payload	Ergebnisdaten	String-Array

```
1 {
2   "id": "aa1361a9-96e8-474d-8b6b",
3   "payload": {
4     ["Time: 1492761412913, input_price: 0.30, bat_charge:
    0.75, state: healthy",
5
6    "Time: 1492761413913, input_price: 0.36, bat_charge:
    0.55, state: healthy"
7
8    "Time: 1492761414913, input_price: 0.50, bat_charge:
    0.40, state: healthy"
9
10   "Time: 1492761415913, input_price: 0.80, bat_charge:
    0.23, state: low"]
11   }
12 }
```

Listing A.4: Beispiel einer Result-Nachricht

A.5. Definition der Fehlernachricht

Tabelle A.5.: Datenfelder der Fehlernachricht

Name	Inhalt	Datentyp
id	ID	String
location	Name des Fehlerhaften Simulators	String
reason	Fehlergrund	String
message	Fehlertext	String


```
1 {
2   "id": "459a3326-d37c-4b10-8490",
3   "location": "OS4ES_Registry:v2",
4   "reason": "synchronisation error",
5   "message": "Simulator did not respont to synchronization
      signal. terminated.",
6 }
```


Listing A.5: Beispiel einer Fehlernachricht

B. Definition der Datenbankstruktur

B.1. Szenariodatenbank

Tabelle B.1.: Struktur der Szenariodatenbank, Tabelle 1

Key	**Name**	**Inhalt**
Primary	SimID	ID der Simulation
Foreign	TimeframeID	ID des Simulationszeitraums
Foreign	SimulatorslistID	ID der Liste von Simulatoren

Tabelle B.2.: Struktur der Szenariodatenbank, Tabelle 2

Key	**Name**	**Inhalt**
Primary	TimeframeID	ID des Simulationszeitraums
	Start	Startzeitpunkt der Simulation
	End	Endzeitpunkt der Simulation
	Timebase	Zeitbasis der Simulation

Tabelle B.3.: Struktur der Szenariodatenbank, Tabelle 3

Key	**Name**	**Inhalt**
Primary	SimulatorslistID	ID der Liste von Simulatoren
Foreign	SimulatorID	ID des Simulators

Tabelle B.4.: Struktur der Szenariodatenbank, Tabelle 4

Key	**Name**	**Inhalt**
Primary	SimulatorID	ID des Simulators
	Name	Simulatorname
	Typ	Simulatortyp

B.2. Ergebnisdatenbank

Tabelle B.5.: Struktur der Ergebnisdatenbank, Tabelle 1

Key	**Name**	**Inhalt**
Primary	ResultID	ID der Ergebnisliste
Foreign	SimulatorID	ID des Simulators
	Results	Ergebnis-Array

Der Fremdschlüssel dieser Tabelle bezieht sich auf den Primärschlüssel (SimulatorID) von Tabelle B.4. So können die Ergebnisse eindeutig der Simulation zugeordnet werden.

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 11. Mai 2017

Ort, Datum

Unterschrift