Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Skalierung von nebenläufigen und verteilten Simulationssystemen für interagierende Agenten

An der Universität Hamburg eingereichte
Dissertation
zur Erlangung des akademischen Grads
Dr. rer. nat.

Jan Henrik Röwekamp
aus Hamburg

Hamburg, 2023

Dissertation zur Erlangung der Würde des Doktors der Naturwissenschaften der Fakultät für Mathematik, Informatik und Naturwissenschaften, Fachbereich Informatik der Universität Hamburg vorgelegt von Jan Henrik Röwekamp aus Hamburg.

Gutachten wurden erstellt von:

Dr. Daniel Moldt
Prof. Dr. Leonie Dreschler-Fischer
Prof. Dr. Karsten Wolf

Prüfungskomission:

Prof. Dr. Peter Kling (Vorsitz)
Dr. Daniel Moldt
Prof. Dr. Leonie Dreschler-Fischer
Prof. Dr. Norbert Ritter

Tag der Disputation:

22. Juli 2022

Betreuung:

Dr. Daniel Moldt
Prof. Dr. Leonie Dreschler-Fischer

Zusammenfassung

Diese Arbeit behandelt die Integration von dynamischer Skalierbarkeit, dynamischer Anpassung und modellbasierten Simulationssystemen. Diese Simulationssysteme sind dabei auf die nebenläufige Simulation von auf Plattformen interagierenden Agenten ausgelegt. Das zentrale Ergebnis hierbei ist die im Rahmen der Arbeit eingeführte »Mushu-Architektur«. Diese beschreibt in diesem Kontext ein Agentensystem, in welchem Agenten, heterogene Plattformen und das Management von Plattformen in einer integrierten Architektur bestehen. Dabei sind Agenten durch das Design der Architektur explizit in der Lage dynamischen Einfluss auf die Form ihrer Plattformen und deren Anzahl zu nehmen.

In der realen Welt treten Entitäten wie Systeme und insbesondere Menschen stets in einem bestimmten Raum oder Rahmen miteinander in Interaktion. Die Mushu-Architektur beabsichtigt die Möglichkeit dieser Entitäten spontan Räume für die Kommunikation und Interaktion zu schaffen konzeptuell abzubilden. Die agentenorientierte Softwareentwicklung bietet hierfür eine solide Grundlage und interpretiert die genannten Entitäten als Agenten. Die Simulation großer Systeme ist jedoch ab einer gewissen Größe auf einzelnen Maschinen nicht mehr möglich oder kosteneffizient. Daher wurde die Mushu-Architektur unter der Maßgabe entworfen, dass in einer Realisierung die Gesamtarchitektur ein verteiltes System beschreibt und die Plattformen die größten lokalen Einheiten darstellen. Es wird hergeleitet, dass die dynamische Erzeugung von Plattformen durch Agenten als Skalierungsoperation auf der technischen Ebene beschrieben werden kann. Auf technischer Ebene beschreibt die Mushu-Architektur gleichermaßen Anteile von Softwarekomponente bis Deploymentlandschaft.

Als letzter verbleibender Punkt wird die Abbildung von Realweltsituationen auf derartige Modelle adressiert. Eine geeignete Methodik dafür besteht im Kontext der Szenenanalyse, bei welcher aus (Bild)daten und Wissen Hypothesen über beobachtete Situationen abgeleitet werden, passend zum generellen agentenorientierten Ansatz. Da eine umfassende Betrachtung dieses Ablaufs den Umfang mehrerer eigener Arbeiten hätte, motiviert diese Arbeit lediglich diesen Weg. Dabei wird dargelegt, dass es gerade bei umfangreichen Berechnungen vorteilhaft ist, eine solche Abbildung schneller durchführen zu können. Um hierbei einen Beitrag zu leisten, wird eine Heuristik eingeführt, welche grundlegend für die Beschleunigung einer Klasse an Mustererkennungsalgorithmen eingesetzt werden kann.

Die Arbeit verfolgt einen dreistufigen Ansatz von Konzeptualisierung, Realisierungskonzept und Beschreibung von Prototypen. In der Konzeptualisierung wird die abstrakte Architektur beschrieben, im Realisierungskonzept die Abbildung auf bestehende (oder zu konstruierende) Technologieformen und schlussendlich in den Prototypen die Abbildung des Realisierungskonzepts auf konkrete Technologien, Frameworks, etc. sowie erfolgte Implementationen.

Abstract

This thesis addresses the integration of dynamic scalability, dynamic adaptation and model-based simulation systems, which are designed for the concurrent simulation of agents interacting on platforms. The main result is the newly introduced "Mushu architecture". It describes an agent system, in which agents, heterogeneous platforms and the management of platforms exist in an integrated architecture, where agents are explicitly able to dynamically influence their platforms themselves as well as the platform quantity through the design of the architecture.

Based on the real-world observation that entities, such as systems and especially humans, interact with each other in certain spaces or contexts, the Mushu architecture intends to cover the possibility of these entities to spontaneously create spaces for communication and interaction on a conceptual level. Agent-oriented software development provides a solid foundation for this and interprets the entities mentioned as agents. However, the simulation of large systems is no longer possible or cost-efficient for single machines above a certain extent. Therefore, the Mushu architecture was designed with the idea in mind that within a realisation, the overall architecture describes a distributed system and the platforms are the largest local entities. It is then deduced that the dynamic generation of platforms by agents can be described as scaling operations from a technical point of view. On the technical level, the Mushu architecture thus describes different parts ranging from software components to the deployment landscape.

The last remaining aspect is the mapping of real-world situations to such models. A viable methodology for this exists in the context of scene analysis, in which hypotheses about observed situations are derived from (image) data and contextual knowledge, in line with the general agent-oriented approach. Since a comprehensive consideration of this process would take up the scope of several separate theses, this thesis only discusses the approach. In doing so, it is explained that a mapping for large-scale calculations should nevertheless be quick to compute. In order to make a generalised contribution to this aspect, a heuristic is introduced which is fundamental for the acceleration of a class of pattern recognition algorithms.

The thesis follows a three-step approach of a conceptualisation, the creation of a realisation concept and the description of prototypes. In the conceptualisation the abstract architecture is described, the creation of a realisation concept addresses the mapping to existing (or to be constructed) kinds of technology, and finally prototypes apply the realisation concepts to concrete technologies, frameworks, etc., and also describe the implementations that have taken place within the context of the thesis.

Danksagung

Mein Dank geht an meinen Betreuer Dr. Daniel Moldt und meine Betreuerin Prof. Dr. Leonie Dreschler-Fischer für ihre Hilfestellungen im Rahmen der Arbeit. Insbesondere durch die Hinweise, die unzähligen Gespräche und gemeinsamen Projekte mit Dr. Daniel Moldt und der damit einhergehenden exzellenten Unterstützung war ich in der Lage die Thematik der Arbeit in dieser Form bearbeiten zu können. Darüber hinaus danke ich Prof. Dr. Dreschler-Fischer für die vielen Gespräche und die fortlaufende Unterstützung während der Arbeit. Ferner gilt mein Dank Prof. Dr. Peter Kling für die Übernahme des Vorsitzes der Prüfungskomission, Prof. Dr. Karsten Wolf für die Begutachtung und die Unterstützung der Prüfungskomission, sowie Prof. Dr. Norbert Ritter für die Teilnahme an der Prüfungskomission.

Ich möchte weiterhin allen Menschen in meiner Familie für die fortlaufende Unterstützung während der Anfertigung der Dissertation und allen Höhen und Tiefen während des ganzen Promotionsverfahrens danken. Mein Dank gilt dabei meinen Eltern Gaby und Claus, meiner Partnerin Manuela und ihrer Mutter Birgit.

Im weiteren Rahmen der Universität gilt mein Dank allen Dozenten, Doktoranden, Postdocs, Kommilitonen und Studierenden, welche über die Jahre meines Bachelor-, Master- und Promotionsstudiums hilfreiches Feedback geliefert und zu spannenden Diskussionen beigetragen haben. Insbesondere sind dies (ohne spezifische Reihenfolge): Michael Haustermann, David Mosteller, Dennis Schmitz, Michael Simon, Dr. Benjamin Seppke, Dr. Thomas Wagner, Prof. Dr. Michael Köhler-Bußmeier, Eva Müller, Dr. Frank Heitmann, Dr. Matthias Wester-Ebbinghaus, Prof. Dr. Rüdiger Valk, Prof. Dr. Peer Stelldinger, Matthias Feldmann, Felix Beese, Rainer Jürgensen, Jan Robert Janneck und Laif-Oke Clasen. Darüber hinaus gilt mein spezieller Dank Marvin Taube, Patrick Mohr, Henri Engelhardt, Laszlo Korte, Alexander Senger, Sven Willrodt, Laif-Oke Clasen und Manuela Buchholz für die Unterstützung bei der technischen Implementationsarbeit im Rahmen dieser Arbeit.

Dann möchte ich allen Freunden, Familie und Bekannten danken, die sich die Zeit genommen haben diese Arbeit korrekturzulesen: Bennet, Jann, Adrian, Kay, Benni, Peer, Dennis, Voßi, Elly, Gaby, Felix, Laif, Jürgen, Manuela, Sven und Thomas. Zuletzt danke ich noch Hanna für die Ermöglichung äußerst flexibler Arbeitszeiten im letzten Jahr der Anfertigung der Arbeit.

Inhaltsverzeichnis

Teil I.

Ausgangspunkt

1. Einleitung

Bei genauer Betrachtung der Welt und der in ihr agierenden Menschen, Systeme, Software, Tiere und weiteren sowie deren Umgebungen fällt auf, dass sich bestimmte Strukturen an vielen Orten wiederholen. So kann beispielsweise ein Austausch zwischen drei alten Freunden in einem Café stattfinden, ein Mensch eine Katze streicheln, ein Analytiker die soeben berechneten Geschäftszahlen eines Unternehmens lesen oder aber sich ein Smartphone mit einem Nachrichtenserver synchronisieren. Diese auf den ersten Blick sehr verschiedenen Situationen haben jedoch eine gemeinsame Basis. Entitäten, die individuell und für sich entscheiden, planen und handeln, gehen eine gemeinsame Interaktion ein, sie *synchronisieren* sich.

Derartige beschriebene Interaktionen finden immer innerhalb eines Kontextes, bzw. einer Räumlichkeit statt. So befinden sich die Freunde in einem Café, der Mensch mit der Katze beispielsweise im Wohnzimmer seiner Wohnung, der Analytiker im Büro und die Synchronisation zwischen Smartphone und Server gewissermaßen in einem geschlossenen Kommunikationskanal im Netzwerk. In unserer Umgebung geschehen derartige Prozesse millionenfach und zu jeder denkbaren Zeit. Sie folgen dabei keiner gemeinsamen globalen Taktung, sondern geschehen vielfach zeitlich vollständig unabhängig voneinander, sie geschehen *nebenläufig*. Dies bedeutet jedoch keineswegs, dass zwangsläufig alle Aktionen unabhängig voneinander geschehen müssen. So kann der Mensch, der die Katze streichelt, beispielsweise gleichzeitig noch ein Telefonat führen und auf einen stummgeschalteten Fernseher schauen. Einzelne Aktionen können sich auch bedingen und somit in einem Kausalzusammenhang stehen, so könnte das Telefonat aufgrund einer entsprechenden Nachricht auf dem Smartphone initiiert worden sein. Die verschiedenen Aktionen können sich beliebig überlappen.

Jede Entität in diesem globalen System handelt getrieben durch innere Prozesse, je nach Komplexität basierend auf einer Form der Wissensspeicherung. Sie handeln alleine, können sich jedoch auch in Kooperation mit anderen begeben, um gemeinsam eine Handlung zu vollziehen. Dabei treffen sie an einem physischen oder virtuellen Ort – oder allgemeiner gesagt – einer Plattform aufeinander. Beispiele für eine solche Plattform sind gegeben durch Räume, Konferenzen, Telefonate und viele weitere.

Darüber hinaus sind Entitäten in der Lage diese Plattformen zu manipulieren, sie zu erzeugen, zu zerstören bzw. abzubauen und anzupassen. Räume können gebaut, Telefonate gestartet, Internetverbindungen hergestellt und Konferenzen verabredet werden. Die Änderung an Plattformen findet meist indirekt durch die Entitäten statt, so kann z.B. in einem Raum das Licht angeschaltet werden oder das Routing einer Verbindung geändert werden. Bei größeren Änderungen ist meist eine übergeordnete Instanz involviert, welche die Plattform bereitstellt, an die die Anfrage gerichtet werden kann. So könnte ein Unternehmen die konkrete Ausrichtung einer Abteilung ändern oder durch den Gebäudeverwalter eine technische Ausstattung in einem Raum installiert werden. Als Beispiele für die Zerstörung von Räumen kann die Beendigung eines Telefonates, das Ende einer Konferenz, der Abriss eines Gebäudes und weitere genannt werden.

Wenn die Idee der Entitäten und Plattformen weiter abstrahiert wird, können in bestimmten Fällen Plattformen wiederum mit anderen Plattformen kooperieren und sich somit selbst wie Entitäten verhalten. Reale Beispiele sind Abteilungen in einem Unternehmen oder gemeinsam ausgerichtete Konferenzen.

Eine der großen Erleichterungen durch die Informatik besteht darin durch Simulationen bereits im Vorfeld Aussagen über Verhalten unter bestimmten Parametern treffen zu können. Von dieser Technologie profitieren täglich Milliarden von Menschen, sei es durch berechnete Wettervorhersagen, Bahnfahrpläne oder Verkehrsplanung um beispielsweise Ampelschaltungen zu optimieren. Häufig existiert zu jedem speziellen Anwendungsfall eine sehr spezialisierte Simulationssoftware, die mit großer Präzision versucht die Domäne eines spezifischen Problems abzubilden. Simulationen, die allerdings generalisierte Informationen und Vorhersagen zulassen, sind seltener anzutreffen.

Für Simulationen ist es stets sinnvoll ein geeignetes Modell der zu simulierenden Systeme zu erzeugen. Die modellgetriebene Softwareentwicklung ist eines der Resultate aus diesem Umstand. Für die Ableitung von Modellen aus der realen Welt existieren verschiedene Herangehensweisen. Eine mögliche dieser Herangehensweisen ist die Szenenanalyse, welche ihrerseits wieder eine Unterkategorie der Bild(folgen)verarbeitung ist. Die Szenenanalyse beschäftigt sich mit der Extraktion von Hypothesen aus Bildfolgen auf der Basis von Bildinformationen und Kontextwissen. Dabei sind die Hypothesen auf höherer Ebene zu verorten als einfache Bilddaten. So sind beabsichtigte Ergebnisse beispielsweise die Differenzierung zwischen einem Tanz und einem Handgemenge. Je genauer das extrahierte Modell der zugehörigen Welt ausfällt, desto detailliertere Hypothesen können extrahiert werden.

Trotz der offensichtlichen Zusammenhänge zwischen nebenläufigen Handlungen, großer Anzahl der Entitäten und der Möglichkeit die eigene Laufzeitumgebung anzupassen und zu ändern, existieren jedoch bislang wenige Softwarearchitekturen, welche diese Anforderungen umfassend in einem System in Einklang bringen und

eine ausreichende Basis liefern, um daraus Ableitungen zu ermöglichen. Dies ist in erster Linie der hohen Komplexität der Kombination der einzelnen Bestandteile geschuldet. Weiterführende Argumentationen zu dieser These werden in Kapitel 4 geführt.

Da somit noch grundlegende Fragen für die Modellierung dieses Verständnisses von Realweltabläufen für die Interpretation im Rahmen der Szenenanalyse ungeklärt sind, adressiert diese Arbeit die Schaffung einer entsprechenden architekturellen Grundlage. Diese kann sodann sowohl in der Szenenanalyse, aber auch in anderen Anwendungsszenarien eingesetzt werden. Beispielhafte Anwendungsszenarien jenseits der Szenenanalyse werden in Abschnitt 1.2 noch einmal genauer ausgeführt.

Zur Bearbeitung der Aufgabe ist es zunächst angebracht einen entsprechenden Grundansatz der Softwarearchitektur zu betrachten. Die agentenorientierte Softwareentwicklung ist dabei hervorragend geeignet, da sie den sogenannten Softwareagenten autonome Eigenschaften zugesteht. Die Definition eines Softwareagenten ist durch die Literatur hinweg nicht eindeutig, meist wird damit jedoch ein gewisses Maß an Autonomie, Zielorientierung, Interaktionsvermögen und Wissen verbunden. Eine genauere Definition des Begriffes für diese Arbeit folgt in Kapitel 2. Diese Grundeigenschaften machen Agenten zu einem guten Ansatz, um interagierende Entitäten zu modellieren. Folglich bezieht sich diese Arbeit auf das Paradigma der agentenorientierten Softwareentwicklung.

Um den Ansatz weiter eingrenzen und zielgerichteter bearbeiten zu können, werden im Rahmen der Arbeit Agenten im Sinne des Mulan Modells (Rölke, 2004) und dessen Erweiterungen betrachtet. Sie entsprechen damit nach und analog zu (Wagner, 2018) dem Agentenbegriff aus (Wooldridge, 2009, S.26-27). Damit einhergehend umfasst das betrachtete Agentenmodell keine kognitiven Konzepte oder solche, welche künstlicher Intelligenz entsprechen würden. Insbesondere Agenten nach dem BDI-Ansatz (»Believe«, »Desire«, »Intention«) (Bratman, 1987; Rao und Georgeff, 1991) und Verwandte sind damit außerhalb des Umfangs der Arbeit, wenngleich sich die BDI-Konzepte und die Intelligenz in Mulan konzeptionell gut eingliedern (Rölke, 2004).

Zahlreiche andere Arbeiten setzen bereits auf dem Mulan-Ansatz auf, wie beispielsweise (Reese, 2009), (Wester-Ebbinghaus, 2010), (Cabac, 2010), (Wagner, 2018) und weitere. Analog zu diesen Publikationen setzt auch diese Arbeit als Modellierungstechnik Petrinetze und im speziellen Referenznetze (Kummer, 2002) ein.

Petrinetze eignen sich hervorragend als Modellierungswerkzeug für nebenläufige Systeme und auch verteilte Systeme. Durch ihre strikte Trennung von aktiven und passiven Komponenten und auch durch die Grundidee der Marken sind sie in der Lage, Ressourcen und Abläufe prägnant zu modellieren. Petrinetze untergliedern

sich in eine Vielzahl verschiedener Formalismen, bei denen die Referenznetze innerhalb dieser Arbeit eine gesonderte Stellung einnehmen. Je nach Kontext bzw. Bedarf oder Vermögen der Werkzeuge können verschiedene Semantiken wie die (echte) Nebenläufigkeit (Petri, 1962), die Schritt- oder Interleaving-Semantik zur Simulation von Petrinetzen zum Einsatz kommen.

Referenznetze gehören den Objektnetzen an, basieren auf Netzen-in-Netzen (Valk, 1998) und integrieren objektorientierte Netze (Moldt, 1996) und synchrone Kanäle (J. Christensen, 1994). Zugehörige formale Arbeiten bestehen durch (Köhler, 2004) und (Heitmann, 2013). Weitere Beiträge zum Thema Objektnetze umfassen beispielsweise die Publikationen (Buchs, Flumet und Racloz, 1992; Lakos, 1995; Buchs und Guelfi, 2000; Farwer und I. Lomazova, 2001; Irina Lomazova und Ermakova, 2016). Referenznetze können auf einfache Art und Weise inhärent hierarchische, nebenläufige Systeme kompakt modellieren. Eben diese Art von Systemen stellen Agenten, welche auf Plattformen miteinander interagieren, dar.

Während Petrinetze einen formalen Rahmen vorgeben und viele Beweise und Ansätze der Verifikation für einfache Klassen der Petrinetze bestehen, sind solche Verfahren bei Referenznetzen Gegenstand aktueller Forschung. Entsprechende Grundlagen wurden bereits in den Arbeiten Kummers (Kummer, 2002) auf der Basis von Graphtransformationssystemen und weiteren formalen Hintergründen geschaffen. Aktuellere Arbeiten umfassen beispielsweise die Beiträge (Willrodt, 2019), (Willrodt, Moldt und M. Simon, 2020) und (Engelhardt, 2020). Entsprechende Untersuchungen bildeten die Grundlage der Implementation des Referenznetzsimulators Renew (Kummer und Wienberg, 1999b; Kummer, Wienberg, Duvigneau, Cabac u. a., 2020a), in dessen Kontext auch eine technische Implementation der Mulan-Architektur erfolgte. Deshalb und da die Untersuchungen aktuell aktiv vorangetrieben werden, wird argumentiert, dass der fehlende umfassende formale Rahmen von Mulan und von Referenznetzen in Kauf genommen werden kann, um das Grundmodell der Referenznetze und ihren hierarchischen Modellierungseigenschaften dennoch einsetzen zu können. Dies gilt insbesondere, wenn es sich um praktische und angewandte Modellierung handelt. Ferner sind aus diesem Grund Ansätze formaler Verifikation und ähnliches nicht Teil des Umfangs dieser Arbeit.

Der Petri net-based, Agent- and Organization-Oriented Software Engineering(Paose)-Ansatz nach (Moldt, 2005) (Cabac, Dörges, Duvigneau, Moldt u. a., 2008) (Cabac, 2010) erweitert das Verständnis des Agentensystems auf die Entwicklung derartiger Multiagentensysteme und etabliert es als übergreifendes Leitbild (Cabac, 2007). Der Paose-Ansatz ist zugeschnitten auf die Entwicklung im Capa-Umfeld (Concurrent Agent Platform Architecture) (Duvigneau, 2002), (Duvigneau, Moldt und Rölke, 2002),

(Duvigneau, Moldt und Rölke, 2003), wobei Capa eine FIPA[1]-konforme spezifische Implementation von Mulan darstellt. Paose selbst ist auch allgemeiner einsetzbar. Capa wird im Rahmen der Arbeit zwar referenziert, dient jedoch nicht als direkte Grundlage. Das generelle Verfahren in der Erstellung der Arbeit ist an den Paose-Ansatz angelehnt, setzt ihn aber nicht vollumfänglich um, da die Arbeit lediglich Überschneidungen mit dem Rahmen von Capa aufweist. Einzelne Artefakte wie beispielsweise Agenteninteraktionsprotokolle (AIPs) zur Beschreibung von Teilsystemen und Abläufen kommen aber dennoch zum Einsatz. Umgekehrt stellt diese Arbeit jedoch in Hinblick auf eine konzeptionelle und softwaretechnische ergänzende Verbesserung der Modellierung, Verteilung und Nutzung einen Beitrag zur Erweiterung des Paose-Ansatzes dar.

Eine Besonderheit in der Simulation derartiger agentenorientierter Systeme im Kontext der Szenenanalyse ist der Umstand, dass Algorithmen im direkten Bereich der Bildverarbeitung meist entweder auf reine Umsetzbarkeit untersucht werden oder aber den harschen Anforderungen einer Echtzeitanwendung unterliegen. Beide Ansätze sind unglücklich für den hier intendierten Einsatzzweck der Abbildung und Simulation realer Prozesse. Übermäßig langsame Algorithmen bremsen die Simulation aus und Echtzeitalgorithmen bringen größere Opfer als es für den Kontext notwendig wäre.

1.1. Ziele der Arbeit und Forschungsfragen

Während viele der im vergangenen Abschnitt angesprochenen Bereiche interessante Untersuchungsfelder darstellen können, ist jedoch der Umfang einer jeden Arbeit auf ein spezifisches Thema eingegrenzt. Für diese Arbeit wird daher im Kontext der bisherigen Ausführungen zu interagierenden Agenten und deren Wirkung auf die Plattformen, auf denen sie interagieren, und insbesondere die Erzeugung und Zerstörung dieser Plattformen durch die Agenten, die folgende Forschungsfrage gestellt:

Wie können, entsprechend realweltlichen Interaktionen, nebenläufige, interagierende Agenten mit Einfluss auf die Plattformen, auf denen ihre Interaktion stattfindet, sowie auf deren Gesamtmenge als Modell bzw. Architektur eines verteilten Systems, beschrieben werden?

Die Forschungsfrage umfasst damit zwei Hauptbestandteile: Den Einfluss von Agenten auf die Plattform sowie den Einfluss der Agenten auf die Skalierung der Menge der verfügbaren Plattformen.

[1] Foundation for Intelligent Physical Agents

Während der erste Teil im Kern die Interaktion und dynamische Modifikation einer einzelnen Plattform betrifft, adressiert der zweite Teil die Interaktion und dynamische Modifikation der Gesamtmenge der Plattformen. Die Dynamisierung der Plattformen führt jedoch unweigerlich zu potentiell problematischer Kommunikation zwischen Agenten. Plattformen könnten durch Skalierungsoperationen temporär oder permanent verloren gehen. Die klassischen Ansätze für Kommunikation (Nachrichtenzustellung) in verteilten Multiagentensystemen könnten dabei auf Probleme stoßen. Dies betrifft insbesondere Interaktionen, welche auf bilateralem Informationsaustausch basieren, da hierbei meist eine gewisse Form der Garantie für Zustellungsoperationen nötig ist.

Aus diesen Herleitungen soll daher die zweite betrachtete Forschungsfrage lauten:

Wie kann eine nebenläufige bilaterale Kommunikation zwischen Agenten gewährleistet werden, wenn ihre Plattformen zur Interaktion potentiell flüchtige Entitäten sind?

Trotz der natürlichen Relevanz der Forschungsfragen existieren – wie sich in Abschnitt 4.3.4 zeigen wird – bisher nur unzureichende Lösungen. Auf die Relevanz wird in Abschnitt 1.2 noch weiterführend eingegangen. Existierende Lösungen decken zumeist wenigstens einen der betrachteten Aspekte nicht ab: Modellbasierte Entwicklung, echte Nebenläufigkeit, Autonomie der Entitäten, Einfluss der Entitäten auf Plattformen oder die Modellierung von lokalen und verteilten Anteilen des Systems. Weitere Informationen zu dieser Aussage werden in Kapitel 3 zum Stand der Forschung sowie in Kapitel 4 zur Anforderungsanalyse aufgearbeitet.

Um möglichst universell einsatzfähig zu sein, sollte das Architekturkonzept der Forschungsfragen möglichst abstrakt gehalten werden und nicht auf konkrete technische Lösungen eingehen. Eine Umsetzung einer entsprechenden Architektur, wie sie von den Forschungsfragen gefordert wird, umfasst zwangsläufig den Einsatz mehrerer physikalischer Recheneinheiten und damit die Konstruktion eines verteilten Systems. Die Konstruktion verteilter Systeme ist im besonderen Maße anspruchsvoll. Unter diesem Aspekt ist es daher angebracht neben dem abstrakten Architekturkonzept ebenso ein Realisierungskonzept vorzustellen, welches als Bindeglied zwischen technischer Implementation und abstrakter Spezifikation dient. Auf Basis des Realisierungskonzepts können dann Proof-of-Concept Implementationen (Prototypen) erfolgen. Die Gliederung der Arbeit folgt diesen Überlegungen. Eine genaue Übersicht erfolgt am Ende der Einleitung in Abschnitt 1.3.

Jenseits der direkten Anforderungen der aufgestellten Forschungsfragen wurde bereits motiviert, dass die moderate Beschleunigung von Bildverarbeitungsprozessen ein wenig untersuchtes Feld ist. Aus diesem Grund soll eine weitere Forschungsfrage angeschlossen werden, welche sich jedoch nur im Kontext der Anwendungsdomäne der Szenenanalyse formulieren lässt:

Ist es möglich, für einige Bildverarbeitungsalgorithmen grundlegende und beschleunigende Heuristiken zu definieren, deren Leistung zwischen reiner Umsetzbarkeit der Aufgabe und den Anforderungen einer Echtzeitanwendung liegt?

1.2. Anwendungsbeispiele

Die generelle Abbildung der Forschungsfragen auf realweltliche Zusammenhänge wurde bereits in den vergangenen Abschnitten erläutert. Dennoch ist zur Illustration die Darstellung entsprechender realweltlicher Beispiele hilfreich. Um zu motivieren, wie ein Ergebnis in diesem Bereich bedeutsame Verbesserungen mit sich ziehen kann, werden im Folgenden einige Anwendungsszenarien umrissen.

Krankheitsstand ausgleichen In diesem Szenario können Mitarbeiter für andere, erkrankte Mitarbeiter einspringen und deren Aufgaben übernehmen. Dabei sind die Fähigkeiten der Vertretung bezogen auf die übernommene Aufgabe jedoch nicht so stark ausgeprägt wie bei dem ausgefallenen Mitarbeiter. Die Arbeit erfordert den Austausch mit Kollegen und die Interaktion mit der Arbeitsstätte. Neben den weitestgehend festen Arbeitsstätten interagieren Kollegen auch miteinander und schaffen sich zu diesem Zweck dynamisch Plattformen wie Telefonverbindungen oder Meetings. Bei größeren Unternehmen ist eine Kooperation verschiedener Arbeitsstätten denkbar (Fertigungsstraßen). Mit einem Modell der Belegschaft und den Arbeitsstätten könnten verschiedene Vertretungs-Szenarien exploriert werden, um gute Lösungen für die Vertretungspläne vorauszuberechnen. Als weitere untersuchbare Eigenschaften in der Simulation sind Verletzungsrisiken, soziale Dynamiken zwischen den Mitarbeitern, Zufriedenheit und andere denkbar.

Kooperierende Roboter Hierbei werden mehrere autonome kooperierende Roboter betrachtet. Dabei kann die Aufgabenstellung vielfältig sein. So könnte ein Roboter Waren aus einem Lagerregal heben und einem zweiten Roboter zielsicher übergeben, sodass letzterer diese dann zur Versandstation bringt, wo er mit einem weiteren Roboter kooperiert. Für Interaktionen erzeugen oder nutzen die Roboter dynamisch Plattformen wie Räume oder den Aufbau einer Verbindung mit einer Kommunikationstechnologie.

Verkehrsfluss Eine andere Anwendung sind in diesem Kontext Kraftfahrzeuge auf der Straße. Durch die Kooperation der einzelnen Individualfahrzeuge entsteht ein globaler Verkehrsfluss. Durch eine skalierbare Simulation der Situation können Verkehrsdichten vorausberechnet werden, während auf den individuellen Charakter der Einzelfahrzeuge eingegangen werden kann. Klassische Verkehrssimulationen sind gezwungen an dieser Stelle auf

Vereinfachungen wie die Aufgabe der Modellierung inidivideller Verkehrsteilnehmer zurückzugreifen, um die Geschehnisse in überschaubare mathematische Formeln zu überführen. Auch Änderungen an der Straßenführung wären denkbar durch die Möglichkeit der Agenten Einfluss auf die Plattform zu nehmen. So wären dynamische Anpassungen im Modell möglich. Entsprechende Modelle können durch Verkehrskameras oder Fahrzeuge mit entsprechenden Sensoren auf der Basis der Szenenanalyse unterstützt werden, um bestimmte Situationen zu erkennen.

Balancierung von Gesellschafts- oder Onlinespielen Kompetitive Spiele erhalten ihre Spannung dadurch, dass die Chancen der Spielteilnehmer ausgeglichen sind, egal mit welchen der auswählbaren Startbedingungen sie jeweils beginnen. Die Balancierung von Spielen ist in sich keine leichte Aufgabe; eine Vorhersage zu treffen, welche Ausprägung die Gesamtheit der (späteren) Spieler haben wird, ist äußert schwierig. Durch die Modellierung der interagierenden Spieler und der Codierung der Spielregeln kann durch eine agentenorientierte Simulation jedoch darüber eine Aussage getroffen werden, indem die Simulationsläufe in Bezug auf Gewinnhäufigkeit analysiert werden. Durch die dynamische Erzeugung von Plattformen können Kommunikationen im und neben dem Spiel modelliert werden.

Simulation von Märkten Märkte sind komplexe Systeme mit vielen autonom agierenden Teilnehmern. Handel kann zwischen verschiedenen Teilnehmern auf unterschiedlichen (Handels)plattformen stattfinden, Teilnehmer können in direkten Austausch treten und (Handels)plattformen weisen unterschiedliche Ausprägungen aus. Mit der Modellierung entsprechender Agenten könnten sich mit Simulationen Vorhersagen über Marktverhalten treffen lassen. Die Hauptschwierigkeit besteht selbstverständlich in der korrekten Modellierung der Agenten und ihrer jeweiligen Verhaltensmuster. Mit einer Architektur, wie sie in dieser Arbeit angestrebt wird, wäre ein konkreter Rahmen und eine Realisierungsmöglichkeit einer naturgetreuen Abbildung geschaffen.

Allen Beispielen ist gemein, dass die Erschließung neuer Kommunikationskanäle bzw. Plattformen der abgebildeten Entitäten maßgeblichen Einfluss auf die jeweils skizzierte Simulation hat. Auch die Flüchtigkeit dieser Plattformen spielt eine entscheidende Rolle bei dem Verlauf der Simulation. Mit dem Bestehen einer entsprechenden Architektur, welche diese Eigenschaften adressiert, könnte ein Beitrag für die genannten Anwendungsszenarien geleistet werden.

1.3. Aufbau der Arbeit

Die Arbeit gliedert sich in fünf Hauptbestandteile. Teil I führt in Kapitel 2 zunächst die notwendigen Grundlagen ein, um eine Diskussion bestehender Lösungen zu ermöglichen. In Kapitel 3 werden danach bestehende Lösungen und bereits im erweiterten Kontext der Forschungsfragen erfolgte Forschung diskutiert. Kapitel 4 analysiert und evaluiert diese Lösungen insbesondere im Kontext der Forschungsfragen und zeigt nötige Arbeiten für die Beantwortung der Forschungsfragen auf. Ferner wird dabei das generelle Vorgehen hierzu ausgeführt.

Anschließend widmet sich Teil II der konzeptuellen Aufarbeitung einer Architektur für interagierende Agenten mit Einfluss auf die Skalierung ihrer Plattformmenge. Entlang der Forschungsfragen adressiert Kapitel 5 die Kommunikation zwischen Agenten, Kapitel 6 die Interaktion mit der Plattform durch Agenten und Kapitel 7 konzipiert eine Verwaltungsebene für Plattformen selbst. Abschließend fasst Kapitel 8 die Ergebnisse in eine kombinierte Architektur zusammen.

Ausgehend von der in Kapitel 8 zusammengefügten Architektur beschreibt Teil III eine Abbildung der abstrakten Architektur auf verschiedene Bausteine und Systemkomponenten in der Form von Realisierungskonzepten. Ein Realisierungskonzept ist dabei nicht als Pflichtenheft der Wasserfall-Softwareentwicklung zu verstehen, sondern als konzeptuelle Zusammenstellung verschiedener technologischer Lösungsansätze mit dem Ziel der Umsetzung der Architektur. Dabei wird eine etwas niedrigere Abstraktionsebene eingenommen und der Einsatz von Renew als Referenznetzsimulator vorgegeben, jedoch darüber hinaus keine Einschränkung auf konkrete Technologien vorgenommen. Kapitel 9 erörtert zunächst die technischen Rahmenbedingungen bestehender Lösungen und Anknüpfungspunkte. Auf dieser Basis diskutiert Kapitel 10 eine mögliche Realisierung der Verwaltungsebene von Plattformen, Kapitel 11 die Realisierung einer entsprechenden Schnittstelle zu Plattformen und Kapitel 12 eine mögliche Umsetzung der Agentenkommunikation. Zuletzt adressiert Kapitel 13 die letzte Forschungsfrage im Kontext der Anwendungsdomäne der Szenenanalyse. Das Ziel hierbei ist eine grundlegende und beschleunigende Heuristik für häufig in dem Kontext eingesetzte Algorithmen.

Abschließend widmet sich Teil IV der Evaluation der Untersuchungen. In Kapitel 14 folgt eine Beschreibung konkreter Prototypen und Implementationen im Rahmen der Arbeit. Dabei handelt es sich um konkrete technische Ausgestaltung der Realisierungskonzepte (oder Vorläufer davon) aus Teil III. Anschließend evaluiert Kapitel 15 die Ergebnisse der Arbeit bezogen auf die Forschungsfragen.

Im letzten Teil V folgt in Kapitel 16 eine Gesamtzusammenfassung der Arbeit und Kapitel 17 wirft neben einem Gesamtfazit einen Blick auf mögliche zukünftige und anschließende Arbeiten.

2. Grundlagen

Zunächst soll eine Orientierung der möglichen Umsetzungsmöglichkeiten der Fragestellung erfolgen. Dabei werden verschiedene Vorarbeiten beleuchtet und vorgestellt. Das Ziel dieses Kapitels umfasst die Bereitstellung eines grundlegenden Rahmens für die Analyse und Beurteilung bestehender Arbeiten im Kontext der Forschungsfragen sowie darin die Basis für die darauf folgenden Teile der Arbeit zu schaffen. Dieses Kapitel enthält *keine* Neuentwicklungen des Autors, sondern lediglich eine Aufarbeitung von Vorarbeiten für einen generellen Kontext der Arbeit.

2.1. Nebenläufigkeit, Modelle und Simulationen

Zunächst sollen einige der zentralen Begriffe der Arbeit eingeführt werden. Dabei handelt es sich um den Begriff der Nebenläufigkeit, des Modells und der Simulation. Diese sind erforderlich, um weitere darauf aufbauende Konzepte zu erörtern.

2.1.1. Nebenläufigkeit und Parallelität

Im Laufe der Arbeit werden die Begriffe »Nebenläufigkeit« und »Parallelität« häufig fallen. In vielen Referenzen werden die beiden Begriffe synonym verwendet. Während dies in vielen Situationen sinnvoll ist, sollen beide Begriffe im Rahmen dieser Arbeit *unterschiedliche Konzepte* beschreiben. Die Nebenläufigkeit wird im Sinne der Nebenläufigkeitstheorie Petris verstanden (Petri, 1962).

Intuitiv beschreiben beide Begriffe zwei Ereignisse, welche gemeinsame vorausgehende Ereignisse und/oder gemeinsame nachfolgende Ereignisse besitzen oder aber in keinerlei derartiger Beziehung stehen und sich selbst gegenseitig nicht in eine zeitliche Abfolge oder in Reihenfolge bringen lassen. *Parallelität* sieht dabei jedoch einen globalen Takt bzw. Schrittgeber vor, sodass Ereignisse in diskreten, gleichen Takten (Schritten) auftreten können. *Nebenläufigkeit* hingegen sieht diese Form von Taktung nicht vor, sodass nebenläufige Ereignisse gänzlich unvergleichbar sind. *Nebenläufigkeit* wird im Rahmen der Arbeit gelegentlich auch als »echte Nebenläufigkeit« bezeichnet, um gesondert auf diesen Umstand hinzuweisen. *Ech-*

te Nebenläufigkeit und *Nebenläufigkeit* bezeichnen im Rahmen der Arbeit jedoch ein und dasselbe Konzept.

Beispiele für Parallelität sind häufig anzutreffen, insbesondere in verteilten Systemen kommt Parallelität häufig in der Form von Nachrichtenrunden zum Einsatz. Auch ein zentraler Taktgeber für Arbeitsschritte in einem Prozessor oder einem verteilten System entspricht dem Ansatz der Parallelität. Während Parallelität die zeitliche Abfolge diskretisiert und diese somit deutlich leichter handhabbar macht, wirkt die Abbildung von Realweltinteraktionen darauf wenig realistisch.

Konversationen und weitere Interaktionen aller Entitäten der Welt unterliegen keiner globalen Taktung. Aus diesem Grund soll im Rahmen dieser Arbeit das Konzept der *Nebenläufigkeit* eine federführende Rolle einnehmen.

2.1.2. Modell

Durch den Einsatz von Nebenläufigkeit ist eine formal mathematische Beschreibung komplexer Systeme schwierig. Um die umfassenden Untersuchungen für den Rahmen einer einzelnen Arbeit handhabbar zu erhalten, soll innerhalb der Arbeit ein modell- und simulationsgetriebener Ansatz verfolgt werden. Zu diesem Zweck soll zunächst der Modellbegriff definiert werden.

Stachowiak definiert Modelleigenschaften hierbei beispielsweise wie folgt:

> *Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können. [...] Modelle erfassen im allgemeinen nicht alle Attribute des durch sie repräsentierten Originals, sondern nur solche, die den jeweiligen Modellerschaffern und/oder Modellbenutzern relevant scheinen. [...] Modelle sind ihren Originalen nicht per se eindeutig zugeordnet. Sie erfüllen ihre Ersetzungsfunktion a) für bestimmte - erkennende und/oder handelnde, modellbenutzende Subjekte, b) innerhalb bestimmter Zeitintervalle und c) unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen.*
>
> — (Stachowiak, 1973, Seiten 131-133)

Für die Repräsentation der im einleitenden Teil der Arbeit beschriebenen Kommunikationen bilden Modelle die Abbildung der Kommunikationsteilnehmer sowie der Plattformen der Kommunikation auf verarbeitbare Konzepte. Diese Modelle sollen so konstruiert sein, dass über sie Rückschlüsse auf reale Zusammenhänge möglich sind.

2.1.3. Simulation

Die »Simulation« bezeichnet eine Nachahmung bzw. Nachbildung realtweltlicher Systeme und Zusammenhänge. Gewöhnlich basiert sie auf einem Modell und ahmt die Ausführung bestimmter Interaktionen der Komponenten des Modells nach.

Oberkampf und Roy definieren:

> *Simulation: the exercise or use of a model to produce a result. (dt.: Simulation: Die Ausübung oder Verwendung eines Modells, um ein Ergebnis zu erzielen.)*
> — (Oberkampf und Roy, 2010, Seite 92)

Dementsprechend soll auch der Simulationsbegriff im Rahmen der Arbeit verstanden werden.

Eine besondere Form der Simulation soll an dieser Stelle noch eingeführt werden. Während Simulationen eingesetzt werden können, um verteilte Systeme zu simulieren, findet in vielen Fällen die Simulationen der Systeme selbst wiederum lokal und unverteilt statt. Die Sinnhaftigkeit dieses Vorgangs wird aus den oben genannten Definitionen von Modell und Simulation ersichtlich: Aus der Simulation sollen Rückschlüsse auf das Verhalten des simulierten Systems ermöglicht werden. Hierzu ist eine echte Verteilung nicht immer unbedingt nötig. Die Verteilung kann ebenfalls simuliert werden.

In Fällen großer Simulationen genügen einzelne Rechner jedoch unter Umständen nicht oder erzeugen unverhältnismäßig hohe Kosten. Unter Umständen lassen sich Aspekte der Verteilung jedoch auch nur unzureichend simulieren. In diesem Fall ist es sinnvoll die Simulation selbst ebenfalls verteilt auszuführen; es ist die Rede von *verteilter Simulation verteilter Systeme*. Die Schnitte zwischen verschiedenen lokalen Einheiten der Simulation können sich dabei an den Schnitten des Originals orientieren, müssen dies jedoch nicht zwangsläufig. Entlang der Argumentation verteilte Systeme auch lokal simulieren zu können, liegt es häufig nahe, mehr Operationen lokal auszuführen als es im Original der Fall wäre, da wie beschrieben ein Modell einen Ausschnitt des Originals abbildet.

Diese Arbeit beschäftigt sich im Wesentlichen mit der *verteilten Simulation verteilter Systeme*.

2.2. Petrinetze

Petrinetze sind eine Modellierungstechnik für nebenläufige Prozesse (Petri, 1962, 1977, 1987). Häufig wird bei Modellierungsaufgaben die Unified Modeling Lan-

guage[1] (UML; dt. Vereinheitlichte Modellierungssprache) eingesetzt. Neben ihren Stärken liefert sie jedoch keine formale Fundierung und auch keine operationale Semantik. Ebenso ist keine interaktive Simulation auf der Basis von UML möglich. Aus diesem Grund wird argumentiert, dass Petrinetze als Modellierungstechnik besser für den Kontext dieser Arbeit geeignet sind als die UML.

Die folgenden Ausführungen richten sich im Wesentlichen nach der Beschreibung von Petrinetzen in (Girault und Valk, 2003). Petrinetze können das Erzeugen und Verbrauchen von Ressourcen modellieren und werden meist als bipartiter, gerichteter Graph dargestellt, bestehend aus den disjunkten Mengen der Plätze und der Transitionen. Im Laufe der Arbeit werden Petrinetze auch verkürzt nur als *Netze* bezeichnet. In Plätzen können Marken existieren, wobei die Gesamtmenge aller vorhandenen Marken in allen Plätzen den aktuellen Zustand des Systems beschreibt. Transitionen können Marken von Plätzen abziehen und Marken in ihnen ablegen. Diese Aktion wird gemeinhin als das *Feuern* oder *Schalten* einer Transition bezeichnet. Durch wiederholtes (nebenläufiges) Feuern von Transitionen kann eine Simulation des durch das Petrinetz abgebildeten Modells erfolgen.

Eine fundamentale Eigenschaft von Petrinetzen ist die Nebenläufigkeit, welche besagt, dass verschiedene Transitionen in einem Netz in vollständiger zeitlicher Unabhängigkeit voneinander feuern können, sofern genug Marken zur Verfügung stehen. Selbst ein und dieselbe Transition kann nebenläufig zu sich selbst feuern.

An dieser Stelle sei erneut auf den Unterschied zwischen *Nebenläufigkeit* und *Parallelität* hingewiesen, welcher in Abschnitt 2.1.1 eingeführt wurde und so durchweg in dieser Arbeit Verwendung findet. Parallelität beschreibt innerhalb dieser Arbeit immer ein getaktetes Verhalten, bei dem alle an dem globalen Prozess beteiligten Knoten in vorgegebenen Intervallen Operationen ausführen. Bei der Nebenläufigkeit hingegen entfällt diese Bedingung der Taktung und zwei Operationen können mit vollständig beliebigem zeitlichen Delta zueinander erfolgen.

Formal ist das Verhalten von Petrinetzen durch verschiedene Prinzipien definiert. Das *Prinzip der Dualität* besagt wie eingangs beschrieben, dass zwei disjunkte Mengen an Grundelementen existieren: Die Stellen, in denen Marken abgelegt sein können, welche wiederum Daten bzw. Ressourcen modellieren können, sowie die Transitionen, welche beim Feuern Marken aus Stellen entnehmen und in einer einzelnen Aktion in andere Stellen hineinlegen können. Wichtig ist dabei, dass die Marken von der Transition stets verbraucht und erzeugt werden und nicht etwa verschoben. Auch die Anzahl der erzeugten und verbrauchten Marken einer einzelnen Transition muss nicht identisch sein.

Das *Prinzip der Lokalität* besagt, dass das Verhalten einer Transition ausschließlich durch die Gesamtheit ihrer Eingangs- und Ausgangselemente (der *Lokalität*)

[1] Siehe auch: https://www.omg.org/spec/UML/ - Zuletzt abgerufen am 24.01.2022

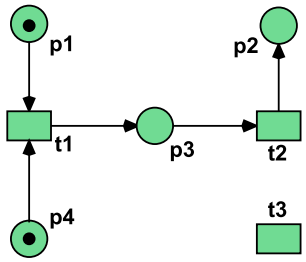

Abbildung 2.1.: Ein Beispiel für ein Petrinetz

bestimmt wird. Das *Prinzip der Nebenläufigkeit* besagt, dass Transitionen mit einer disjunkten Lokalität vollständig unabhängig voneinander schalten können.

Darüber hinaus existieren zwei grundlegende Arten und Weisen Petrinetze darzustellen: Zum einen die grafische Darstellung, bei der nach Konvention die Menge von Plätzen durch Kreise und die Menge von Transitionen durch Rechtecke und die Zuordnung als Eingangs- oder Ausgangselement durch eine gerichtete Kante dargestellt werden. Zum anderen eine formaltextuelle Darstellung, bei der die Spezifikation eines Netzes mithilfe von mathematischen Konstrukten wie Mengen angegeben wird. Dabei ist vor allem darauf zu achten, dass die beiden Darstellungsformen jeweils zueinander äquivalent sind. Beide Darstellungsformen haben ihre jeweiligen Vorteile, so sind Beweise auf der formaltextuellen Darstellungsform wesentlich einfacher zu führen, während Beispiele in grafischer Darstellung um ein Vielfaches anschaulicher sind.

Beispiel 2.1. In Abbildung 2.1 befindet sich ein Beispiel für ein Petrinetz in grafischer Darstellung. In $p1$ und $p4$ steht jeweils eine Marke zur Verfügung, sodass $t1$ feuern und eine Marke in $p3$ ablegen kann. $t2$ ist nun aktiviert und kann schalten und $p3$ wieder leeren, da der Vorbereich von $t2$, welcher nur $p3$ umfasst, nun eine Marke beinhaltet. Nach dem Feuern legt $t2$ eine weitere Marke in $p2$ ab. $t3$ kann unabhängig und beliebig oft zu jeder Zeit nebenläufig feuern.

Darüber hinaus wird im Folgenden eine formale Definition von Netzen erfolgen. Diese ist entnommen und übersetzt aus (Girault und Valk, 2003, Seite 14), ist aber für den Kontext der Arbeit auf endliche Netze angepasst.

Nicht endliche Netze sind für die Modellierung nicht endender Prozesse durchaus üblich. Tiefergehende semantische Betrachtungen in Hinblick auf das Prozessverhalten sind jedoch nicht Teil dieser Arbeit, sodass eine Einschränkung auf endliche Netze erfolgen kann. Alle Netzmodelle, welche als explizite Repräsentationen von Systemen zum Einsatz kommen, sind endlich.

Definition 2.2 (Netz)**.** Ein Netz ist ein Tripel $N = (P, T, F)$, wobei:

- P eine [endliche] Menge an Plätzen ist.
- T eine [endliche,] von P disjunkte Menge an Transitionen ist.
- $F \subseteq (P \times T) \cup (T \times P)$ eine Flussrelation für die Menge der Kanten ist.

— (Girault und Valk, 2003, Seite 14)

Die vereinigte Menge $P \cup T$ aller Plätze und Transitionen in einem Netz wird auch als Menge der *Netzelemente* bezeichnet. In einem Netz $N = (P, T, F)$ werden für ein gegebenes Netzelement $e \in P \cup T$ alle anderen Netzelemente $x \in (P \cup T) \backslash \{e\}$, als *Vorbereich von* e bezeichnet, falls $(x, e) \in F$ und als *Nachbereich von* e, falls $(e, x) \in F$. Der Vorbereich von e wird mit »$\bullet e$« und der Nachbereich mit »$e\bullet$« bezeichnet. Es ist zu beachten, dass sowohl Vor- als auch Nachbereich stets gänzlich vom jeweils anderen Typ (Platz oder Transition) sind wie ihr Bezugselement.

In einem gegebenen Netz kann ein Teil der Plätze markiert sein. In Plätzen können einzelne *Marken* existieren, wobei die Gesamtmenge aller Marken als *Markierung* bezeichnet wird. Eine Marke wird in der graphischen Repräsentation in einem Platz zumeist mit einem schwarzen Punkt dargestellt. Eine Marke in einem Platz ist notwendige (aber nicht immer hinreichende) Voraussetzung für das Feuern einer Transition im Nachbereich des Platzes. Eine Transition kann nur dann feuern, wenn alle Plätze in ihrem Vorbereich eine Marke beinhalten. Die Transition wird in diesem Fall als *aktiviert* bezeichnet.

2.2.1. P/T-Netze

Während klassische Petrinetze gut dafür geeignet sind nebenläufige Prozesse abzubilden, ist ihre Ausdrucksmächtigkeit doch eingeschränkt. Insbesondere bei großen Systemen ist eine sehr große Zahl an Stellen für eine korrekte Abbildung notwendig. Aus diesem Grund existieren neben dem klassischen Petrinetzformalismus noch eine Vielzahl weiterer Formalismen. Eine sehr einfache Erweiterung bilden die sogenannten P/T Netze, welche lediglich eine Anfangsmarkierung, sowie eine Gewichtungsfunktion der Flussrelation hinzufügen. Darüber hinaus erlauben P/T Netze die Anwesenheit mehrerer Marken auf einem Platz. In der graphischen Darstellung können an Kanten ganze Zahlen aufgeführt sein, welche die für einen Feuervorgang benötigte Markenanzahl von dem entsprechenden Platz bzw. die Anzahl an produzierten Marken angeben.

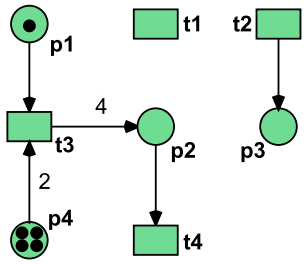

Abbildung 2.2.: Ein Beispiel für ein P/T-Netz

Analog zur Definition von Petrinetzen sollen P/T Netze im Wesentlichen nach (Girault und Valk, 2003) wie folgt definiert werden:

Definition 2.3 (P/T-Netz)**.** Ein P/T-Netz $N = (P, T, F, W, \mathbf{m_0})$ ist ein 5-Tupel. Dabei ist:

- (P, T, F) ein Netz.
- $W : F \to \mathbb{N}_+$ eine Gewichtungsfunktion.
- $\mathbf{m_0} : P \to \mathbb{N}_0$ eine Initialmarkierung.

Beispiel 2.4. Ein Beispiel für ein P/T-Netz findet sich in Abbildung 2.2. Das Netz besteht aus drei zusammenhängenden Komponenten. Die Transition $t1$ besitzt weder Vor- noch Nachbereich, die Transition $t2$ hingegen einen Platz in ihrem Nachbereich. $t3$ benötigt eine Marke aus dem Platz $p1$, sowie zwei Marken aus dem Platz $p4$ und erzeugt beim Feuern vier Marken, welche in den Platz $p2$ gelegt werden. Da sowohl $t1$ als auch $t2$ keinen Vorbereich haben, können beide beliebig oft feuern. Obwohl in $p4$ insgesamt vier Marken zur Verfügung stehen, steht in $p1$ nur eine Marke zur Verfügung, sodass $t3$ nur ein einziges Mal schalten kann. Folglich kann $t4$ insgesamt vier mal schalten, jedoch erst nachdem $t3$ geschaltet hat.

2.2.2. Erreichbarkeitsgraphen

Für die Beantwortung einzelner Fragestellungen im Kontext eines Netzes ist es hilfreich einen systematischen Verlauf aller möglichen Schaltungen durch das Netz abbilden zu können. Dies wird durch die Konstruktion eines *Erreichbarkeitsgraphen* umgesetzt. In einfachen (üblichen) Erreichbarkeitsgraphen kann das Schaltverhalten eines Netz nach Interleaving-Semantik (sequentielles Schalten) abgelesen werden. Genauere Beschreibungen zur Interleaving- und anderen Petrinetzsemantiken folgen in Abschnitt 2.2.5.

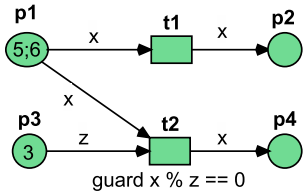

Abbildung 2.3.: Ein Beispiel für ein gefärbtes Netz

Dabei wird ausgehend von der Initialmarkierung sequenziell jede mögliche Schaltung einer aktivierten Transition betrachtet. Die jeweiligen Folgemarkierungen werden in den Graphen aufgenommen, dessen erster Knoten der Initialmarkierung entspricht. Gleiche Markierungen sind unabhängig von ihrer Entstehung auch *dieselben* Knoten im Erreichbarkeitsgraphen. Auf diese Weise werden systematisch alle erreichbaren Zustände des Systems gefunden. Erreichbarkeitsgraphen sind nur endlich, sofern das zugehörige Netz *beschränkt* ist. Ohne eine genaue Definition anzugeben, bedeutet *Beschränktheit*, dass kein Platz im Netz durch wiederholte Schaltvorgänge beliebig viele Marken aufnehmen kann. Ferner sind Erreichbarkeitsgraphen im schlimmsten Fall hyperexponentiell groß bzw. nicht primitiv-rekursiv in Hinblick auf die Größe des gesamten Netzes.

2.2.3. Gefärbte Netze

Eine Erweiterung der P/T-Netze ergibt sich, indem für bestimmte Bedeutungen verschiedene Marken verwendet werden. Die Bezeichnung »Farbe« ist für die Typen der Marken gebräuchlich, daher die Bezeichnung *gefärbtes Netz*.

Es liegt nahe, Transitionen die Möglichkeit einzuräumen zwischen Marken zu unterscheiden. Zu diesem Zwecke können mit sogenannten Transitions-*Guards* Bedingungen (Prädikate) hinzugefügt werden, welche der Marken zum Schalten verwendet werden dürfen und welcher Typ Marken produziert wird. An den Kanten werden in diesen Fällen Variablen eingesetzt, welche durch die Guard-Prädikate referenziert werden. Im Gegensatz zu den einfachen P/T-Netzen sind gefärbte Netze bereits Turing-mächtig (sofern sie beliebige Farbmengen aufweisen).

Beispiel 2.5. In Abbildung 2.3 findet sich ein Beispiel für ein gefärbtes Netz. Im Platz $p1$ befinden sich je eine Marke des Markentyps »5« und »6«, sowie eine des Typs »3« in $p3$. Die Transition $t1$ kann beliebig schalten, während die Transition $t2$ durch ein Guard-Prädikat eingeschränkt ist. Die Transition $t2$ kann nur mit solchen Marken schalten, bei denen der Typ (als Ganzzahl betrachtet) der Marke aus $p3$ ein Teiler der Marke aus $p1$ ist. Folglich können hier zwei finale

Markierungen erreicht werden: Entweder liegen zwei Marken des Typs der Marken aus $p1$ in $p2$, wenn $t1$ zwei mal geschaltet hat, oder aber eine Marke »5« liegt in $p2$ und eine Marke »6« in $p4$. Dies ist der Fall, wenn $t1$ einmal mit $x = 5$ und $t2$ ebenfalls einmal geschaltet hat. Es gibt jedoch keine Möglichkeit eine Marke vom Typ »5« in den Platz $p4$ abzulegen, da dies durch das Guard-Prädikat verhindert wird.

2.2.4. Netze-in-Netzen

Eine weitere Möglichkeit Netze zu erweitern besteht darin, das Konzept von Netzen-in-Netzen (Valk, 1998) zuzulassen. Dabei kann eine Marke ein ganzes eigenständiges (Sub-) Netz darstellen, welches durch das übergeordnete Netz bewegt werden kann. Maßgebliches Leitbild dabei war die Idee mobiler Agenten, welche sich durch eine Plattform bewegen und an unterschiedlichen Orten unterschiedliche Operationen ausführen können. Die Agenten ändern intern dabei jedoch nicht ihre Verhaltensstruktur, sondern lediglich ggf. ihren gegenwärtigen Zustand. Netze-in-Netzen bzw. Objektnetze verfolgen zusätzlich Ansätze der objektorientierten Programmierung, da Marken einen Verweis auf ein anderes Netz darstellen, genau wie in der objektorientierten Programmierung Objekte Referenzen auf andere Objekte halten können.

Eine Erweiterung auf Basis der Netze-in-Netzen findet sich in Form des Referenznetzformalismus. Da dieser eine zentrale Vorarbeit dieser Arbeit ist, wurde ihm der eigenständige Abschnitt 2.3 gewidmet.

2.2.5. Petrinetzsemantiken

Bei der Simulation bzw. Ausführung von Petrinetz-Modellen existieren verschiedene Semantiken. Als einfachste Semantik kann die *Schaltsequenz*-Semantik genannt werden, bei der zu einem Zeitpunkt nur eine Transition feuert und alle produzierten und konsumierten Marken zum Zeitpunkt des nächsten Transitionsfeuerns eindeutig feststehen. Durch diese Semantik kann selbstverständlich keine echte Nebenläufigkeit erzeugt werden, da alle Transitionen in einer zeitlichen Abfolge gefeuert werden. Die Garantien an das Vorhandensein von Marken und den Gesamtzustand sind hierbei jedoch am höchsten. Die Schaltsequenz-Semantik wird auch als *Interleaving*-Semantik bezeichnet.

Werden innerhalb eines Zeitpunktes nicht nur eine einzige Transition, sondern gleich mehrere Transitionen auf einmal geschaltet, so wird das Netz mittels *Schritt*-Semantik bzw. *Step*-Semantik simuliert. Diese Semantik ermöglicht eine gewisse Form von Nebenläufigkeit, basiert aber durch die Existenz von Takten (Schritten) auf *Parallelität*. Dies bedeutet, dass das Netz immer nach einem Schritt und

vor dem nächsten Schritt in einen konsistenten Zustand überführt wird. Die Ausführung unterschiedlich komplexer Operation ist jedoch nur schwer abbildbar, da Zeitschritte diskret betrachtet werden. Im Zweifel führt die Parallelität zu längeren Wartezeiten bei kürzeren Operationen.

Die letzte betrachtete Semantik ist die sogenannte *True-Concurrency-Semantik* (dt. »echte Nebenläufigkeit«). Sie wird auch als *Partial-Order-Semantik* (dt. »Semantik partieller Ordnung«) oder auch als *TC-Semantik* bezeichnet. Hierbei sind Markierungen von Plätzen und Transitionsschaltungen lediglich durch eine partielle Ordnung in Relation zueinander gesetzt. Dies hat zur Folge, dass manche Schaltungen in ihrer zeitlichen Reihenfolge unvergleichbar sind. Diese Semantik hat daher die stärkste Anlehnung an ein natürliches System aus interagierenden Agenten, wie sie auch in der realen Welt anzutreffen sind. Als Nachteil kann aufgeführt werden, dass Netzsimulationen nach der True-Concurrency-Semantik nicht zwangsläufig zu einem bestimmten Zeitpunkt einen Zustand aufweisen müssen, bei dem keine Transition aktiv feuert. Diese Eigenschaft wird später noch einmal adressiert und trägt erheblich zur Komplexität der Überlegungen bei. Wird jedoch der Gesamtablauf einer Schaltung als Kausalnetz (ein Netz ohne Nichtdeterminismus) dargestellt, lassen sich durch Schnitte durch den Graphen, welche ausschließlich durch Plätze verlaufen (sogenannte P-Schnitte), mögliche Zustände des Systems beschreiben. Diese Überlegung wird im Kontext von True-Concurrency Checkpoints in Abschnitt 11.6.6 erneut aufgegriffen werden.

Für den verbleibenden Teil der Arbeit wird bei der Simulation von Petrinetzen jeder Form stets der Einsatz von *True-Concurrency-Semantik* angenommen, sofern dies nicht explizit abweichend angegeben ist. Ausnahmen sind mit der Referenz auf die entsprechend verwendete Semantik kenntlich gemacht.

2.2.6. Weitere relevante Konzepte im Kontext von Petrinetzen

Abschließend werden in diesem Abschnitt noch einige weitere Konzepte vorgestellt, welche an einzelnen Stellen innerhalb der Arbeit referenziert werden. Diese haben dabei entweder keine herausragende Stellung oder werden an der betreffenden Stelle noch einmal detaillierter eingeführt.

Invarianten

Invarianten beschreiben bestimmte strukturelle Eigenschaften von Netzen. So kann beispielsweise die (gewichtete) Menge aller Marken in einer bestimmten Teilmenge aller Plätze unabhängig von erfolgten Feuervorgängen konstant sein oder eine bestimmte Folge von Transitionsfeuervorgängen die Markierung eines Netzes wieder in den gleichen Zustand versetzen wie vor der Folge von Feuervorgängen.

Im Allgemeinen wird zwischen Platz- und Transitionsinvarianten unterschieden. Da beide Formen im Kontext der Arbeit nur am Rande erwähnt werden, soll an dieser Stelle die intuitive Einführung genügen und auf eine volle formale Definition verzichtet werden.

Komplementäre Plätze

Komplementäre Plätze beschreiben ebenfalls eine strukturelle Eigenschaft von Netzelementen. Für bestimmte Anwendungsfälle kann es hilfreich sein, ein statisches Limit für die Marken in einem Platz einzuführen. Ein Anwendungsbeispiel könnte die maximale Anzahl an verfügbaren Arbeitsplätzen umfassen. Strukturell kann dies so modelliert werden, dass neben dem Platz, welcher die Arbeitsplätze beschreibt, ein weiterer Platz mit der maximal zulässigen Anzahl an Marken existiert. Dieser Platz besitzt die umgekehrte Flussrelation des Platzes, welche die Arbeitsplätze beschreibt. Legt nun eine Transition eine Marke in den Platz, zieht sie gleichzeitig eine Marke vom komplementären Platz ab. Transitionen, welche Marken vom Platz entfernen, legen gleichzeitig Marken im komplementären Platz ab. Komplementäre Plätze sind ein effektives Mittel, um Plätze zu beschränken, ohne dass die Ausdrucksmächtigkeit von Netzen eingeschränkt wird.

Verklemmung

Die Verklemmung ist eine dynamische Eigenschaft der Simulation eines Petrinetzes. Sie beschreibt den Zustand, in dem keine Transition des Netzes aktiviert ist und somit auch keine Transition feuern kann. Die Bedeutung für das simulierte System ist dabei nicht vorweggenommen. Ein verklemmtes Netz kann ebenso einen Fehlerzustand darstellen wie auch eine erfolgreich abgeschlossene Bearbeitung seiner Aufgaben. Durch eine Verklemmung terminiert die Simulation eines Netzes.

Workflownetze

Workflownetze sind eine spezielle Kategorie von Petrinetzen. Sie können eingesetzt werden, um Geschäftsprozesse zu modellieren. Sie werden im Kontext der Agentenkommunikation erneut aufgegriffen werden. Workflownetze definieren sich nach (Aalst, 1997) (übersetzt) wie folgt:

Definition 2.6 (Workflow Netz)**.** Ein Netz $N = (P, T, F)$ ist ein Workflow Netz, genau dann wenn:

- N zwei spezielle Plätze i und o besitzt. i ist eine Quelle: $\bullet i = \emptyset$ und o eine Senke: $o\bullet = \emptyset$.
- Falls eine Transition t^* zu N hinzugefügt wird, welche die Plätze o und i verbindet ($\bullet t^* = \{o\}$ und $t^*\bullet = \{i\}$), dann ist das resultierende Netz streng zusammenhängend.

— (Aalst, 1997, Seite 413)

Darüber hinaus kann ein Workflownetz als *korrekt* bezeichnet werden, falls:

Definition 2.7 (Korrektes Workflow Netz)**.** Eine Prozedur, welche durch ein Workflow Netz $N = (P, T, F)$ modelliert wird, ist genau dann korrekt, wenn:

- Für jeden Zustand M, welcher von Zustand i erreicht werden kann, eine Feuersequenz existiert, welche von M zu o führt.
  Formal: $\forall M(i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o)$
- Zustand o ist der einzig erreichbare Zustand aus Zustand i mit wenigstens einer Marke im Platz o.
  Formal: $\forall M(i \xrightarrow{*} M \wedge M \geq o) \Rightarrow (M = o)$
- Es existieren keine nicht lebendigen Transitionen in (N, i).
  Formal: $\forall_{t \in T} \exists_{M, M'} i \xrightarrow{*} M \xrightarrow{t} M'$

— (Aalst, 1997, Seite 413-414)

Ein weiteres Konzept von Interesse im Kontext der Workflownetze ist die *Task-Transition* (Aalst, 1997; Jacob, 2002), welche den Fehlschlag des Feuervorgangs unterstützt und im Falle des Fehlschlags alle Marken zurück in die jeweiligen Plätze des Vorbereichs legt. Die Task-Transition wird detaillierter in Abschnitt 5.1 vorgestellt.

2.3. Referenznetze

Ausgehend von der »Netze-in-Netzen« Theorie (Valk, 1998), »Objektorientierten Netzen« (Moldt, 1996) sowie »Synchronen Kanälen« (S. Christensen und Hansen, 1994; Maier und Moldt, 2001), welche im folgenden Abschnitt 2.3.2 noch erläutert werden, liegt die Grundidee bei Referenznetzen darin diese Konzepte zu vereinen. Die Theorie zu den Referenznetzen und der zugehörige Referenznetzformalismus geht im Wesentlichen auf die Arbeiten von Olaf Kummer

(Kummer, 1996, 1998, 1999, 2000, 2002) zurück, nach welchen sich auch die Ausführung an dieser Stelle richtet.

Referenznetze verbinden analog zu Netzen-in-Netzen die Konzepte der objektorientierten Programmierung von Klassen und Instanzen mit der Petrinetztheorie. Somit existieren im Referenznetzformalismus nicht nur Netze mit Plätzen und Transitionen, sondern auch *Netzinstanzen* mit *Platzinstanzen* und *Transitionsinstanzen*. Diese Eigenschaft ist fundamental für die Grundlage dieser Arbeit, da durch diese Eigenschaft von Referenznetzen zur Simulationszeit weitere Instanzen von Netzen generiert werden können. Die physikalische Größe der Simulation (verbrauchter Speicher, benötigte Prozessorressourcen, etc.) sind also nicht zwangsläufig im Vorhinein bekannt. Die Netzstruktur ist dynamisch, sodass Referenznetze eine inhärent höhere Komplexität aufweisen als niedrigere Petrinetzformalismen.

2.3.1. Instanziierung

Wie eingehend beschrieben, ist eine der wichtigsten Eigenschaften des Referenznetzformalismus die Instanziierung von Netzen. Während in klassischen Petrinetz-Formalismen direkt in einem Netz simuliert wurde, betrachtet der Referenznetzformalismus ein Netz wie eine Klasse der objektorientierten Programmierung. Auf Basis dieses Netzes können zur Laufzeit beliebig viele Netzinstanzen erzeugt werden. Dies bedeutet insbesondere auch, dass durch Netzinstanzen sowohl Plätze als auch Transitionen eines Netzes insgesamt in beliebig häufiger Replikation existieren können. Somit liegt es nahe, neben Netzinstanzen auch von Platz- und Transitionsinstanzen zu sprechen. Um zusätzlich den Unterschied zwischen Netzen und Netzinstanzen zu betonen, werden Netze im Rahmen der Arbeit auch als *statische Netze* oder *Netztemplates* bezeichnet. Insbesondere wird in dieser Arbeit durchweg im Kontext von Referenznetzen durch »Netz« stets das Netztemplate bzw. statische Netz und durch »Netzinstanz« die Instanz eines Netzes bezeichnet.

Damit das Konzept der Instanziierung von Netzen einen gewinnbringenden Nutzen aufweist, ist es notwendig innerhalb der Netzinstanzen mit anderen Netzinstanzen umgehen zu können. An dieser Stelle hilft das Konzept von Netzen-in-Netzen (Valk, 1998), bei dem ungerichtete synchrone Kanäle zum Einsatz kommen, sodass das Schalten von Transitionen synchronisiert wird und bei dem einzelne Netze als Marken innerhalb von Plätzen in anderen Netzen existieren können. Im Formalismus der Referenznetze ist dieser Ansatz ebenfalls umgesetzt, jedoch mit Hilfe von Referenzen auf die anderen Netzinstanzen. So begründet sich auch der Name Referenznetze.

Innerhalb von Netzinstanzen von Referenznetzen besteht somit die Möglichkeit neue Netzinstanzen von anderen bekannten statischen Netzen zu erzeugen. Da die

Erzeugung einer neuen Netzinstanz stets eine Aktion ist, liegt es nahe, dass sie mittels einer speziellen Anschrift an einer Transition umgesetzt ist. Feuert eine derartige Transition innerhalb einer Netzinstanz, so entsteht eine Marke mit einer Referenz auf eine neu erzeugte Netzinstanz.

2.3.2. Synchrone Kanäle

Das bloße Erzeugen von Netzinstanzen, welche unabhängig voneinander schalten können und lediglich Referenzen zu jeweils anderen Netzinstanzen halten, bietet auf den ersten Blick keine großen Vorteile. An dieser Stelle fehlt eine Methodik um zwischen den jeweiligen Netzinstanzen zu kommunizieren. Referenznetze realisieren dieses Bedürfnis nach Kommunikation mithilfe sogenannter *synchroner Kanäle*.

Der Austausch von Nachrichten ist bei jeder Form der Programmierung und der Modellierung von Abläufen, Prozessen und Protokollen einer der zentralen Bestandteile. Während in der klassischen Programmierung meist ein Aufrufer und ein Aufgerufener existieren, ist dieses Modell gerade für die Abbildung natürlicher Kommunikationsstrukturen nicht immer zutreffend. Die Annahme eines Kommunikationsinitiators, auf dessen Initiierung ein gleichberechtigter Informationsaustausch folgt ist im Allgemeinen realistischer. Synchrone Kanäle beabsichtigen diesen Zusammenhang in Form einer Gruppensynchronisation zu formalisieren.

Wie alle Inhalte dieser Arbeit zu Referenznetzen orientiert sich auch dieser Abschnitt im Wesentlichen an der Dissertation (Kummer, 2002). Synchrone Kanäle in diesem Kontext als Vorarbeit schafft die Veröffentlichung (S. Christensen und Hansen, 1994).

Um die Abbildung von Gruppensynchronisation auf Petrinetze umsetzen zu können, ist es zunächst notwendig zu betrachten, welche Aspekte bereits von herkömmlichen Petrinetzen, wie P/T-Netzen, umgesetzt werden. So bieten Petrinetze beispielsweise die Möglichkeit individuelles Verhalten der einzelnen Kommunikationspartner durch die inhärente Nebenläufigkeit abzubilden. Dabei spielt die Betrachtung der im Falle der Simulation anzuwendenden Petrinetzsemantik eine entscheidende Rolle. Die True-Concurrency-Semantik bildet wie im vorherigen Abschnitt beschrieben die natürliche Kommunikation am ehesten ab und soll daher betrachtet werden. Da durch die Nebenläufigkeit auch Transitionen zu sich selbst nebenläufig schalten können und das Unterbinden der Möglichkeit zum Schalten ausschließlich durch fehlende Marken erzeugt werden kann, findet sich klassisch keine Möglichkeit mehrere Transitionen exakt gleichzeitig zu schalten und dabei Informationen zwischen den Transitionen auszutauschen. Der Formalismus muss somit um eine Mechanik erweitert werden, welche dieses Verhalten unterstützt. Dies kann beispielsweise durch Transitionsanschriften formalisiert

werden, wie es bei Referenznetzen der Fall ist. Diese benötigen eine Information darüber welche Transitionen an der Synchronisation teilnehmen und welche Art von Informationen ausgetauscht werden kann. Während die erste Komponente durch eine gezielte Benennung des entsprechenden Kanals gewährleistet werden kann, müssen für den zweiten Aspekt Parameter eingesetzt werden.

Während die Einführung von synchronen Kanälen in den Petrinetzformalismus auf den ersten Blick konzeptuell nicht komplex erscheint, ergeben sich bei genauerer Betrachtung jedoch diverse Schwierigkeiten in der detaillierten Umsetzung. Petrinetze erlauben im normalen Fall umfangreiche Zusammenlegungen und Spaltungen bestimmter Kontrollflüsse, insbesondere durch den Einsatz gefärbter Netze. Der Fachbegriff hierzu lautet (Ent-)Faltung. Ein Netz mit synchronen Kanälen kann jedoch bei extrem komplexen Beziehungen zwischen den Synchronisationspartnern in eine Situation laufen, bei der ein endliches Netz mit synchronen Kanälen nicht mehr endlich in seiner entpackten Fassung ist. (S. Christensen und Hansen, 1994) beschreiben in diesem Kontext jedoch hinreichende Bedingungen, die ein Netz mit synchronen Kanälen erfüllen muss, damit es zu solchen Problemen nicht kommt.

2.3.3. Anschriften

Der Referenznetzformalismus zählt zu den höheren[2] Petrinetz-Formalismen. Es ist daher naheliegend, dass dafür eine komplexe Anschriftensprache notwendig ist. Die formale Definition von Referenznetzen im Originalwerk (Kummer, 2002) ist absichtlich allgemein gehalten und auf Basis von Graphersetzungssystemen spezifiziert. In der grundlegenden Definition kommen beschriftete Algebren zum Einsatz und auf eine Ausformulierung von etwaigen Beschriftungen wird im formalen Teil der Arbeit zunächst vollständig verzichtet. Jedoch ist es für den Einsatz in dieser Arbeit hilfreich den Referenznetzformalismus im Kontext der Implementation im Simulator Renew und der Programmiersprache Java zu betrachten und an der zugehörigen Stelle direkt Anschriften zu behandeln. Folglich wird dieser Abschnitt zur Anschriftensprache, auch wenn er formal zu der Definition von Referenznetzen zählt, in Abschnitt 2.8 zum Simulator Renew behandelt.

2.3.4. Beispiele zu Referenznetzen

Die anschauliche Betrachtung von beispielhaften Referenznetzen bedarf der Konkretisierung der Anschriftensprache. Wie im Abschnitt 2.3.3 erläutert, umfasst die Konkretisierung der abstrakten Referenznetze im Rahmen dieser Arbeit die

[2] Zum Begriff der »höheren« Petrinetz-Formalismen siehe beispielsweise (Jensen und Rozenberg, 1991) und (Jensen, 2001)

Anwendung auf die Programmiersprache Java. Aus diesem Grund wird auf die Darstellung eines Beispiels von Referenznetzen nur auf Basis des Formalismus verzichtet und dieses nur im Kontext der Realisierung mit Java betrachtet. Beispiele finden sich somit im Unterabschnitt 2.8.3 zu Renew, nachdem die Details der Implementation in Renew behandelt wurden.

2.4. Softwareagenten

Softwareagenten sind seit langer Zeit ein aktives Forschungs- und Anwendungsfeld. Die zentrale Idee umfasst die Kapselung einzelner Bereiche der Software in Komponenten, welche eine gewisse Autonomie aufweisen. Diese Komponenten werden als Agenten bezeichnet. Agenten besitzen eine Wissensbasis und kommunizieren über Nachrichten. Die Antwort auf eine Nachricht ist dabei nicht explizit ausgestaltet oder für den Agenten verpflichtend, wodurch sich Agenten elementar von beispielsweise Subroutinen oder Objekten unterscheiden. Agenten versuchen das Verhalten von Menschen und anderen autonomen Realweltentitäten untereinander genauer abzubilden als es durch andere Softwareentwicklungsmuster der Fall ist.

Dieser Abschnitt gibt eine Definition eines Softwareagenten für den Rahmen der Arbeit an, führt die grundlegenden Überlegungen der Organisation »Foundation for Intelligent Physical Agents« (FIPA) an und beschreibt den Paose Ansatz.

2.4.1. Definitionen eines Agenten

In der Literatur finden sich sehr verschiedene Definitionen des Agentenbegriffs. So definiert beispielsweise Shoham:

> *[...] An entity that functions continuously and autonomously in an environment in which other processes take place and other agents exist.*
> *(dt.: Eine Entität, welche fortlaufend und autonom in einer Umgebung funktioniert, in der andere Prozesse ablaufen und andere Agenten existieren.)*
>
> — (Shoham, 1993, Seite 52)

Ähnlich definiert die FIPA, deren Vorgaben detaillierter in Abschnitt 2.4.2 aufgegriffen werden:

> *An agent is a computational process that implements the autonomous, communicating functionality of an application.*
> *(dt.: Ein Agent ist ein Berechnungsprozess, welcher die autonome und kommunizierende Funktionalität einer Anwendung implementiert.)*
> — (Foundation for Intelligent Physical Agents (FIPA), 2002, Zeile 1019, Seite 29)

Eine sehr bekannte Definition erfolgt durch Wooldridge, welche auch als Basis in (Wagner, 2018) eingesetzt wird:

> *An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its delegated objectives.*
> *(dt.: Ein Agent ist ein Computersystem, welches sich ein einer Umgebung befindet und welches in der Lage ist, autonome Aktionen in dieser Umgebung auszuführen, um seine übertragenen Aufgaben zu erfüllen.)*
> — (Wooldridge, 2009, Seite 21)

Auf der Basis dieser Definitionen soll nun eine entsprechende eigene Definition für diese Arbeit geschaffen werden, welche zusätzlich den für diese Arbeit relevanten Begriff der Plattform umfasst:

Definition 2.8 (Agent, Plattform)**.** Ein Agent ist ein autonomes Computersystem, welches sich in einer Umgebung befindet und in der Lage ist auf bestimmen Plattformen mit anderen Agenten in Kommunikation sowie mit den Umgebungen bzw. Plattformen in Interaktion zu treten. Agenten nutzen diese Eigenschaften, um ihre designierten Aufgaben zu verfolgen und zu erfüllen. Eine Plattform kann Agenten beherbergen und mit ihnen und den Umgebungen in Interaktion treten.

Die internen Abläufe eines Agenten umfassen verschiedene Prozesse und Operationen:

Definition 2.9 (Protokoll)**.** Die internen Abläufe eines Agenten werden als *Protokolle* bezeichnet.

Protokolle können von Agenten gestartet und beendet werden. Protokolle sind ebenfalls in der Lage auf die *Wissensbasis* des Agenten zuzugreifen. Dabei liegt ein Protokoll stets in einem Agenten, sodass ein anderer Agent keinen Zugriff auf das Protokoll eines gleichberechtigten anderen Agenten hat. Das Protokoll kann jedoch den Agenten veranlassen eine Nachricht zu verschicken, um so mit anderen Agenten zu kommunizieren. Protokolle sind daher in der Arbeit anders definiert

als beispielsweise in der Netzwerkkommunikation. Als Abgrenzung werden derartige Netzwerk-Protokolle verschiedener Ebenen wie beispielsweise TCP, HTTP, FTP, usw. im Rahmen dieser Arbeit als *Kommunikationsprotokolle* bezeichnet.

Bei wiederverwendbaren Protokollfragmenten können *Subprotokolle* zum Einsatz kommen:

Definition 2.10 (Subprotokoll)**.** Ein *Subprotokoll* ist ein eigenständiger Ablauf innerhalb eines Agenten, welcher nur durch ein übergeordnetes Protokoll gestartet werden kann, jedoch nicht direkt durch den Agenten.

Wegen der Nachrichtenbasiertheit der Kommunikation können eingehende Nachrichten den Start eines Protokolls verursachen. Diese Form des Protokollstarts wird als *reaktiv* bezeichnet. Die gegenteilige Form tritt auf, falls der Agent das Protokoll ohne äußeres Einwirken startet, beispielsweise durch ein anderes Protokoll. In diesem Fall wird von einem *proaktiven* Start des Protokolls gesprochen.

In der Kombination von Umgebung und der Gesamtheit der Agenten ergibt sich ein *Multiagentensystem*.

Definition 2.11 (Multiagentensystem)**.** Die Gesamtheit aus Agenten, Umgebungen und etwaigen Plattformen wird als *Multiagentensystem* bezeichnet.

Während Multiagentensysteme in ihrem konzeptuellen Aufbau als verteiltes System zu verstehen sind, muss eine Simulation eines Multiagentensystem nicht zwangsläufig verteilt sein, ähnlich wie auch andere verteilte Systeme auch ohne Verteilungsaspekt simuliert werden können.

2.4.2. FIPA

Die »Foundation for Intelligent Physical Agents« (FIPA)[3] ist eine Standardisierungsorganisation des Institue of Electrical and Electronics Engineers (IEEE). Sie wurde 1996 in der Schweiz gegründet und befasst sich mit der Standardisierung von Agenten- und Multiagentensystemen. Im Laufe ihrer Arbeit hat die FIPA diverse Standards zu dem Thema hervorgebracht, welche weitreichende Adaption im Bereich der agentenorientierten Softwareentwicklung erfahren haben.

Die wesentliche Architektur nach der FIPA findet sich in Abbildung 2.4. Die einzelnen Bestandteile umfassen nach (Foundation for Intelligent Physical Agents (FIPA), 2004):

Agent Management System (AMS) Bezeichnet einen Verwaltungsdienst, welcher wie ein herkömmlicher Agent ansprechbar sein muss.

[3] http://www.fipa.org/ - Zuletzt abgerufen am 15.12.2021

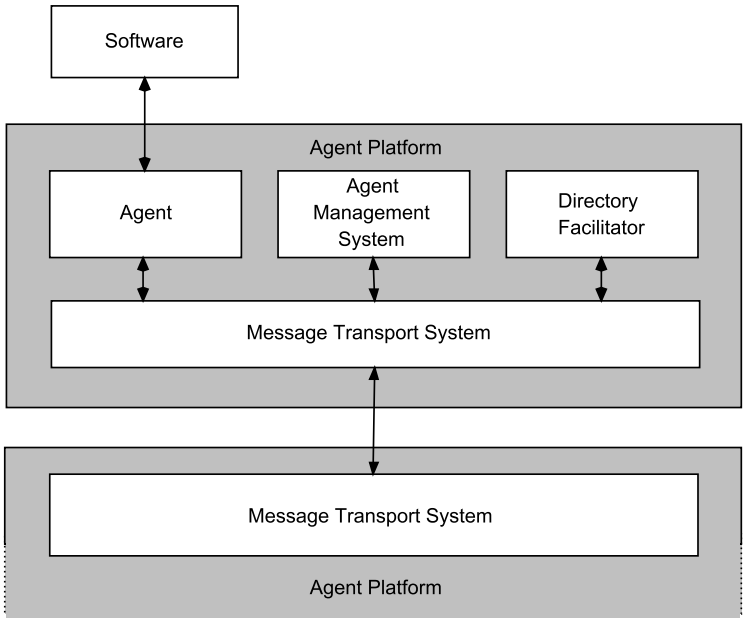

Abbildung 2.4.: Das »Agent Management Reference Model«. Entnommen aus (Foundation for Intelligent Physical Agents (FIPA), 2004, Seite 5).

Directory Facilitator (DF) Bezeichnet einen Verzeichnisdienst.

Message Transport Service (MTS) Bezeichnet einen Transportservice für Nachrichten, dessen interne Schnittstelle beliebig aufgebaut sein kann.

Agent Communication Channel (ACC) Bezeichnet die externe Schnittstelle des MTS, die einheitlich ist.

Agent Message Transport Protocol (MTP) Bezeichnet die Protokolle, die von der FIPA als Protokolle zum Nachrichtenaustausch zwischen Agenten vorgesehen sind.

Agent Communication Language (ACL) Bezeichnet die Sprache, die innerhalb des MTP eingesetzt wird. Sie folgt der Sprechakttheorie von (Searle, 1969). Die Grundbausteine der ACL sind Sprechakte im Sinne der Sprechakttheorie.

»Software« in der Abbildung beschreibt alle Anteile der lokalen Anwendung, welche nicht agentenbasiert sind.

Des Weiteren definiert die FIPA Abläufe für den Agenten-Lebenszyklus in (Foundation for Intelligent Physical Agents (FIPA), 2004). Ein Agent kann initialisiert bzw. erzeugt werden und ist sodann aktiv. Anstatt aktiv zu sein, kann

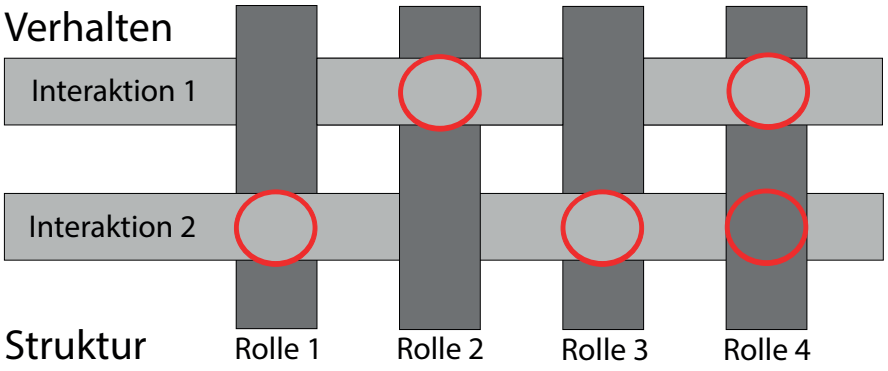

Abbildung 2.5.: Die Paose Matrix. Entnommen und übersetzt aus (Cabac, 2010, Seite 123).

er warten, pausiert sein oder umziehen. Sind die jeweiligen Vorgänge abgeschlossen kehrt er in den aktiven Zustand zurück. Wird der Agent beendet, kehrt er in einen »unbekannten« Zustand zurück, bis er erneut erzeugt bzw. initialisiert wird.

2.4.3. PAOSE

Der Ansatz des Petri net-based, Agent- and Organization-Oriented Software Engineering(Paose) (Moldt, 2005) (Cabac, Dörges, Duvigneau, Moldt u. a., 2008) (Cabac, 2010) erweitert das Design und die Entwicklung eines Multiagentensystems zu einem Leitbild des gesamten Entwicklungsprozesses (Cabac, 2007). Der Ansatz wurde seit den 1990er-Jahren entwickelt und trug im Laufe der Zeit verschiedene Namen. Anfang der 2000er-Jahre etablierte sich das Akronym Paose (Moldt, 2005). Der Prozess ist in der Beschreibung der Quellen eng mit der Capa-Plattform im Kontext der Mulan-Architektur verknüpft, welche erst in Abschnitt 3.3.1 vorgestellt werden wird. Paose ist auf konzeptueller Ebene jedoch allgemein gehalten und der Ansatz ist ohne den konkreten Zusammenhang zu Capa nutzbar. Einen guten Einstieg in die Thematik liefert auch (Wagner, 2018, Seiten 26-30).

Der Grundgedanke von Paose umfasst die Interpretation eines Entwicklerteams als Multiagentensystem, bei dem einzelne Entwickler als Agenten verstanden werden. Paose setzt damit auf der gleichen Ausgangsüberlegung wie das Gesetz von Conway (Conway, 1968) auf. Dieses besagt, dass die Struktur einer Software der Struktur der Organisation entspricht, welche sie entwickelt. Der Einsatz des umgekehrten Gesetzes von Conway beinhaltet die Organisation derart zu strukturieren, dass sie die gewünschte Struktur der Software abbildet.

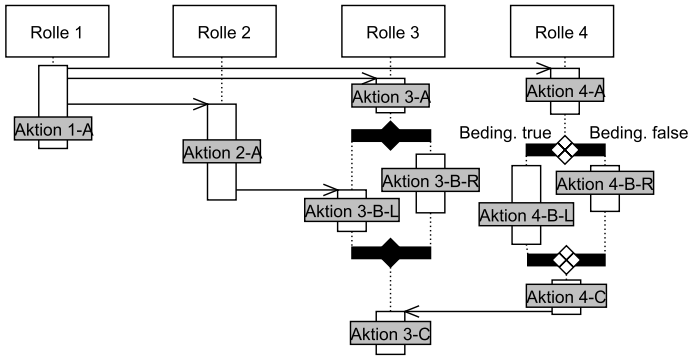

Abbildung 2.6.: Beispielhaftes Agenteninteraktionsprotokoll (AIP).

Das zentrale Konstrukt des Paose Ansatzes und die Sicht auf Multiagentensysteme ist die Paose Matrix. Paose umfasst drei Betrachtungsperspektiven bzw. Dimensionen, welche ein Multiagentensystem hinreichend beschreiben: Die Struktur, das Verhalten und die Ontologie.

Die Struktur umfasst die verschiedenen *Rollen*, welche von Agenten eingenommen werden können. Das Verhalten beschreibt die möglichen *Interaktionen* der einzelnen Rollen. Nicht alle Rollen interagieren mit allen anderen Rollen, jedoch ist bei einer möglichen Interaktion ein gemeinsames Verständnis von Konzepten und Terminologien nötig. Dieses Verständnis wird durch die dritte Dimension, der *Ontologie*, ausgedrückt. Abbildung 2.5 aus (Cabac, 2010) zeigt diesen Zusammenhang beispielhaft und für Verhalten und Struktur bildlich. Ontologien müssen für die mit roten Kreisen markierten Überschneidungen definiert sein, da dort eine Interaktionskomponente mit einer Rollenkomponente in Beziehung steht.

Entwicklungsprozess

Der Entwicklungsprozess nach Paose umfasst mehrere Stufen. Zunächst werden im Sinne des *Requirements Engineerings* grundlegende Rollen identifiziert und in einem »Coarse Design Diagram (CDD)« festgehalten. Auf dieser Basis werden sodann Interaktionen mit der Hilfe sogenannter »Agenteninteraktionsprotokolle (AIPs)« beschrieben. Weitere Ableitungen sind Rollen und Abhängigkeiten und die benötigten Ontologien. Im vollen Prozess werden aus den identifizierten AIPs Protokolle in Form von Petrinetzen (sogenannte »Protokollnetze«) für die Agenten konstruiert, aus Rollen und Abhängigkeiten werden Entscheidungskomponenten generiert und aus den Ontologien Ontologieklassen und Wissensbasen.

Viele Aspekte des Paose-Ansatzes und -Prozesses sind auch für diese Arbeit interessant und sollen Anwendung finden. Insbesondere die Beschreibung von In-

teraktionen durch AIPs wird genutzt werden sowie das (implizite) Verständnis von Rollen im System. Der vollständige Weg bis hin zur Capa Plattform wird jedoch nicht umgesetzt. Die Begründung liegt darin, dass die Capa Plattform in ihrer Form die Notwendigkeiten im Rahmen der Forschungsfragen der Arbeit nicht in hinreichendem Umfang liefert. Weitere Details und Begründungen hierzu werden in Abschnitt 4.3.3 aufgegriffen werden.

Abbildung 2.6 zeigt ein beispielhaftes AIP, wie es auch im verbleibenden Teil der Arbeit auftreten wird. Einige Details sind dabei vereinfacht dargestellt wie beispielsweise die explizite Unterscheidung von proaktiven und reaktiven Anteilen. Beschrieben sind im Beispiel vier Rollen und deren Interaktionen untereinander. Rolle 1 startet den Prozess proaktiv. Rolle 2, Rolle 3 und Rolle 4 starten durch den eingehenden Pfeil die Aktionen 2-A, 3-A und 4-A jeweils reaktiv. In Rolle 4 ist ein sogenannter »XOR-Split« zu beobachten, bei dem je nach einer Bedingung nur eine der Aktionen ausgeführt wird aber niemals beide. Rolle 3 weist einen entsprechenden »AND-Split« auf, welcher stets beide Aktionen ausführt. Sind beide Aktionen ausgeführt, wartet Rolle 3 auf eine Nachricht von Rolle 4 (Aktion 3-C), welche abgeschickt wird, sobald die entsprechend der Bedingung ausgewählte Aktion von Rolle 4 und nachfolgend Aktion 4-C ausgeführt wurde.

2.5. Architekturen und Systeme

Die Architektur einer Software ist ein entscheidender Aspekt der Softwareentwicklung. Da im Rahmen der Arbeit eine Architektur beschrieben bzw. entworfen werden soll, liegt es nahe, zunächst den Begriff zu definieren. Zu diesem Zweck soll das Standardwerk von Bass et al. in seiner aktuellen, vierten Auflage herangezogen werden:

> **Definition 2.12** (Softwarearchitektur)**.** The software architecture of a system is the set of structures needed to reason about the system. These structures comprise software elements, relations among them, and properties of both.
> (dt.: Die Softwarearchitektur eines Systems ist die Menge an Strukturen, welche benötigt wird, um über ein System nachzudenken. Diese Strukturen umfassen Softwareelemente, die Beziehungen zwischen diesen und die jeweils zugehörigen Eigenschaften.)
>
> — (Bass u. a., 2021, Kapitel 1.1)

Das Ziel ist somit sowohl *Komponenten* als auch *Beziehungen* zu beschreiben, um ein zielgerichtetes Durchdenken der Software zu ermöglichen. Vor diesem Hin-

tergrund widmet sich dieser Abschnitt diesem Aspekt insbesondere im Kontext der verteilten Systeme. Weitere Diskussionen in diesem Kontext umfassen insbesondere den Aspekt der Skalierbarkeit und werden separat in Abschnitt 2.6 adressiert. Neben den Inhalten zu verteilten Systemen soll abschließend noch die Einheitentheorie beschrieben werden, welche als Grundkonstrukt für den Entwurf der Architektur im Rahmen der Arbeit Anwendung finden wird.

2.5.1. Kohäsion und Koppelung

Die Begriffe *Kohäsion* und *Koppelung* beschreiben die Menge an Beziehungen innerhalb von Modulen einer Software sowie zwischen solchen. Gerade im Kontext der zuvor genannten Definition einer Softwarearchitektur fällt den beiden Begriffen Bedeutung zu. Die Konstruktion von Modulen selbst geht auf das *Geheimnisprinzip* (engl.: *information hiding*) (Parnas, 1971) zurück, welches besagt, dass einzelne Entwurfsentscheidungen innerhalb einzelner Module gekapselt sein sollten. Module sollten die gekapselten Entwurfsentscheidungen anderer Module nicht kennen und dürfen nur so viele Verbindungen zueinander aufweisen wie unbedingt nötig.

Brügge und Dutoit definieren:

> **Definition 2.13** (Koppelung)**.** Coupling is the number of dependencies between two subsystems.
> (dt.: Koppelung ist die Anzahl an Abhängigkeiten zwischen zwei Subsystemen.)
>
> — (Brügge und Dutoit, 2004, Seite 230)

> **Definition 2.14** (Kohäsion)**.** Cohesion is the number of dependencies within a subsystem.
> (dt.: Kohäsion ist die Anzahl an Abhängigkeiten innerhalb eines Subsystems.)
>
> — (Brügge und Dutoit, 2004, Seite 231)

Nach den Ausführungen von (Brügge und Dutoit, 2004) ist es erstrebenswert Systeme mit niedriger Koppelung und hoher Kohäsion zu konstruieren. Niedrige Koppelung erhöht die Flexibilität und Autonomie der einzelnen Komponenten, während eine hohe Kohäsion die Zielgerichtetheit der einzelnen Komponenten forciert. Zusammenhangslose Bestandteile einzelner Komponenten führen zu einer niedrigen Kohäsion. Die Gestaltung entlang dieser Richtlinie bezüglich Koppelung und Kohäsion ist insbesondere in verteilten Systemen wichtig.

2.5.2. Verteilte Systeme

Verteilte Systeme werden seit vielen Jahrzehnten untersucht. Die Vorläufer verteilter Systeme finden sich nach Andrews (Andrews, 1999) in Algorithmen zum Nachrichtenversand in Betriebssystemen in den 1960er Jahren. Bereits 1962 veröffentlichte Carl Adam Petri seine Dissertation zum Thema »Kommunikation mit Automaten« (Petri, 1962) und legte die Grundlage der gesamten heutigen Petrinetzforschung.

In den 1970er Jahren schritt die Entwicklung stark voran, Ethernet wurde erfunden, Firmen wie Microsoft und Apple gegründet. Auf dem Gebiet der verteilten Systeme wurde zentrale Grundlagenforschung veröffentlicht, wie beispielsweise Dijkstras Arbeit zu selbst-stabilisierenden Systemen (Dijkstra, 1974) oder Lamports Arbeit zu verteilten Uhren (Lamport, 1978). Ende der 1970er Jahre wurde das OSI Schichtenmodell vorgestellt, welches Anfang der 1980er Jahre dann standardisiert wurde.

Mit dem Wandel der Jahre ging die Entwicklung über serviceorientierte Architekturen zur Jahrtausendwende zu modernen Cloud-Architekturen über. Diese aktuelleren Entwicklungen werden vermehrt im Abschnitt 2.6 zu Skalierbarkeit adressiert, da diese eine der zentralen Herausforderungen des Designs verteilter Systeme in den 2010er und frühen 2020er Jahren ist. Dieser Abschnitt adressiert vor allem einige grundlegende Überlegungen im Kontext verteilter Systeme.

Grundbegriffe

Zunächst erfolgt die Definition einiger Grundbegriffe, welche so immer wieder in der Arbeit auftreten werden.

Knoten/Instanz Im Kontext von Rechnernetzen wird sich der Begriff *Instanz* bzw. *Knoten* auf eine physikalische Maschine innerhalb eines Verbundes von physikalischen Maschinen beziehen.

(Netzwerk)koordinaten Um die Kommunikation mit einem Knoten aufnehmen zu können, müssen entsprechende Nachrichten an eine Adresse gerichtet werden. Diese Adresse wird durch den Begriff *Netzwerkkoordinate* oder schlicht *Koordinate* bezeichnet. Eine konkrete technische Ausprägung einer Koordinate ist eine IP-Adresse, ein konkretes Beispiel für eine Koordinate 192.168.1.1. Beim Begriff *Koordinate* schwingt mit, dass diese leicht änderbar und/oder temporär sind. Dies ist beabsichtigt, da in verwalteten Rechnerverbünden genau dies der Fall sein kann und erwartet werden sollte.

Headless Klassische Clientanwendungen weisen im Normalfall eine grafische Oberfläche für Nutzerinteraktionen auf. Ist es möglich diese Anwendungen eben-

falls ohne Oberfläche einzusetzen, wird dies mit dem Adjektiv/Adverb *headless* bezeichnet. Eine Anwendung ist *headless* oder aber wird *headless* ausgeführt. Bei Serveranwendungen, welche so oder so ohne grafische Oberfläche konzipiert wurden, ist der Begriff zwar anwendbar und nicht falsch, ist jedoch ungebräuchlich.

Architektur der Datenhaltung

Über die Jahrzehnte haben sich verschiedene Architekturen für Datenmodelle für nebenläufige Anwendungen herauskristallisiert. Als wichtigste Vertreter der Ansätze sind *shared-everything*, *shared-memory*, *shared-disk* und *shared-nothing* zu nennen. Dabei beschreibt jeder Ansatz die Art und Weise, wie ein Prozessor im Gesamtsystem Datenhaltung in Bezug auf die anderen Prozessoren betreibt. Da es sich dabei um die grundlegenden Ansätze der Integration nebenläufiger Prozesse handelt und jedes verteilte System im Allgemeinen auch nebenläufig[4] ist, ist es unumgänglich die Vor- und Nachteile im Kontext der verteilten Referenznetzsimulation zu erörtern. Die folgenden Konzepte werden beispielsweise in (Stonebraker, 1986) beschrieben.

shared-everything Bei diesem Ansatz teilen sich die Prozessoren alle verbleibenden Systemkomponenten. Dieser Ansatz ist in seiner Reinform heute selten anzutreffen, da moderne Mehrkernprozessoren zumindest einen eigenen Cache Speicher pro Prozessorkern aufweisen. Im Kontext eines verteilten Systems ist dieser Ansatz nicht relevant oder zielführend, da die Komponenten sehr eng gekoppelt sein müssen. Für jede Lese- oder Schreiboperation des Prozessors müsste im Zweifel ein Nachrichtenaustausch stattfinden. Dieser Overhead ist praktisch in keinem Fall zu rechtfertigen.

shared-memory Beim shared-memory Ansatz teilen sich die verschiedenen Prozessoren einen gemeinsamen Hauptspeicher. Diese Architektur ist heute in Mehrkernprozessorsystemen weit verbreitet. Da die Prozessoren den gleichen Speicher verwenden, kann dieser Ansatz mit sehr schneller Kommunikation aufwarten. Ein Nachteil besteht jedoch in der erschwerten Konsistenzerhaltung der Daten, wenn beispielsweise mehrere Prozessoren an der gleichen Speicheradresse Schreiboperationen ausführen. Zusätzlich kann bei speicherintensiven Anwendungen der gemeinsame Speicher zum Flaschenhals werden. Auf den ersten Blick wirkt dieser Ansatz ungeeignet für ein verteiltes System, da im Zweifel bei jedem Hauptspeicherzugriff ein Nachrichtenaustausch stattfinden müsste. Im Zuge der Containerisierung[5], bei dem mehrere abgeschlossene Systemteile (Container) durchaus auf dem

[4] Es lassen sich auch parallele, getaktete verteilte Systeme konstruieren, diese bauen aber auf der grundlegenden Nebenläufigkeit des verteilten Systems auf.

[5] siehe Abschnitt 2.6.4

gleichen Hostsystem ausgeführt werden können, ist eine Kommunikation über gemeinsamen Hauptspeicher jedoch wieder ein interessanter Ansatz. Grundsätzlich würde es sich dabei streng genommen jedoch nicht mehr um ein verteiltes System handeln, sofern alle Komponenten auf dem gleichen Hostsystem ausgeführt werden.

shared-disk Der shared-disk Ansatz findet u.A. Anwendung bei Applikationen mit einer zentralisierten Datenbank. Alle Prozessoren beziehen sich auf den gleichen persistenten Speicher. Ein großer Vorteil dieses Ansatzes ist die vereinfachte Konsistenzerhaltung der Daten. Die Datenbanktechnologie wurde über die Jahrzehnte immer weiter verfeinert und perfektioniert, sodass heutige Datenbanksysteme hervorragend mit konkurrierenden Speicherzugriffen umgehen können. In einer shared-disk Architektur kann auf diese Erkenntnisse direkt zugegriffen werden, sodass ein fertiges Datenbankmanagementsystem zum Einsatz kommen kann. Ein Nachteil bei diesem Ansatz ergibt sich daraus, dass die zentrale Datenbank als single point of failure angesehen werden muss. Sofern die Datenbank ausfällt und nicht in Replikation vorliegt, erliegt das gesamte verteilte System dem Ausfall. Analog zum shared-memory Modell ist auch hier die Datenbank der zentrale Flaschenhals, sofern datenintensive Aufgaben vom System bewältigt werden. Es muss zusätzlich ein globales Datenmodell definiert werden. Dies kann dazu führen, dass im Falle komplexer Entitäten viel mehr Daten übertragen werden müssen, als beispielsweise bei einer einfachen Berechnung nötig wären. Darüber hinaus gestaltet sich die Skalierbarkeit der Anwendung als schwierig, da die zentrale Datenbank im Normalfall auf einem Rechnerknoten ausgeführt wird. Nichtsdestotrotz gibt es in modernen Datenbanksystemen Bestrebungen zur Verteilung und Replikation, wie beispielsweise das sharding. Kapitel 7 geht detaillierter auf Skalierbarkeitsfragen ein. Für eine erste Fassung der verteilten Referenznetzsimulation ist der shared-disk Ansatz aber durchaus geeignet, da komplizierte Konsistenzfragen an bestehende Datenbanksysteme ausgelagert werden können und der Fokus gänzlich auf die Algorithmen der Referenznetzsimulation gerichtet werden kann.

shared-nothing Falls keine besondere Technologie im verteilten System zum Einsatz kommt, folgt automatisch ein naives shared-nothing Datenmodell. Wie der Name suggeriert, teilen sich die Rechnerknoten keinerlei Bestandteile. Jeder Knoten nutzt und wartet seinen eigenen Hauptspeicher und persistenten Speicher. Die Vorteile liegen darin, dass jeder Knoten sein eigenes Datenmodell und eigene Technologie nutzen kann. Konkurrierende Zugriffe existieren nicht und müssen daher auch nicht behandelt werden. Die Skalierung im shared-nothing Ansatz ist ebenfalls hervorragend, da jeder Bestandteil der verteilten Anwendung seine eigene Datenhaltung betreibt. Die jedoch größte Problematik besteht in der Datenkonsistenz. Durch verschiedene Datenbanken und Datenmodelle existiert im Allgemeinen keine zentrale

Kontrolle oder Transaktionsmanager. Anwendungen müssen die Konsistenz selbst sicherstellen.

Konsistenz in verteilten Systemen

CAP-Theorem

Im Rahmen der Erörterung von Konsistenz in verteilten Referenznetzsimulationen wird es notwendig werden das CAP-Theorem zu betrachten. Da es sich dabei um ein grundlegendes Ergebnis für alle verteilten Systeme handelt, soll es an dieser Stelle einleitend vorgestellt werden.

Das CAP-Theorem wurde erstmals bei einem Keynote Vortrag von Eric Brewer auf dem ACM Symposium on Principles of Distributed Computing (PODC) in 2000 vorgestellt (Brewer, 2000). Durch die Vorstellung adressierte Brewer das Problem, dass die Forschergemeinschaft, die sich verteilten Systemen widmet, und die, die sich der Datenbankentheorie widmen, deutliche Überlappungen haben, aber weitestgehend getrennt agieren und keine übergreifenden Ergebnisse und Theoreme verfügbar waren (Brewer, 2000). Das CAP-Theorem besagt im Kern, dass in jedem verteilten System und jeder verteilten Datenbank zu einer gegebenen Zeit nur zwei der drei Eigenschaften erfüllt sein können:

- Konsistenz (**C**onsistency)
- Verfügbarkeit (**A**vailability)
- Toleranz gegenüber Netzwerkpartitionierung (**P**artition tolerance)

Für zwei Jahre handelte es sich beim CAP-Theorem nur um eine Hypothese, bis im Jahre 2002 ein Beweis des CAP-Theorems von Nancy Lynch und Seth Gilbert vorgestellt wurde (Gilbert und Lynch, 2002). Somit ist unumstritten, dass sowohl damalige als auch heutige – sowie alle zukünftigen – verteilten Systeme diesem Gesetz unterworfen sind.

Die zentrale Frage ist dabei jedoch inwiefern sich das CAP-Theroem konkret auf das Design verteilter Anwendungen auswirkt. Jede der drei Eigenschaften ist von deutlicher Wichtigkeit. Die Erwartungshaltung gegenüber einem System ist es, dass alle drei Eigenschaften zu jeder Zeit erfüllt sind. Systeme großer kommerzieller Anbieter scheinen auch alle Eigenschaften erfüllen zu können, doch der Schein trügt.

Ein Fehlschluss ist außerdem die Annahme, dass sich ein System fix zu zwei dieser drei Eigenschaften bekennen muss. Der Spielraum, den das Theorem offen lässt, umfasst sowohl das Design von Subsystemen als auch eine zeitliche Varianz bezüglich der Erfüllung der Eigenschaften. So kann ein System problemlos sowohl Konsistenz als auch Verfügbarkeit gleichzeitig gewährleisten, solange es zu kei-

ner Netzwerkpartition kommt. Sollte es jedoch zu einer Partitionierung kommen, so muss das zu Grunde liegende Protokoll eine Entscheidung für Verfügbarkeit oder unmittelbare Konsistenz treffen. Konkret bedeutet dies entweder die letzten bekannten Daten auszuliefern oder aber nur solche, die sicher konsistent sind.

ACID

ACID ist eins der bekanntesten konsistenzorientierten Grundprinzipien, welches vielfach in der Datenbanktheorie speziell bei Transaktionen zum Einsatz kommt. Es wurde erstmals in (Härder und Reuter, 1983) beschrieben. Eine umfassende Einführung wird beispielsweise in (Kemper und Eickler, 2011) gegeben. ACID ist ein Akronym und steht für die vier Eigenschaften einer Transaktion:

(A)tomarität Die Operation soll äquivalent zu einer atomaren Änderung sein, es sollen also alle oder keine Änderungen erfolgen.

(C)onsistency / dt. Konsistenz Nach Ausführung sollen die Daten in einem konsistenten Zustand sein, sofern sie dies vor der Ausführung bereits waren. Insbesondere sollen keine Datenbankconstraints verletzt sein oder widersprüchliche Daten im System (bzw. der Datenbank) vorhanden sein.

(I)solation Jede Operation darf nicht derart durch nebenläufig laufende Operationen beeinflusst werden wie es nicht auch in einer sequentiellen Ausführung beider Operationen möglich wäre.

(D)urability / dt. Persistenz Änderungen sollen persistent gesichert sein und Systemabstürze oder beispielsweise Stromausfälle überdauern.

Zu diesen verschiedenen Eigenschaften existiert eine Vielzahl an Abstufungen und verschiedenen Implementationen. Da im Rahmen dieser Arbeit insbesondere die folgende Abgrenzung zum *BASE*-Ansatz von Interesse ist, sei an dieser Stelle aber auf die Ausführung verzichtet.

Verteilte Transaktionen

Einer der ältesten und bekanntesten Ansätze in verteilten Systemen Datenkonsistenz herzustellen bildet die verteilte Transaktion und insbesondere die verteilte Commitverwaltung. Es existieren verschiedene Ansätze und Standards, wobei der wohl bekannteste von der Open Group[6] betreut und unter dem Namen X/Open DTP[7] geführt wird. Den meisten verteilten Commitprotokollen ist gemein, dass mit Hilfe eines Koordinators ab einem gewissen Punkt in der Abarbeitung die Knotenautonomie der Nicht-Koordinatoren aufgegeben wird und diese auf die Anweisung zum Commit des Koordinators warten. Die Zahl der Zwischenschritte, die notwendig ist, bis es zur Aufgabe dieser Autonomie kommt, bestimmt meist maßgeblich das Protokoll. Im Allgemeinen werden ein-, zwei- und drei-

[6] https://www.opengroup.org/

[7] »Distributed Transaction Processing«, dt. »Verteilte Transaktionsverwaltung«, siehe beispielsweise (Gray und Reuter, 1992)

Phasen-Commit-Protokolle betrachtet, wobei die zweiphasige Variante am häufigsten anzutreffend ist. Verteilte Transaktionsverwaltung verfolgt das primäre Ziel der Konsistenz in den entfernten Datenbankinstanzen. Im Zweifel und bei Netzwerkausfall wird das System unreaktiv, da Sperren das Lesen von (unsauberen) Daten verhindern können. Verteilte Transaktionsverwaltung stellt somit Konsistenz über Verfügbarkeit bei Auftreten von Netzwerkpartitionierung.

BASE
Das Akronym BASE steht für die Bestandteile:

(B)asically (A)vailable / dt. Generell verfügbar Daten sind generell verfügbar, wenn sie angefordert werden.

(S)oft state / dt. »Weicher« Zustand Der Zustand der Daten ist nicht immer eindeutig (global) definiert und kann zeitweilig inkonsistent sein.

(E)ventual Consistency / dt. Letztendliche Konsistenz Das System konvergiert über die Zeit sicher in einen konsistenten Zustand.

BASE wurde insbesondere entwickelt, um mit den wachsenden Datenmengen in sozialen Netzwerken und anderen Big Data Anwendungen umzugehen. Der Artikel (Pritchett, 2008) gibt beispielsweise eine Einführung in das BASE Prinzip. BASE entwickelte sich in direktem Anschluss an die Erkenntnisse aus dem CAP Theorem.

BASE gibt wesentlich weniger strikte Vorgaben als das verwandte ACID-Prinzip. Das Grundprinzip sieht vor, dass Daten zu jeder Zeit gelesen oder geschrieben werden können. Dabei kann es jedoch passieren, dass durch die Konvergenz zu einem konsistenten Zustand Daten nachträglich geändert werden und somit getätigte Schreibvorgänge wieder überschrieben werden. Beim Lesen aus einer lokalen Datenbank kann somit nicht generell ein konsistenter Zustand garantiert werden, da sich das System zu jeder Zeit im Prozess der Konvergenz befinden kann. Stammen Daten aus Datenbanken, welche nach dem BASE Prinzip agieren, sollten diese mit Vorsicht behandelt werden und gegebenenfalls nach einiger Zeit erneut geprüft werden.

Saga-Pattern
Eine weiterer Ansatz der Konsistenzherstellung in einem verteilen System besteht in der Verwendung des sog. Saga-Patterns. Es ist eine mögliche Implementation einer verteilten Konsistenzherstellung nach dem BASE-Ansatz. Eine Saga wird hierbei absichtlich nicht als »verteilte Transaktion« bezeichnet, da sie nicht alle Eigenschaften einer Transaktion besitzt. Das Saga-Pattern wurde ursprünglich von Hector Garcia-Molina und Kenneth Salem 1987 vorgestellt (Garcia-Molina und Salem, 1987) und wurde seitdem mehrere tausend Male zitiert. Im Kontext des Aufstiegs von Microservices erlebte es dann zur Zeit der 2010er Jahre einen Aufschwung.

Zu aktuellerer Literatur diesbezüglich zählt beispielsweise das Buch »Microservices Patterns« (Richardson, 2019). Entsprechende Onlinequellen zur Veröffentlichung Richardsons existieren ebenso[8]. Die hier vorgestellte Form des Saga-Patterns und seine Bedeutung im Kontext von verteilten Systemen und insbesondere einer Microservicearchitektur richtet sich im Wesentlichen nach (Richardson, 2019). Microservices werden im Abschnitt 2.6.1 noch einmal gesondert eingeführt und definiert.

Das Saga-Pattern garantiert die Eigenschaften Atomizität, Konsistenz und Dauerhaftigkeit. In seiner grundlegenden Form garantiert es keine Isolation, sodass mit entsprechenden Anomalien umgegangen werden muss. Ein großer Vorteil besteht darin, dass es ohne Blockierung der Systemkomponenten auskommt, was ein entscheidender Vorteil gegenüber der klassischen verteilten Transaktionsverwaltung ist.

Eine Saga ist eine Folge von lokalen Transaktionen, welche auf den verschiedenen Komponenten des verteilten Systems ausgeführt werden. Dabei ist nicht vorgegeben welche Techniken, Datenbankformate, etc. dabei lokal zum Einsatz kommen. So kann eine Komponente auf einer klassischen relationalen Datenbank basieren und eine andere eine dokumentenorientierte NoSQL Datenbank einsetzen.

Der Saga-Ansatz sieht kein Zurückrollen (»Rollback«) einer Saga im Fehlerfall vor, sondern arbeitet mit kompensierenden Transaktionen. Dabei unterscheidet der Saga Ansatz zwischen drei Arten von lokalen Transaktionen: »Kompensierbare Transaktionen«, »Wiederholbare und garantiert erfolgreiche Transaktionen« und die sogenannten »PivotTransaktionen«.

Kompensierbare Transaktionen Diese Transaktionen haben so überschaubaren Einfluss im lokalen System, dass ihre Effekte problemlos durch eine weitere Transaktion aufgehoben werden können. Im Falle eines Fehlschlags der ganzen Saga, können die lokalen Änderungen so wieder in den ursprünglichen Zustand versetzt werden. Beispiele für kompensierbare Transaktionen sind das Anlegen eines neuen Auftragsobjekts in einem Bestellsystem sowie das Verbrauchen einer Marke in einem Petrinetz.

Wiederholbare Transaktionen Bei wiederholbaren Transaktionen handelt es sich um solche, die bei einem Fehlschlag erneut ausgeführt werden können, da sie garantiert erfolgreich sind und nur durch Systemabstürze oder Netzwerkpartitionierungen oder ähnliche äußere Einflüsse zum Scheitern gebracht werden können. Im Fehlerfall kann die Transaktion einfach erneut ausgeführt werden. Beispiele wären die Ablage von Dokumenten in einem Archiv oder das Erzeugen einer Marke durch eine Transition in einem Petrinetz. Wichtig ist an dieser Stelle noch, dass es sich bei der Transaktion nicht zwangsläufig um eine idempotente Operation handeln muss. Die Erwartung

[8] https://microservices.io/patterns/data/saga.html - Zuletzt abgerufen am 22.09.2019

ist lediglich so weit, dass der lokale Transaktionsmanager die eigene Datenhaltung in einen Zustand versetzt, die dem einmaligen Ausführen entspricht, sei es durch Idempotenz der Operation oder durch entsprechendes Logging und Fehlerbehandlungen.

Pivot Transaktionen Diese Transaktionen sind sowohl zentrales Element des Saga-Patterns, als auch eine weitere Einschränkung, die damit einhergeht. Eine Pivot-Transaktion ist eine Transaktion, die nicht kompensierbar ist und deren Ausgang im Vorhinein ungewiss ist. Sie stellt den zentralen (mittleren) Bereich einer Saga dar und bestimmt über Erfolg oder Misserfolg der ganzen Saga. Ein Beispiel wäre die Abbuchung von einer Kreditkarte von einem Online-Zahlungsdienstleister oder speziell im Kontext der Referenznetze die Ausführung einer »action« Anschrift, welche explizit für seiteneffektbehaftete Operationen vorgesehen ist.

Eine Saga wird im Normalfall so konzipiert, dass zunächst alle kompensierbaren Transaktionen ausgeführt werden, dann genau eine Pivot Transaktion und abschließend alle wiederholbaren Transaktionen.

Für die konkrete Ausführung der Saga existieren zwei Ansätze, die (Richardson, 2019) als *Choreography* (dt. Choreographie) und *Orchestration* (dt. ebenso) vorstellt. Die grundlegenden Unterschiede finden sich darin, dass eine Choreographie einen dezentralen, verteilten Charakter hat, wohingegen eine Orchestration mit einem zentralen Verwalter agiert.

Choreographie Beim diesem Ansatz existiert keine zentrale Verwaltungsinstanz innerhalb des verteilten Systems, welche die Saga koordiniert. Die Vorteile liegen durch eine vollständige Dezentralität auf der Hand, es gibt keinen Single Point of Failure und Teile des verteilten Systems können autonom agieren. Damit geht jedoch einher, dass jeder Systemteil zu jeder möglichen Saga die Services kennen muss, die in der Reihenfolge dahinter liegen. Da wie bereits erwähnt in einer Saga auch kompensierbare Transaktionen auftreten können, muss ein Service für diese auch den Vorgängerservice kennen, um ein etwaiges Kompensieren zu ermöglichen. Dadurch entsteht eine deutlich engere Kopplung zwischen den Services. Ein weiterer Nachteil besteht darin, dass bei der Änderung einer Saga diverse Komponenten geändert werden müssen, in einer Entwicklungsumgebung mit mehreren Teams, unter Umständen durch mehrere oder alle Teams. Zusätzlich muss etwaige Nebenläufigkeit an vielen Stellen gleichzeitig umgesetzt werden, beispielsweise dadurch, dass ein Service zwei Nachfolge-Services anstößt.

Orchestration Das logische Gegenstück zur Choreographie bildet die Orchestration. Hierbei übernimmt ein Koordinator die Ausführung der Saga. Der Koordinator kennt alle Services, die an der Saga beteiligt sind und führt die Operationen in entsprechender Reihenfolge durch. Da nur der Koordinator

Wissen über eine Saga hält, können die Services mit einem neutralen Interface implementiert werden. Die Kopplung sinkt und der Wartungsaufwand für eine Saga wird übersichtlicher. Auch kann durch Monitoring Features wesentlich einfacher ein Zustand über die Abarbeitung verschiedener Sagas abgerufen werden. Ein weiterer Vorteil ergibt sich durch die Möglichkeit der Beeinflussung der Abarbeitung durch die zentrale Kontrollinstanz. Dieser Aspekt wird sich später im Rahmen der Arbeit als hilfreich erweisen. Im Gegensatz zur Choreographie ergibt sich der offensichtliche Nachteil des Single Point of Failures beim Orchestration-Ansatz.

In der Literatur wird der Ablauf einer gegebenen Saga meist durch eine *finite state machine* (dt. endliche Zustandsmaschine, endlicher Automat) beschrieben. Für einfache Abläufe ist dieser Ansatz sicher ausreichend, es können damit einfache Abläufe und Konditionale modelliert werden. Um eine Saga zu implementieren übernimmt beispielsweise der Koordinator in einer Orchestration die Ausführung des endlichen Automaten.

Zustandslosigkeit und REST

Zustandslosigkeit beschreibt dem Namen entsprechend die Abwesenheit eines Zustandes. Der Begriff kann sowohl auf Protokolle als auch auf Systeme oder Systemteile angewandt werden. Die Beschreibung in diesem Abschnitt richtet sich im Wesentlichen nach (Fielding, 2000).

Zustandslose Systemkomponenten wie beispielsweise Webserver halten keine clientbezogenen Informationen z.B. in der Form von Sessions vor. Dies ist nicht per Definition als Vorteil zu werten, sondern umfasst wie viele architekturelle Entscheidungen einen trade-off. Zunächst fällt auf, dass derartige Systemkomponenten als Sammlung verschiedener Prozeduren gesehen werden können, da sie per Definition keine (persistenten) Daten enthalten. Aus diesem Grund ist eine Replikation des Systemteils problemlos möglich, da keine Datenkonflikte entstehen können. Auf der Negativseite müssen jedoch mehr Daten zum System übermittelt werden, um eine Abarbeitung zu ermöglichen. Eine derartige Bereitstellung der Daten lässt sich jedoch meist nebenläufig realisieren, insbesondere, wenn mehrere Clients involviert sind. Aus diesem Grund sind zustandslose Systeme generell als günstig im Bezug auf ihre Skalierbarkeit zu bewerten, auch wenn individuell und fallbezogen einzelne Unterschiede existieren.

Anfragen an zustandslose Systemkomponenten erfolgen über zustandslose Protokolle. Bei einer Anfrage müssen daher alle relevanten Daten entweder direkt im Aufruf mitgeliefert werden oder aber während der Abarbeitung aus externen Quellen bezogen werden. Das Übermitteln eines solchen repräsentierenden Zustands ist der Kerninhalt des REST (»Representational State Transfer«) Paradigmas.

REST geht ebenfalls auf (Fielding, 2000) zurück. In produktiven Anwendungen wird REST fast ausschließlich über das HTTP Protokoll realisiert, ist jedoch konzeptuell unabhängig davon. REST definiert noch weitere Eigenschaften, für den hier gesteckten Kontext soll diese knappe Einführung jedoch zunächst genügen.

2.5.3. Einheitentheorie

Die Einheitentheorie geht im Wesentlichen auf Daniel Moldt zurück. Dabei existieren verschiedene Veröffentlichungen, bei denen das grundlegende Konzept beschrieben wird wie beispielsweise (Moldt, 2005; Tell, 2005; Tell und Moldt, 2005). Eine weitere Arbeit, welche sich mit der Thematik befasst ist (Hewelt, 2010). Die Einheitentheorie beschreibt den Ansatz, dass sich Einheiten stets durch drei konkrete Eigenschaften charakterisieren:

- Die Einbettung in eine übergeordnete Einheit und die Natur der Kommunikation mit dieser.
- Die Kommunikation mit gleichberechtigten Einheiten, diese erfolgt normalerweise mit Hilfe der übergeordneten Einheit.
- Die Kommunikation mit untergeordneten Einheiten

Der Begriff »Einheit« ist dabei vielseitig verwendbar und dient als eine der höchstmöglichen Abstraktionen. Es kann sich dabei um beliebige physikalische oder logische Entitäten handeln, welche ihre konkrete Ausprägung über die Ausdefinition der oben genannten drei Eigenschaften erlangen.

Die wesentlichen Vorteile durch die Einheitentheorie ergeben sich daraus, dass sich mit ihrer Hilfe auf Basis der Nebenläufigkeitstheorie, modelliert mittels Petrinetzen, Konzepte ineinander überführen lassen. Diese Eigenschaft wird im Rahmen der Anforderungsanalyse in Abschnitt 4.1 wertvolle Hilfestellungen bei der konzeptionellen Modellierung des skalierenden Agentensystems bieten.

Zur Illustration der Generalität des Konzepts folgen dieser Stelle noch einige Beispiele für sehr verschiedene konkrete Einheiten:

Agenten Agenten betten sich in einer Plattform ein, von der sie erzeugt und zerstört werden (bzw. sie betreten und verlassen oder dort erscheinen und verschwinden) können. Sie kommunizieren untereinander durch den Versand von Nachrichten und kontrollieren ihre internen Protokolle, welche auf der Basis von Wissensdatenbanken, Events und Verhalten Entscheidungen treffen.

Computer im Netzwerk Computer sind in einem Netzwerk eingebettet, sie können etwaige Router oder andere Hardware des Netzwerkes steuern, es jedoch

auch zur Kommunikation untereinander verwenden. Nach unten gerichtet steuern Computer ihre Interna durch Programmcode und Datenbestände.

Unternehmen am Markt Unternehmen sind in all ihrer wirtschaftlichen Tätigkeit in den globalen Weltmarkt eingebettet. Durch das Schalten von Angeboten und Nachfragen können sie mit dem Markt interagieren, durch den An- und Verkauf von Waren und Dienstleistungen über den Markt mit anderen Unternehmen kommunizieren (bzw. in diesem Fall kooperieren). Gleichzeitig steuert das Unternehmen seine Interna durch die Geschäftsleitung, Mitarbeiter, Vermögen und Geschäftsprozesse.

Webservices Webservices betten sich in eine Deployment-Landschaft (beispielsweise innerhalb einer Cloud-Struktur) ein. Sie kommunizieren durch Mittel des Deployments, wie beispielsweise Nachrichtenbroker, können durch das Deployment erschaffen oder vernichtet werden und sich beispielsweise bei Service Registern anmelden. Nach innen erbringen sie Leistungen mithilfe interner Programmierung und durch Nachrichten an gleichberechtigte Datenbanken innerhalb des Deployments.

2.6. Skalierbarkeit

Skalierbarkeit beschreibt die Beschleunigung von paralleler sowie nebenläufiger Software bei der Änderung bzw. Erhöhung der ausführenden Prozessoren. Eine einheitliche Definition des Begriffs existiert jedoch nicht in der Literatur, wie beispielsweise (Hill, 1990) oder (Duboc, Rosenblum und Wicks, 2006) anmerken.

(Grama, Gupta und Kumar, 1993) definieren Skalierbarkeit wie folgt:

> *Analyzing the performance of a given parallel algorithm/architecture calls for a comprehensive method that accounts for scalability: the system's ability to increase speedup as the number of processors increases.*
> *(dt.: Die Analyse der Performanz eines gegebenen parallelen Algorithmus/einer Architektur benötigt eine umfassende Methode, die Skalierbarkeit einbezieht: Die Fähigkeit des Systems die Beschleunigung zu erhöhen, wenn die Zahl der Prozessoren steigt.)*
> — (Grama, Gupta und Kumar, 1993, Seite 12)

Eine weitere Definition durch (Nicol, 1998) lautet:

> *Scalability analysis asks how performance of a certain application (or application class) behaves as the application problem size increases and the parallel architecture executing it increases.*
> *(dt.: Die Skalierbarkeitsanalyse adressiert, wie sich die Performanz einer bestimmten Anwendung (oder Anwendungsklasse) verhält, wenn die Größe des Anwendungsproblems und die parallele Architektur, welche es ausführt, größer wird.)*
> — (Nicol, 1998, Seite 4)

Die verschiedenen Facetten einer nebenläufigen Architektur führen dazu, dass der Begriff der Skalierbarkeit nicht leicht einzugrenzen ist. Wird der Aspekt der Verteilung zusätzlich in die Betrachtung einbezogen, erhöhen sich die Zahl der Variablen und die inhärente Unsicherheit weiter. Aus diesem Grund soll eine Definition von Skalierbarkeit erfolgen, welche den Rahmenbedingungen der Arbeit bestmöglich Rechnung trägt. Die Abbildung interagierender Agenten, welche Plattformen für ihre Interaktion schaffen können, kann durch die dynamische Anpassung der Menge der existierenden Plattformen beschrieben werden. In einer entsprechenden Simulation liegt es nahe, Plattformen als Systembestandteile bzw. eigenständige Softwaresysteme zu sehen. Im Sinne der agentenorientierten Softwareentwicklung und in Hinblick auf die Autonomie eines Agentensystems und seiner Agenten erfolgt im Rahmen dieser Arbeit die Definition:

Definition 2.15 (Skalierbarkeit)**.** *Skalierbarkeit* beschreibt die Fähigkeit eines Softwaresystems neu zur Verfügung gestellte Ressourcen automatisiert derart ausnutzen zu können, dass dabei eine bessere Leistung entsteht, als wenn diese Ressourcen nicht zur Verfügung stünden.

Dabei sei zu beachten, dass diese Definition allgemeiner als die im Vorfeld zitierten Definitionen gehalten ist und explizit den Aspekt der Automatisierung herausstellt. Für die Fälle von einfachen Algorithmen, welche homogene Probleme adressieren, kollabiert die Anforderung in die in den oben genannten Zitaten adressierte Erhöhung der Performanz mit Erhöhung der Prozessoranzahl. Da jedoch heterogene Umgebungen elementarer Teil der Betrachtungen sind, ist eine weiter gefasste Definition für den Kontext der Arbeit hilfreich.

Für die Umsetzung von Skalierbarkeit existiert eine Vielzahl an Untersuchungen. Im Sinne der Forschungsfragen sind hier insbesondere Softwarearchitekturen und Entwicklungsmuster von Interesse. Diese existieren hierbei auf sehr unterschiedlichen Granularitätsebenen. Manche beschreiben das System als Gesamtkonzept, einzelne nur die individuellen Anwendungen, manche nur einzelne Aspekte. Durch diese Unterschiede sind die bestehenden Architekturen meist nicht umfänglich vergleichbar, da sie verschiedene, teils orthogonale Aspekte adressieren. Dennoch lassen sich grobe Kategorien ausmachen: Somit behandeln die nachfolgenden Unterabschnitte zunächst grobgranulare Entwurfsmuster auf der Systemebene und

anschließend feingranulare Entwurfsmuster auf der Ebene der Einzelanwendungen.

Da neben der agentenorientierten Softwareentwicklung die Szenenanalyse Teil des Rahmens der Arbeit ist, sei an dieser Stelle noch einmal darauf hingewiesen, dass der Begriff *Skalierbarkeit* im Rahmen der Arbeit *nicht* den Begriff der Skalierbarkeit innerhalb eines Gaussschen Skalenraums bezeichnet, wie er in Bildverarbeitungsarbeiten häufig eingesetzt wird.

2.6.1. Grobgranulare Entwicklungsmuster

Abschnitt 2.5.1 hat bereits den Begriff der Kohäsion und der Koppelung eingeführt. Die erste Betrachtung übergreifender Systemlösungen soll daher insbesondere diesen Aspekt adressieren. Dabei spielt eine besondere Rolle, inwiefern sich mit diesen Ansätzen ein skalierbares verteiltes System realisieren lässt.

Monolith

Ein monolithischer Aufbau bezeichnet die Konstruktion einer Software in einem großen Einzelverbund, welcher ausschließlich in seiner Gesamtheit oder gar nicht ausgeführt werden kann. Eng betrachtet widerspricht durch diese Eigenschaft diese Form der Auslieferung dem Vorhaben ein verteiltes System zu entwickeln, da es sich um lediglich eine Komponente handelt. Jedoch existieren Formen, bei denen bestimmte Bestandteile des Systems auf anderen Rechnerknoten ausgelagert sind, dabei jedoch nicht ohne den übrigen Kontext lauffähig sind. Dies kann sich durch synchrone Aufrufe, zyklische Abhängigkeiten zwischen den Klassen, Notwendigkeit von bestimmten Verhaltensmustern anderer Komponenten und ähnlichen Zusammenhängen begründen.

Eine weitere Möglichkeit dieser Form der Architektur besteht darin mehrere gleichartige Instanzen eines Monolithen zu betreiben. Ein vorgeschalteter Load-Balancer kann dann Anfragen an das korrekte System weiterleiten.

All diese Varianten eines Monolithen haben jedoch gemeinsam, dass sie kaum Sicherheitsvorkehrungen bereitstellen, falls einige Bestandteile des Systems in einen Fehlerzustand geraten. So kann ein Speicher-Leck in einer Komponente den Ausfall des gesamten Systems nach sich ziehen. Auch Updates sind aufwändig umzusetzen und ziehen zwangsläufig eine Nicht-Verfügbarkeit des Systems nach sich. Monolithen fallen durch eine starke Koppelung und eine geringe Kohäsion auf. Bei Skalierungsfragen verhalten sich monolithische Systeme somit ungünstig.

Monolithische Systeme sind inhärent einfacher zu konzipieren, da durch die umfassende Zentralisierung auf viele komplizierte Lösungen verzichtet werden kann. Viele natürlich gewachsene Systeme sind monolithisch aufgebaut.

Auf der positiven Seite ist die geringere Komplexität und das vergleichsweise einfache Deployment eines Monolithen zu erwähnen. Tests können meist im Kontext des Gesamtsystems erfolgen. Auf die Verwendung von Infrastrukturmanagern kann zudem meist verzichtet werden und diese Form von Deployments synergiert gut mit dem klassischen Ansatz dedizierte Bare-Metal-Hosts[9] zu betreiben. Dort kann dann ein unkompliziertes 1:1 Deployment zwischen Server und Anwendung realisiert werden.

Modulith

Der Modulith ist ein Kunstwort aus den Begriffen »Modul« und »Monolith«. Mit ihm wird eine monolithische Anwendung bezeichnet, die jedoch in ihrer internen Struktur auf Entkopplung und Modularisierung setzt. Das Deployment orientiert sich im Wesentlichen am Deployment eines Monolithen. Je nach intendiertem Nutzen können jedoch bestimmte Bestandteile des Systems aktiviert oder deaktiviert werden, beispielsweise durch Konfigurationen oder ein Plugin-System.

Zu den zentralen Vorteilen zählt hier die Flexibilität, da mit überschaubarem Aufwand ein individualisiertes System zusammengestellt werden kann. Auch Tests können analog zum Monolithen in einer definierten Form des Gesamtsystems ausgeführt werden. Infrastrukturmanagement und Verzeichnisdienste sind nur bedingt erforderlich, da viele Bestandteile im lokalen Einflussbereich der Anwendung liegen.

Bei der Konstruktion von verteilten Systemen bestehen im Wesentlichen dieselben Probleme und Nachteile, wie beim Einsatz monolithischer Software. Durch die Modularisierung kann jedoch eine weitere Instanz auf etwaige zusätzlich benötigte Anwendungsprofile angepasst werden. Somit muss nicht stets ein ganzer weiterer Monolith hinzugeschaltet werden. Modulithen können eine höhere Kohäsion als Monolithen besitzen, sofern die Modularität ausreichend ausgenutzt wird. Die Koppelung ist nach wie vor hoch. Insgesamt lässt sich auch mit dieser Architektur nur schwer Skalierbarkeit herstellen.

[9] »Bare-Metal« bezeichnet das Deployment direkt auf physikalischen Computern im Gegensatz zu verschiedenen Virtualisierungslösungen. Weitere Details werden in Abschnitt 2.6.4 eingeführt.

Microservices

Microservices sind eine relativ neue Entwicklung und sind bereits mit dem Einsatz als verteiltes System und mit der Intention skalierbar zu sein entworfen worden. Es existiert keine komplett eindeutige Definition von Microservices, sondern viele verschiedene Abhandlungen verschiedener Autoren. Eine gute Einführung bietet (Fowler und Lewis, 2014). An dieser Stelle wird eine Definition für Microservices gegeben, wie sie im Kontext der Arbeit verstanden werden sollen:

Definition 2.16. *Microservices* sind ein Architekturstil, bei dem die einzelnen Aufgaben und Funktionalitäten eines Systems als jeweilige eigenständige Services realisiert werden. Services sind autonom, haben eine einzelne klar umrissene und explizite Aufgabe und können stets ohne weitere Services existieren. Microservices setzen ein shared-nothing Datenmodell ein.

Wenn abhängige Services nicht vorhanden sind, können Anfragen ggf. nicht zufriedenstellend bearbeitet werden, die Lauffähigkeit des betrachteten Services ist jedoch nicht beeinträchtigt. Ein Service wird unabhängig von anderen Services entwickelt und deployed. Darüber hinaus können die Services in Bezug auf Programmiersprache, Datenbanktechnologie, etc. starke Heterogenität aufweisen.

Services kommunizieren über ein leichtgewichtiges Kommunikationsprotokoll. Sie sind darüber hinaus in der Verantwortung ihr eigenes Datenmodell umzusetzen.

In Abgrenzung zur sogenannten SOA (»Service-oriented Architecture«), welche nicht mehr dem aktuellen Stand der Technik entspricht und welche daher hier nicht detaillierter eingeführt wird, setzen Microservices insbesondere sogenannte »dumb pipes« (dt.: »dumme Kanäle«), während in einer SOA »smart pipes« (dt.: »intelligente Kanäle«) vorgesehen sind. Die Bezeichnungen betreffen das Routing der Nachrichten. In einer SOA verteilt eine zentrale Einrichtung die Nachrichten an die korrekten Empfänger, bei der Microservice-Architketur ist dies nicht vorgesehen. Einer der Hauptgründe umfasst die verbesserte Skalierbarkeit durch den Verzicht auf eine zentrale Komponente.

Microservices können die Problematik des Informationsaustausches auf verschiedene Weise lösen. Die Möglichkeiten umfassen beispielsweise den Einsatz eines Poll-basierten Publish-Subscribe Brokers oder aber die direkte Kommunikation und Netzwerkkoordinatenauflösung durch einen Verzeichnisdienst.

Das Ziel von Microservices ist die Reduktion von Koppelung und die Erhöhung der Kohäsion. Sie sind auf natürliche Weise für eine skalierbare Architektur geeignet. Die Umsetzung ist jedoch häufig nicht trivial und erfordert meist weitere Komponenten in der Gesamtarchitektur.

Event-Sourcing
Eine der Herausforderungen in einer Microservice-Umgebung ist die Übermittlung von Daten und die Beschreibung des Zustands des Systems. Durch seine inhärent verteilte Natur kann nicht gewährleistet werden, dass jeder Service zu jeder Zeit an allen Kommunikationen teilnehmen kann. Auch wäre dieser Ansatz sehr limitierend, da viele Kommunikationen gegebenenfalls einige Services nicht betreffen. Aus diesem Grund wird der Ansatz des *Event-Sourcing* vermehrt bei Microservice Deployments eingesetzt.

Der Grundgedanke bei diesem Entwicklungsmuster liegt darin, den Zustand des Systems nicht durch einen absoluten Zustand zu beschreiben, sondern vielmehr als Summe verschiedener Ereignisse (*Events*) zu betrachten. Der Ansatz ist nicht neu und wurde auch nicht für den Kontext von Microservices erfunden. Er entspricht im Wesentlichen beispielsweise dem Führen eines Kassenbuchs oder Kontos, bei dem sich der aktuelle Zustand (der aktuelle Saldo) ebenfalls durch gespeicherte Buchungen errechnen lässt. Dies ist insbesondere auch dann möglich, wenn der Betrachter (bzw. in diesem Kontext ein bestimmter Service) zum Zeitpunkt der Buchung nicht an der Kommunikation beteiligt war, sondern diese nur nachträglich mitgeteilt bekommt. Event-Sourcing kann analog zu einem Konto auch durch das Speichern von Zwischenzuständen angereichert werden ebenso wie beim Konto zu einem Stichtag ein bestimmter Saldo gespeichert wird.

Auch das in Abschnitt 2.5.2 vorgestellte Saga-Pattern kann gut auf der Basis von Event-Sourcing implementiert werden. Durch die eben beschriebenen Eigenschaften kann leicht nachvollzogen werden, dass es sich bei einem Protokoll auf Basis des Event-Sourcings um ein Protokoll handelt, welches Eventual Consistency (siehe »BASE« in Abschnitt 2.5.2) umsetzt.

2.6.2. Feingranulare Entwicklungsmuster

Wie bereits einleitend erwähnt weisen die nun folgenden feingranularen Entwicklungsmuster unterschiedliche Umfänge auf. Sie stehen daher nur bedingt in Konkurrenz zueinander, geben aber alle nützliche Einblicke in die Gestaltung skalierbarer Systeme.

Hexagonale Architektur und Clean Architecture

Die Bezeichnung »Hexagonale Architektur« oder auch »Ports und Adapter Architektur« bzw. »Clean Architecture« geht zurück auf Alistair Cockburn[10]. Der

[10] https://web.archive.org/web/20170916120520/http://alistair.cockburn.us/Hexagonal+architecture - Zuletzt abgerufen am 11.12.2021.

wesentliche Ansatz der Architektur besteht darin, die Anwendungslogik unabhängig von externen Referenzen in einem eigenständigen Kern zu implementieren. Diese Anwendungslogik stellt gewisse Ports bereit, an welche wiederum Adapter angeschlossen werden können, die externe Quellen involvieren. Adapter können auf die zentralen Abläufe zugreifen, die zentralen Abläufe besitzen jedoch keinen Zugang zu (speziellen) Adaptern.

Ein zentraler Vorteil der Architektur ist die Unabhängigkeit der fachlichen Implementation von konkreten technischen Umsetzungen in der Peripherie der Anwendung. Darüber hinaus ist die Umsetzung von sogenannten Anti-Corruption-Layern[11] einfach, da diese als Adapter umgesetzt werden können.

Der Einsatz in skalierenden Systemen ist von Interesse, da Anwendungen nach der hexagonalen Architektur häufig resistenter gegen Ausfälle von Fremdsystemen sind und können daher insbesondere in Microservice-Deployments von Vorteil sein.

Die Twelve-factor App

Der Architekturstil der Twelve-factor App geht zurück auf Adam Wiggins und viele weitere Mitautoren. Er entstand im Rahmen der Arbeiten am Heroku[12] Projekt, welches einen der ersten Begründer moderner Cloud-Plattformen darstellt. Eine Twelve-factor App besitzt nach den Autoren die folgenden Eigenschaften:

1. *Codebase: Eine im Versionsmanagementsystem verwaltete Codebase, viele Deployments*
2. *Abhängigkeiten: Abhängigkeiten explizit deklarieren und isolieren*
3. *Konfiguration: Die Konfiguration in Umgebungsvariablen ablegen*
4. *Unterstützende Dienste: Unterstützende Dienste als angehängte Ressourcen behandeln*
5. *Build, release, run: Build- und Run-Phase strikt trennen*
6. *Prozesse: Die App als einen oder mehrere Prozesse ausführen*

[11] Anti-Corruption-Layer (ACLs) bieten eine verlässliche Schnittstelle innerhalb einer Anwendung und beheben etwaige Inkonsistenzen und Ausfälle einer externen Informationsquelle.
[12] https://www.heroku.com/ - Zuletzt abgerufen am 13.12.2021

> 7. *Bindung an Ports: Dienste durch das Binden von Ports exportieren*
> 8. *Nebenläufigkeit: Mit dem Prozess-Modell skalieren*
> 9. *Einweggebrauch: Robuster mit schnellem Start und problemlosen Stopp*
> 10. *Dev-Prod-Vergleichbarkeit: Entwicklung, Staging und Produktion so ähnlich wie möglich halten*
> 11. *Logs: Logs als Strom von Ereignissen behandeln*
> 12. *Admin-Prozesse: Admin/Management-Aufgaben als einmalige Vorgänge behandeln*
>
> — Adam Wiggins et. al.: »The Twelve-factor App« (https://12factor.net/de/)

Die Vorgaben umfassen sehr viele Bereiche und umfassen weitestgehend die gesamte Deploymentlandschaft. Die eigentliche Architektur der Anwendung wird nur in geringem Umfang und auf Metaebene beschrieben. Die Vorgaben sind technisch orientiert und teilweise sehr spezifisch (beispielsweise das Binden an Ports oder der Einsatz von Umgebunsgvariablen). Während viele der Eigenschaften zunächst generisch erscheinen, liefert das Zusammenspiel vielfach die Basis moderner Cloud-Anwendungen.

Eine Twelve-Factor App zeichnet sich durch den Einsatz verschiedener zustandsloser Prozesse in einem Zusammenschluss aus, welche in ihrer Gesamtheit die Aufgaben bewältigen. Skalierbarkeit entsteht durch das Replizieren einzelner Prozesse. Prozesse sind nach Bedarf herunter- und hochfahrbar und austauschbar.

Für die Umgebungen (Entwicklung, Qualitätssicherung, Produktion) werden ferner die Vorgaben gemacht, dass diese möglichst ähnlich und vergleichbar sein sollten. Ein Codestand im Repository wird stets zunächst auf die niedrigeren Umgebungen ausgerollt, es werden keine Code-Stände direkt in die Produktion gebracht. Dabei wird strikt zwischen Konstruktion und Ausführung getrennt, sodass keine Anpassung im laufenden Betrieb ohne vorheriges Release erfolgt.

Cloud-Nativity

Als Anlehnung an die Twelve-factor App wird in neuerer Zeit der Begriff der Cloud-Nativity diskutiert. Cloud-Nativity ist dabei ein vielfach verwendeter Begriff, der sich gerade im Marketing in Unternehmenskontexten einer großen Beliebtheit erfreut. Nach der Definition, wie sie im Rahmen dieser Arbeit verwendet werden soll, verbirgt sich dahinter ein äußerst nützliches Konzept, welches zu

den Möglichkeiten der Skalierung einer Anwendung entschieden beitragen kann und das Deployment auf moderne Infrastrukturlösungen wie Clustermanagementsysteme entschieden vereinfachen kann. Die hier gewählte Einführung in Cloud-Nativity richtet sich im Wesentlichen nach (Garrison und Nova, 2017). Es existieren weitere Definitionen des Begriffs beispielsweise durch die »Cloud Native Computing Foundation«[13].

Um einen einfachen Einstieg in die Thematik zu ermöglichen ist es hilfreich zu erläutern, welche Eigenschaften *nicht* zur Cloud-Nativity einer Anwendung beiträgt. So ist es beispielsweise nicht notwendig, dass eine Anwendung nach der Microservice Architektur aufgebaut ist. Monolithische und auch Microservice Anwendungen können beide das Cloud-Nativity Paradigma umsetzen. Genauso handelt es sich bei Cloud-Nativity nicht um »Infrastructure-as-Code« oder darum Anwendungen in Containern[14] oder auf Containermanagern auszuführen.

Das zentrale Ziel der Cloud-Nativity beschreibt die Ausführbarkeit von Anwendungen in *unbekannten* und *nicht zugreifbaren Systemen*, wie sie insbesondere in Cloud-Umgebungen auftreten. Die Konzepte der Cloud-Nativity sind deutlich allgemeiner gehalten als die der Twelve-Factor App. Die Twelve-Factor App kann daher in gewissen Punkten als eine konkrete Ausprägung einer Cloud-Native-Architektur gesehen werden.

(Garrison und Nova, 2017) beschreiben eine Cloud-Native-Anwendung als eine Anwendung, welche die folgenden vier Charakteristika aufweist:

Abhärtung bezeichnet die Eigenschaft einer Anwendung mit den Unwägbarkeiten innerhalb eines verteilten Systems eigenständig umgehen zu können. Besonderes Augenmerk erhalten hierbei kaskadierende Fehlschläge, welche in jedem Fall unterbunden werden sollten. Ein abgehärtetes System erwartet Fehlschläge der Systeme, von denen es abhängt und versucht nicht sie mit aufwändigen Mitteln zu verhindern. Dieses Design bietet insofern Vorteile, als niemals alle möglichen Fehlergründe für externe Systeme abgefangen werden können. Eine beliebte Implementierung dieses Verhaltens besteht in der Form der bereits erwähnten Anti-Corruption Layern (ACLs).

Ein weiterer Aspekt der Abhärtung besteht darin mit extremen Lastsituationen des Systems umgehen zu können. Ein unerwünschtes Verhalten eines Systems in einem solchen Fall wäre es nicht mehr zu reagieren. Vielmehr soll unter einer bestimmten Last die Bearbeitung von Arbeitsaufträgen soweit heruntergefahren werden, dass diese zwar noch beantwortet werden können, allerdings nicht mehr vollumfänglich, sondern durch eine einfach zu berechnende Antwort. Auf diese Weise kann sichergestellt werden, dass sich das System in Lastsituation weiterhin steuern lässt.

[13] https://www.cncf.io/ - Zuletzt abgerufen am 13.12.2021
[14] Eine genauere Beschreibung von Containern folgt in Abschnitt 2.6.4

Agilität bezeichnet die Eigenschaft der Anwendung in eine Art und Weise entwickelt werden zu können, dass schnelle Anpassungen und schnelle Deployments möglich sind. Einen großen Einfluss haben an dieser Stelle agile Softwareentwicklungsmethoden, sowie der gesamte Ansatz von Continuous Integration und Continuous Deployment, welcher in Abschnitt 2.6.3 noch einmal detaillierter adressiert wird.

Operabilität bezeichnet die Eigenschaft der Anwendung aus sich selbst heraus die Möglichkeit anzubieten administrative Tätigkeiten auszuführen. Klassische Anwendungen wurden gemeinhin von Administratoren in manueller Handarbeit auf lokal verfügbare Server oder aber später auf virtuelle Maschinen installiert. Dort standen dann eine Reihe an externen Werkzeugen und Hilfsmitteln zur Verfügung, um Einfluss auf den gegenwärtigen Zustand der Anwendung zu nehmen. Wird jedoch eine Anwendung speziell für Cloud-Umgebungen konstruiert, so kann bereits im Design der Anwendung davon ausgegangen werden, dass im laufenden Betrieb kein Administrator mehr direkten und unmittelbaren Zugriff auf die ausführende Systemumgebung hat. Folglich müssen die immer noch notwendigen Befehle zur Administration direkt in die Anwendung integriert werden. Dies hat darüber hinaus den Vorteil, dass die Anwendung durch das Design der Befehle durch die Anwendungsentwickler sehr viel präziser und genauer kontrolliert werden kann als es zuvor mit standardisierten Werkzeugen der Fall war. Jedoch sollte dabei in jedem Fall angemerkt werden, dass dies einen deutlichen zusätzlichen Programmieraufwand mit sich bringt.

Beobachtbarkeit stellt die letzte betrachtete Eigenschaft einer Cloud-Native-Anwendung dar. In Analogie zur Operabilität umfasst die Beobachtbarkeit alle auf die Anwendung bezogenen lesenden Tätigkeiten. Ein Kriterium, welches besonders oft von erhöhtem Interesse ist, ist die Information darüber, ob die Anwendung in einem korrekten Zustand und produktiv nutzbar ist. Auch hier waren Administratoren bislang darauf angewiesen sich mit externen Tools ein Bild über den aktuellen Zustand der Anwendung zu machen. Jedoch ist durch die Implementation von Beobachtungsklassen eine viel genauere Aussage über den Zustand der Anwendung möglich.

Reconciliation

Reconciliation (dt.: Abgleich; Versöhnung) bzw. das *Reconciler-Pattern* (dt.: Versöhnermuster) ist ein Muster, welches häufig in verteilten Systemen und bei der Umsetzung von (automatisierten) Skalierungsoperationen eingesetzt wird. Eine Einführung gibt beispielsweise ebenfalls (Garrison und Nova, 2017).

Die Grundidee dabei umfasst den Einsatz einer speziellen Softwarekomponente, eines sogenannten *Reconcilers*. Der Reconciler kann den Ist-Zustand eines Systems an einen angegebenen Soll-Zustand angleichen. Das Arbeiten mit Aufträgen für gewünschte Ist-Zustände anstatt expliziter Änderungsaufträge bewirkt den Einsatz idempotenter[15] Operationen. Insbesondere in Systemen mit unsicherer Kommunikation und potentiellem Datenverlust besteht hierdurch die Möglichkeit die gewünschte Anfrage mehrfach abzusenden, ohne dabei ein ungewünschtes Systemverhalten zu erzeugen, falls doch mehr als eine der Anfragen letztendlich ausgeführt werden sollten.

2.6.3. Softwareentwicklungsmodalitäten und Technologien

Um die vollständige Motivation hinter dem Wandel zu Cloud-nativen Anwendung nachvollziehen zu können, ist es notwendig den Wandel der Modalitäten der Softwareentwicklung zu betrachten und den Einsatz der damit einhergehenden Technologien. Substantielle finanzielle Mittel wurden über Dekaden in Software investiert, welche nicht den Wünschen und Bedürfnissen der intendierten Anwender entsprachen. Das Einholen von schnellem Feedback gewann zusehends an Bedeutung. Das Ausliefern von Software war bereits historisch ein komplexer Prozess und wird heute durch den Einsatz verschiedener Technologien und massiv gestiegenen Anforderungen und Erwartungshaltungen weiter verkompliziert. Folglich war es notwendig, dass sich neue Formen des Vertriebs von Software etablieren. Auf Basis dessen werden in diesem Abschnitt einige der Technologien und Modalitäten der Entwicklung vorgestellt, welche die direkte Grundlage für verschiedene Teile der Arbeit bilden.

Vom Wasserfallmodell zur Agilität

Seit Anfang der 2000er durchlebt die (kommerzielle) Softwareentwicklung eine Trendwende von geplanten Projekten mit langer Entwicklungszeit, seltenen Auslieferungen und weitestgehender Abkoppelung der Softwareentwickler von Kunden hin zu neueren *agilen* Arbeitsformen. Die agilen Arbeitsmethoden sind vielfältig und eine genaue Erörterung wäre für den Rahmen dieser Arbeit wenig zielführend. Die Gemeinsamkeit der agilen Methoden ergibt sich jedoch durch eine enge Integration von Entwicklern und Kunden, ständigem Anpassen der Ziele ohne langfristige Detailplanung und häufigen Auslieferungen (*Deployments*) der Software.

[15] Eine Operation ist idempotent, falls eine beliebige Wiederholung der Operation den identischen Effekt wie die einmalige Ausführung hat.

Von Dev und Ops zu DevOps

Traditionell ist das Ausliefern von Software ein aufwändiger Prozess, bei dem Softwareentwickler und Systemadministratoren Tage oder gar Wochen beschäftigt waren. Zur Beschreibung dieses Verfahrens ist eine Definition des Begriffs »Deployment«, wie er im Rahmen der Arbeit verstanden werden soll, hilfreich:

Definition 2.17 (Deployment)**.** Deployment bezeichnet das Verfügbarmachen eines bestimmten Standes an Software in einer für die Software geeigneten Umgebung. Es ist Teil vom Entwicklungsprozess.

Im Zeichen des agilen Arbeitens, bei dem häufige Deployments gewünscht sind, um möglichst schnell Feedback vom Kunden einzuholen, liegt die Gefahr jedoch nahe, dass der Overhead für Deployments einen großen Anteil der Zeit vereinnahmt oder die Administration vor gänzlich unlösbare Probleme stellt. Aus dieser Problematik ist der Berufszweig des »DevOps-Engineers« entstanden. Bei »DevOps« handelt es sich um ein Kunstwort aus »Developer« (Entwickler) und »Operator« (Administrator). Die Aufgabe des DevOps-Engineers besteht darin das Deployment mit Programmierung nach Möglichkeit komplett zu automatisieren.

Continuous Integration

Die Automatisierung des Deployments wird gemeinhin als *Pipeline* bezeichnet. Der Grad der Automatisierung des Deployments kann durch die Begriffe *Continuous Integration*, *Continuous Delivery* und *Continuous Deployment* charakterisiert werden. Bei der *Continuous Integration* (CI) wird lediglich ein Kompiliervorgang und Unit/Integrations-Test auf einem unabhängigen System angestrebt, um den Entwicklern möglichst früh Feedback ermöglichen zu können. Die *Continuous Delivery* bildet ein Bereitstellen des kompletten Deployments bis kurz vor die Produktionsumgebung, welches dann manuell im Idealfall mit einem einzelnen Klick erfolgen kann. Fällt der finale manuelle Klick ebenfalls weg, wird von *Continuous Deployment* gesprochen. Die Abkürzung »CD« kann dabei für beide Ansätze - *Continuous Delivery* und *Continuous Deployment* stehen. Insgesamt wird oft von »CI/CD« gesprochen.

In den Anfängen der CI wurden im Deploymentprozess viele Aufgaben manuell auf den Zielsystemen ausgeführt, um etwaige Konfigurationen vorzunehmen und das Zielsystem entsprechend vorzubereiten. Diese Vorgehensweise ist mühsam und fehleranfällig. Läuft die Pipeline auf ein Problem, muss erneut manuell eingegriffen werden. Lösungen in diesem Bereich werden im separaten Abschnitt 2.6.4 erläutert.

Heutzutage kann auf ein reichhaltiges Angebot an Software, Bibliotheken und (Teil)lösungen zurückgegriffen werden. Es ist nicht mehr erforderlich für viele Technologiekomponenten eigene Implementationen umzusetzen. Einige dieser Komponenten bieten eine elementare Grundlage für Prototypen, welche im Rahmen dieser Arbeit entstanden sind.

2.6.4. Deploymentformen

In Abschnitt 2.6.3 wurde bereits der Begriff *Deployment* definiert, wie er im Kontext der Arbeit verstanden werden soll. Neben der abstrakten Beschreibung ist es hilfreich vier gängige Deploymentformen kennenzulernen. Das Augenmerk liegt dabei auf der Art und Weise wie und in welchem Format das Deployment erfolgt und weniger auf dem zugehörigen Prozess, da dieser bereits im Rahmen der Continuous Integration am Ende von Abschnitt 2.6.3 erörtert wurde.

Die einfachste Form umfasst das *vollständig manuelle* Deployment. Diese Form war lange Zeit vorherrschend, benötigt kaum Werkzeugunterstützung, besitzt jedoch auch den mit Abstand größten manuellen Aufwand. Vorbereitung jedes Systems durch Installationen, Laufzeitbibliotheken, etc. erfolgt händisch durch Administratoren, ebenso wie die Installation der Software selbst.

Autokonfiguration

Insbesondere bei der Konfiguration verschiedener Rechnersysteme kann ein manuelles Deployment unwirtschaftlich sein. Viele der Aufgaben sind automatisierbar, sodass bald Autokonfigurations-Tools bzw. Konfigurations-Management Tools entwickelt wurden. Diese haben die Aufgabe(n) automatisiert bestimmte Konfigurationszustände auf vielen physikalischen Clients herzustellen, nach Möglichkeit ohne manuellen Aufwand. Auch die massenhafte Fernwartung und Ausführung von Programmen sind häufig anzutreffende Features von Autokonfigurations-Tools.

Bekannte Vertreter sind beispielsweise die Projekte *puppet*[16], *Ansible*[17] und *Saltstack*[18].

Die grundlegende Autokonfiguration steht nicht zwangsläufig im Konflikt mit den anderen vorgestellten Deploymentformen. Insbesondere mit virtuellen Maschinen existieren Integrationen wie beispielsweise durch das Projekt bzw. Deploymenttool *Vagrant*[19].

[16] https://puppet.com/ - Zuletzt abgerufen am 14.12.2021
[17] https://www.ansible.com/ - Zuletzt abgerufen am 14.12.2021
[18] https://saltproject.io/ - Zuletzt abgerufen am 14.12.2021
[19] https://www.vagrantup.com/ - Zuletzt abgerufen am 14.12.2021

Insbesondere aber der klassische direkte Einsatz auf physikalischen Maschinen (»Bare-Metal«) ist als Konkurrenzentwurf zu den anderen Deploymentformen zu sehen. Analog zum manuellen Deployment besteht beim Einsatz klassischer Server die Problematik nur schwer Zwischenzustände des Systems sichern zu können und ebenfalls die installierte Maschine in ihrer Gesamtheit auf eine andere Hardware umziehen zu können. Autokonfiguration auf klassischen Servern umfasst daher als Nachteil insbesondere eine reduzierte Portabilität.

Virtualisierung

Durch die zuvor geschilderte Problematik mit klassischen Servern entwickelte sich ein Wandel in Richtung der Verwendung virtueller Maschinen (nachfolgend *VMs*) in produktiven Umgebungen. Virtuelle Maschinen weisen den Vorteil der Kapselung und den der Portabilität auf. Eine virtuelle Maschine beinhaltet ein vollständiges virtuelles Betriebssystem und erscheint für die darin deployte Software wie ein eigenständiges System mit eigener Hardware.

Auf einer physikalischen Maschine können viele virtuelle Maschinen im Parallelbetrieb ausgeführt werden. Während VMs die oben genanten Vorteile bieten, so muss doch für jedes Deployment eine komplette VM gestartet werden. Der benötigte Speicher für virtuelle Betriebssystemkomponenten ist somit als (nicht kleiner) Overhead zu sehen. Darüber hinaus sind virtuelle Maschinen zwar portabel aber dennoch nicht klein, sodass ein Transfer abhängig von der Datentransferrate des zu Grunde liegenden Netzwerkes merkliche Zeiträume bis hin zu Minuten oder gar Stunden vereinnahmt. Die Übertragung auf einen entfernten Rechnerknoten ist somit nicht unmöglich, aber im Regelfall auch nicht in Millisekunden realisierbar.

Nichtsdestotrotz war der Wandel zur Virtualisierung sehr gewinnbringend. Anstatt dass stets ganze Server ausgetauscht, neu aufgesetzt und im Ernstfall wiederhergestellt werden mussten, bietet die Virtualisierung die Möglichkeiten der Portabilität des gesamten Systems sowie die Fähigkeit auf einfache Weise das gesamte virtuelle System zu einem zuvor gespeicherten Zustand zurückzusetzen. Darüber hinaus können ganze Systeme auf einen anderen Rechnerknoten automatisiert verlagert werden.

Bekannte Virtualisierungslösungen und -anbieter sind *VirtualBox*[20], *VMware*[21] sowie *Hyper-V*[22].

[20] https://www.virtualbox.org/ - Zuletzt abgerufen am 14.12.2021
[21] https://www.vmware.com - Zuletzt abgerufen am 14.12.2021
[22] https://docs.microsoft.com/de-de/virtualization/hyper-v-on-windows/ - Zuletzt abgerufen am 14.12.2021

Containerisierung

Um eine schnelle Anpassung an wechselnde Anforderungen und insbesondere als Reaktion auf Lastspitzen zu gewährleisten, genügte die Virtualisierung alleine jedoch nicht. Zur Illustration des Problems soll an dieser Stelle ein Beispiel eines Online-Shops dienen. Für gewöhnlich surfen etwa 500 Kunden auf den Seiten des Shops. Die Betreiber haben vorgesorgt und das System bereits mit einem Load Balancer und zwei virtuellen Servern ausgestattet. Im Normalbetrieb können alle Kundenanfragen selbst in Stoßzeiten problemlos bedient werden, indem täglich vor der Stoßzeit ein weiterer virtueller Server automatisiert an den Load Balancer angebunden wird.

Nun findet allerdings eine öffentliche Persönlichkeit einen Artikel in dem Shop, der ihr sehr gefällt. Sie teilt den Fund in einem sozialen Netzwerk und kurz darauf rufen zehntausende Besucher den Shop auf, um besagtes Produkt zu betrachten und ggf. zu bestellen. Das Ergebnis liegt auf der Hand: Der Shop ist stark überlastet und kaum einer der potentiellen Kunden kann noch ordentlich bedient werden. Der Administrator muss – sofern er das System gerade überwacht – schnell reagieren, weitere Server außerhalb des Normalbetriebs anmieten und die virtuelle Umgebung so schnell wie möglich dorthin transferieren. Wurde der Fall von vorn herein bedacht, dauern Übertragung und Start dennoch unter Umständen 20 Minuten. Diese Zeit genügt jedoch leider, damit viele potentielle Kunden frustriert das Interesse verlieren. Von vorn herein genug Server vorzuhalten ist jedoch auch keine Option, da dies für den Normalbetrieb überdimensioniert und nicht wirtschaftlich wäre.

An dieser Stelle wäre eine automatische und sekundenschnelle Skalierung der Serverlandschaft von immensem Nutzen gewesen. Um genau an dieser Stelle anzusetzen erlebte die Technologie der Containerisierung in den 2010er Jahren einen gewaltigen Aufschwung. Containerisierung ist aus praktischen und operativen Gesichtspunkten ähnlich zu Virtualisierung, die darunterliegende Technologie ist jedoch grundverschieden. Während VMs vollwertige Betriebssysteme emulieren, betten sich Container als Prozess im Hostsystem ein und erreichen eine Trennung durch weitgehende Prozessisolation. Aus dem Inneren eines Containers ist die Umgebung nicht von der auf einer VM oder eines physikalischen Servers zu unterscheiden.

Alle Container auf einer physikalischen Maschine teilen sich die Betriebssystemkomponenten, anstatt wie in einer virtuellen Maschine diese selbst individuell bereitzustellen. Dazu muss lediglich das für den Container Wesentliche mitgeliefert werden, häufig wenige Megabyte. Zusätzlich entfällt sämtliches Bootstrapping der Systemkomponenten, wodurch Container in vielen Fällen automatisiert im Millisekundenbereich bereitgestellt werden können.

Container besitzen durch diese Architektur meist eine nicht-schreibbare Umgebung und können meist nur Daten im Hauptspeicher sichern. Persistente Datenhaltung bedarf daher in Containern immer einer besonderen Behandlung. Ebenfalls ist das Erzeugen von Sicherungsständen im Gegensatz zu VM-basierten Lösungen wesentlich weniger leicht umsetzbar.

Bekannte Vertreter von Containertechnologie sind beispielsweise *LXC/LXD*[23], *Docker*[24] oder auch *RKT*[25].

2.7. Szenenanalyse

In der Einleitung wurde die Szenenanalyse als Werkzeug motiviert, um in einem gewissen Umfang Daten aus der realen Welt einzufangen, um daraus wiederum ein Modell generieren zu können. Gerade bei großen Datenmengen kann der Einsatz automatisierter Extraktion von Zusammenhängen hilfreich sein, um entsprechende Agenten zu konstruieren. Während diese Arbeit diesen Aspekt nur motiviert, jedoch nicht holistisch darlegt, sondern lediglich eine Verbesserung für einen Teilbereich liefert, liegt es dennoch nahe, eine grundlegende Einführung zu der Thematik zu geben. Dabei wird der Fokus stark auf die später eingesetzten Konzepte eingeschränkt und eine Vielzahl weiterer Konzepte entfallen, welche sonst im globalen Kontext der Szenenanalyse ohne Ausrichtung auf diese Arbeit erwähnenswert wären.

Die Interpretation von Szenen in der realen Welt ist selten ein einfaches Unterfangen. Dabei ist die Verarbeitung vieler Bilddaten eine zentrale Herausforderung. Die Szeneninterpretation basiert im Wesentlichen auf Beobachtungen, Kontextinformationen, daraus abgeleiteten Konzepten und wiederum daraus abgeleiteten Hypothesen. Die Anbindung an agentenorientierte Systeme ist ein naheliegender Schritt, da Agenten durch ihre Wissensbasen eine natürliche Repräsentationsmöglichkeit liefern. Für Beobachtungen werden meist andere Teilbereiche der Bildverarbeitung eingesetzt, die auf niedrigerer Ebene ansetzen, wie beispielsweise Segmentierungsverfahren, Objekterkennung und Verwandtes.

2.7.1. Grundbegriffe

In der englischsprachigen Literatur finden sich viele Oberbegriffe, wie »Image Processing«, »Computer Vision«, »Pattern Recognition«, »Image Understanding«,

[23] https://linuxcontainers.org/ - Zuletzt abgerufen am 14.12.2021
[24] https://www.docker.com/ - Zuletzt abgerufen am 14.12.2021
[25] https://cloud.redhat.com/learn/topics/rkt - Zuletzt abgerufen am 14.12.2021

»Video Understanding« und weitere. Für ein eindeutiges Verständnis sollen diese nun kurz abgegrenzt werden.

Definition 2.18 (Image Processing; dt.: Bildverarbeitung)**.** Die Bildverarbeitung im weitesten Sinne umfasst die Manipulation und einfachen Analyse von Bilddaten. Übliche Ziele sind beispielsweise Segmentierung, Kantendetektion, Farbkalibrierung, Verfremdung, Unkenntlichmachung, Wiederherstellung und viele weitere. Insbesondere verwenden viele Bildverarbeitungsalgorithmen Bilder sowohl als Eingabe als auch als Ausgabe.

Definition 2.19 (Pattern Recognition; dt.: Mustererkennung)**.** Die Mustererkennung ist ein vergleichsweise grundlegender Bereich, bei dem der Fokus auf der Rekonstruktion von Mustern aus Datenquellen liegt. Eine häufige Verwendung findet sich in der Objekterkennung und Strukturerkennung in Bildern, Mustererkennung kann jedoch auch auf anderen Medien wie geschriebenen Texten erfolgen. Im Bereich der Mehrbildanalyse und insbesondere der Stereoskopie und bei der Betrachtung des optischen Flusses beschreibt das »Korrespondenzproblem« die Frage, welche Bildmerkmale in einem Bild welchen Objektmerkmalen in einem anderen Bild entsprechen.

Definition 2.20 (Computer Vision; dt.: Computer Sehen)**.** Computer Vision umfasst den gesamten Bereich des maschinellen »Sehens«. »Sehen« bedeutet hierbei die Verarbeitung von Bilddaten in Analogie zur Verarbeitung durch das menschliche Gehirn. In Analogie zum biologischen Vorbild kommt hierbei häufig künstliche Intelligenz zum Einsatz sowie Wissensdatenbanken jedweder Form. Trotz beeindruckender Fortschritte ist die internationale Forschergemeinschaft weit davon entfernt ein maschinelles System entwickelt zu haben, welches der Universalität des menschlichen Sehens nahe kommt.

Definition 2.21 (Image Understanding; dt.: Bildverstehen)**.** Der Bereich des Bildverstehens befasst sich mit der Ableitung höherer Konzepte und Informationen aus Bildern. Da dies eine sehr menschliche Betrachtungsweise der Informationen in Bildern ist, liegt es Nahe, dass sich viele Arbeiten in dem Bereich an Konzepten des menschlichen Sehens orientieren. Häufig wird explizites Kontextwissen verwendet oder benötigt, um Aussagen in dieser Form treffen zu können. Bildverstehen ist ein Teilbereich von Computer Vision.

Definition 2.22 (Video Understanding; dt.: Videoverstehen)**.** Videoverstehen und Bildverstehen sind eng verwandt, jedoch fokussiert sich das Videoverstehen auf Videos bzw. Bilderfolgen. Durch die zusätzliche zeitliche Dimension kann auch von einer Analyse im Dreidimensionalen gesprochen werden. Analog zum Bildverstehen ist Videoverstehen ein Teilbereich von Computer Vision.

In klassischen Ansätzen kamen meist deterministische Verfahren zum Einsatz. Dabei war häufig ein zentrales Element die Transformation von *Features* in eigene *Feature*-Räume. Da speziell dieser Ansatz später innerhalb der Arbeit relevant wird, wird er kurz vorgestellt bzw. definiert.

Dazu soll die Definition eines Bildes aus (Seppke, 2013) herangezogen werden:

> **Definition 2.23** (Bildfunktion)**.** Ein Bild $I(x, y)$ der Breite $w \in \mathbb{N}$ und Höhe $h \in \mathbb{N}$ mit n Bändern wird durch folgende Abbildung beschrieben:
>
> $$I : [0, w[\times[0, h[\rightarrow \mathbb{R}^n \tag{2.1}$$
>
> Analog ergibt sich für ein digitales Bild $I(x, y)$:
>
> $$I : \{0, ..., w-1\} \times \{0, ..., h-1\} \rightarrow R^n \subset \mathbb{R}^n \tag{2.2}$$
>
> — (Seppke, 2013, Seite 33)

Ein Band wird gelegentlich auch als *Kanal* bezeichnet. Dieser Begriff ist im Kontext der Arbeit jedoch syntaktisch nah an den *synchronen Kanälen* im Kontext von Referenznetzen, sodass auf die Verwendung des Begriffs *Kanal* im Kontext von Bildern verzichtet werden soll.

2.7.2. Features

Die Nutzung von Features wird seit langem in der Forschung beschrieben. Erste Arbeiten zu dem Thema sind (Moravec, 1981) und (Harris und Stephens, 1988). (Dreschler, 1981) nutzte Features zur Rekonstruktion von geometrischen Objekten in Bildfolgen.

Features sind im weitesten Sinne »interessante« bzw. markante Punkte innerhalb eines Bildes. Sie zeichnen sich meist durch einen hohen Informationsgehalt aus und können beispielsweise Kanten oder Ecken darstellen, sind jedoch nicht auf diese beiden Kategorien limitiert. Features werden aus Bilddaten durch Algorithmen extrahiert. Ein solcher Algorithmus soll als *Feature-Algorithmus* bezeichnet werden.

Die Berechnung oder Bestimmung eines für eine Problemstellung geeigneten Feature-Algorithmus ist nicht trivial und Ergebnis verschiedener wissenschaftlicher Ergebnisse. Verschiedene Umsetzungen von Feature-Algorithmen bezwecken möglichst markante Punkte im Bild zu identifizieren, wie beispielsweise Ecken, welche sich häufig durch schnelle Hell/Dunkel Wechsel in fast allen Richtungen auszeichnen. Später wird mit »SIFT« ein wichtiger Vertreter einer solchen Berechnung vorgestellt werden. Analog zu Feature-Punkten existieren Feature-Kanten und

Feature-Objekte, die jedoch nicht im Fokus der Arbeit stehen und nur der Vollständigkeit halber Erwähnung finden sollen. Im weiteren Verlauf der Arbeit wird sich der Begriff »Feature« stets auf Feature-Punkte beziehen.

Da die alleinige Lokalisation eines Features meist keine großen Vorteile bietet, ist es notwendig zu jedem Feature einen Datensatz zu speichern, welcher eben dieses Feature und ggf. die Umgebung genau beschreibt. Diese Beschreibung wird meist als *Deskriptor* bezeichnet.

Zu diesem Zweck wird meist eine Reihe an gleichartigen Daten gesichert, die sich als Vektor darstellen lassen. Je nach Format existieren verschiedene Datenformate wie Ganzzahlen, Gleitkommazahlen sowie Anzahlen an Datensätzen. Die berechneten und gesicherten Deskriptoren ergeben sodann Punkte im durch die Vektoren aufgespannten sogenannten *Feature-Space*.

Wird nun eine Objekterkennung angestrebt ist, ein häufiger Ansatz die Feature-Daten vieler Bilder bezüglich eines festen Feature-Algorithmus für einen Abgleich zu verwenden. In einem Eingabebild werden die Features im Feature-Space ermittelt und die Deskriptoren mittels einer Nearest-Neighbour-Suche mit dem Datenbestand abgeglichen. Die Suche im Feature-Space reduziert den Abgleich auf ein einfaches Nachbarschaftssuche-Problem.

Durch Übereinstimmungen können ähnliche Bilder schnell gefunden werden, wobei es sich um eine Grundaufgabe der Mustererkennung handelt. Die Qualität dieser »Ähnlichkeit« basiert maßgeblich auf dem verwendeten Feature-Algorithmus. Das Erreichen einer hohen Qualität ist meist vorrangiges Ziel innerhalb dieser Forschungsbestrebungen.

2.8. Technik: Renew - Reference Net Workshop

Der fortgeschrittenste Simulator im Bereich der Referenznetz ist gegenwärtig das Werkzeug Renew. Damit sollte dieser Abschnitt auf den ersten Blick erst im Abschnitt zur Realisierung Platz finden. Einige Konzepte, auf welche sich auch der konzeptuelle Teil der Arbeit beruft, wurden von den zitierten Arbeiten jedoch direkt im Kontext bzw. am Beispiel von Renew eingeführt und diskutiert. Aus diesem Grund soll die Einführung in Renew schon an dieser Stelle erfolgen, um den entsprechenden Kontext bereitzustellen.

Renew entstand im Rahmen von Olaf *Kummer*s Dissertation zu Referenznetzen (Kummer, 2002) und wird seitdem an der Universität Hamburg gewartet. Renew ist ein Akronym für **Re**ferenz**ne**tz**w**erkzeug, oder englisch **Re**ference **Ne**t **W**orkshop und setzt den gesamten Referenznetzformalismus mit Konkretisierung durch die Programmiersprache Java um. Über die Zeit wurden diverse Verbesse-

rungen und Erweiterungen entwickelt, wie eine Plugin-Architektur, das Agenten-Framework Mulan und viele weitere. Dieses Kapitel gibt eine Übersicht zu den für die Arbeit relevanten Eigenschaften von Renew. Renew genießt hier besonderes Augenmerk, da es für die Implementation der Konzepte zur verteilten Referenznetzsimulation die Rahmenumgebung darstellen wird.

2.8.1. Funktionsumfang

Der Kernbestandteil vom Renew Simulator (kurz Renew) umfasst die Simulation verschiedener Typen von Petrinetzen. Da Petrinetze allerdings eine grafische Beschreibungssprache von nebenläufigen Systemen sind, liegt es nahe, dass Renew über einen grafischen Editor zum Erzeugen von Petrinetzen verfügt. Darüber hinaus existieren Funktionalitäten, um die Kompatibilität zu anderen Petrinetzlösungen sicherzustellen. Mit Renew erzeugte Netze können in verschiedener Weise und zu verschiedenen Zwecken im- und exportiert werden. Im Folgenden wird eine kurze Einführung in die von Renew unterstützten Formalismen, die Oberfläche und den Einsatz als Entwicklungsumgebung gegeben.

Formalismen

Wie bereits erwähnt und aus dem Namen Renew ersichtlich, ist der wichtigste Formalismus, der durch Renew umgesetzt wird, der Referenznetzformalismus. Mithilfe des Referenznetzformalismus können auch niedrigere Petrinetze, wie beispielsweise gefärbte Netze oder auch kantenkonstante gefärbte Netze abgebildet werden. Da für einfache P/T-Netze die Syntax des Referenznetzformalismusses, wie er in Renew umgesetzt ist, zu aufwändig erscheint, unterstützt Renew einen dedizierten P/T-Netz Formalismus. Dabei können in den Plätzen beispielsweise einfach Ganzzahlwerte eingesetzt werden, um die in dem Platz enthaltene Anzahl an Marken anzugeben. Darüber hinaus existieren in der Basisversion von Renew zusätzlich noch die Timed Java Netze, sowie boolsche Netze. Da diese beiden Netztypen keine weitere Bedeutung für den Inhalt dieser Arbeit haben, sei an dieser Stelle auf eine weiterführende Beschreibung verzichtet. Da Renew wie eingehend erläutert eine Plugin-Architektur besitzt, existieren jenseits der Basisversion von Renew eine Vielzahl von Plugins, die weitere Formalismen hinzufügen können. Dabei spielt ein Plugin im Kontext dieser Arbeit eine zentrale Rolle: das Distribute-Plugin mit dem Distribute-Formalismus (M. Simon und Moldt, 2016). Aus diesem Grund wird eben dieses Plugin im separaten Abschnitt 2.8.8 vorgestellt.

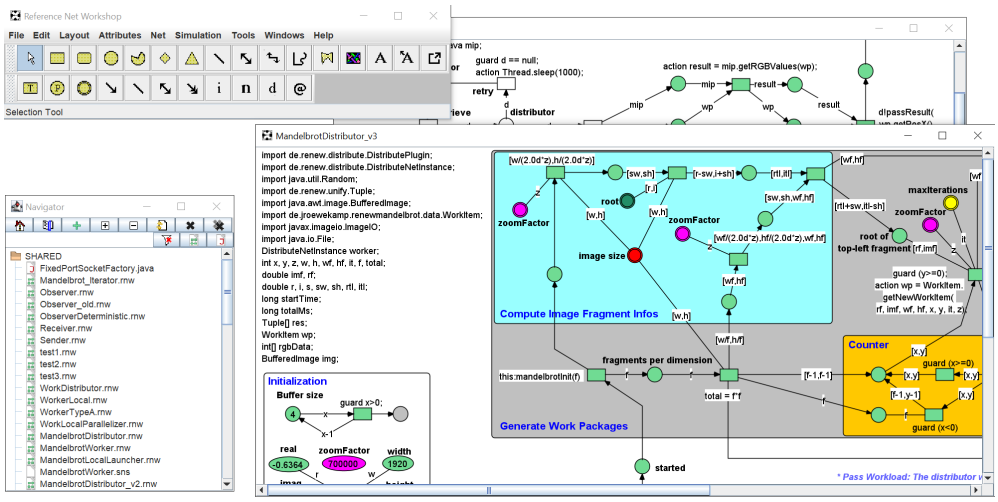

Abbildung 2.7.: Die klassische Oberfläche von Renew

Oberfläche

Die Oberfläche von Renew basiert auf der Bibliothek JHotDraw. In einem Hauptfenster sind Tools für die grafische Gestaltung von Petrinetzen und simplen Formen verfügbar sowie ein Menü zur Einstellung und einer Ansteuerung der verschiedenen Funktionalitäten der Anwendung. Jedes statische Netz wird durch Renew in einem eigenen Fenster dargestellt. Durch Auswahl des geeigneten Grafiktools kann das Aussehen des Netzes manipuliert werden. Die Oberfläche selbst ist relativ eng an die sonstigen Bestandteile von Renew gekoppelt, jedoch gab es in der neueren Vergangenheit zahlreiche Bestrebungen diese Kopplung möglichst lose zu gestalten. Neben der klassischen Java Oberfläche existieren weitere Oberflächen, die neue Funktionalitäten wie beispielsweise einen Zoom hinzufügen (Wincierz, 2018). Darüber hinaus sind auch Oberflächen in der Entwicklung, die auf gänzlich anderen Technologien als Java aufbauen. renew-js versucht die Renew Funktionalität in einer Web Anwendung basierend auf JavaScript darzustellen.

Abbildung 2.7 zeigt einen beispielhaften Ausschnitt der klassischen (Java) Oberfläche von Renew 2.6 unter Windows 10.

Entwicklungsumgebung

Da die Anschriftensprache des in Renew umgesetzten Referenznetzformalismus auf Java basiert, ist es möglich in Form von Referenznetzen mit Renew Java Programme zu entwerfen. Daher kann Renew selbst ebenfalls als Entwicklungs-

umgebung gesehen werden. Speziell mit einigen Erweiterungen bietet Renew dem Entwickler die Möglichkeit mit wenigen Klicks komplexe Netzkomponenten zu erzeugen (Cabac, 2010). Dabei kann das entworfene Referenznetzsystem direkt in Renew ausgeführt, getestet und evaluiert werden. In diesem Kontext ist die Transitionsanschrift »manual« von Bedeutung. Eine Transition, die diese Anschrift trägt, kann nur durch einen Nutzer mit einem Klick auf die Transitionen gefeuert werden, auch, wenn die Transitionen eigentlich aktiviert ist und von der Simulation gefeuert werden könnte.

2.8.2. Anschriftensprache

Wie bereits im vorherigen Abschnitt zur allgemeinen Einführung von Referenznetzen angesprochen, wird im Kontext des Renew Simulators eine ausformulierte Anschriftensprache für Referenznetze definiert. Dies bezieht sich sowohl auf die Anschriften an Transitionen und Kanten als auch auf die Arten und Weisen wie Marken spezifiziert werden können. Die Inhalte in diesem Abschnitt orientieren sich im Wesentlichen an dem Renew User Guide (Kummer, Wienberg, Duvigneau, Cabac u. a., 2016).

Im Kern sind Referenznetze gefärbte Petrinetze. Daher können Marken verschiedene Farben bzw. Typen besitzen. Im Hintergrund werden alle Formen von Marken als Werte behandelt, spezieller formuliert sind Marken in Referenznetzen immer Tupel. Somit kann eine Marke in einem Referenznetz entweder eine Referenz oder ein primitiver Typ sein. Schwarze Marken sind null-elementige Tupel. Sie werden beispielsweise zur Modellierung von Abläufen eingesetzt und mit »[]« notiert. Des Weiteren können Marken spezifiziert werden, indem einigen Hilfsmitteln der Sprache Java um Literale auszudrücken verwendet werden. Da Renew auf JavaCC zum Parsen der Netze aufbaut, sind jedoch nicht alle Literale unterstützt, sondern nur solche, die auch schon in frühen Java Versionen existiert haben.

Die hauptsächliche Komplexität der Anschriften findet sich speziell in den Transitionsanschriften. Hier können im Wesentlichen beliebige und komplexe Java Ausdrücke Verwendung finden. So können beispielsweise logische Operatoren wie und (&&) und oder (||) zum Einsatz kommen, Rechenoperationen wie +, -, usw. und Zuweisungen eingesetzt werden.

Da im Kontext von gefärbten Netzen und speziell den Referenznetzen noch weitere Anforderungen existieren, ist die Menge an möglichen Anschriften durch einige Besonderheiten erweitert. Ein Beispiel mit diesen speziellen Anschriften findet sich im Abschnitt 2.8.3. Da einige dieser Anschriften im Kontext von skalierenden Referenznetzsimulationen von besonderer Wichtigkeit sind, werden diese auszugsweise hier dargestellt:

Synchrone Kanäle Eine der Besonderheiten der Anschriftensprache ist die Art und Weise, wie verwendete synchrone Kanäle definiert werden. Die Verwendung von synchronen Kanälen beinhaltet stets den Einsatz eines **Uplinks** und eines **Downlinks**. Darüber hinaus können an einzelnen Transitionen mehrfache Downlinks, jedoch stets nur ein Uplink vorhanden sein. Dadurch lassen sich auch komplexe Synchronisationsbeziehungen ausdrücken. Die Limitation auf einen Uplink begründet sich in den komplexitätstheoretischen Eigenschaften des zu Grunde liegenden polynomiellen Simulationsalgorithmus und kann in (Kummer, 2002) nachgeschlagen werden.

Bei einem Uplink wird stets ein Doppelpunkt (:) gefolgt von dem eindeutigen Namen des synchronen Kanals verwendet. Der Uplink wartet darauf, dass er durch einen gleichnamigen Downlink aufgerufen wird. Jeder Downlink bezieht sich stets auf die Referenz auf eine weitere Netzinstanz, in welcher der zugehörige Uplink zu finden ist. Downlinks werden daher mit einer Netzinstanz-Variable gefolgt von einem Doppelpunkt und dem Namen des synchronen Kanals notiert.

Verwendet der Kanal Parameter, so müssen diese beim Feuern der Transition unifiziert werden. Durch die Unifikation kann Informationsaustausch durch den synchronen Kanal stattfinden und die zugehörigen Transitionen können nur feuern, wenn eine Unifikation möglich ist. Dabei ist insbesondere ein wechselseitiger Austausch von Informationen zwischen Uplink und Downlink möglich.

Umgang mit Seiteneffekten Die »action« Anschrift ist erwähnenswert, da sie fundamental anders als die sonstigen Anweisungen von Renew gehandhabt wird. Dazu ist es nötig zunächst das Konzept der Bindungssuche zu betrachten. Diese wird ausgeführt, bevor eine Transition schalten kann, um aktivierte Transitionen zu finden. Dabei müssen etwaige Ausdrücke an den Transitionen bereits ausgewertet werden, um möglichst genaue Aussagen über Möglichkeiten zur Unifikation von verwendeten Variablen treffen zu können. Während normale Anschriften im Zuge der Bindungssuche bereits ausgewertet werden, und dabei etwaige Seiteneffekte ebenfalls ausgeführt werden, wird die Ausführung von action-Anschriften bis zum Feuern der betreffenden Transitionen zurückgehalten. Dieses Verhalten ähnelt der Pivot-Transaktion im Saga-Pattern.

Dies hat insbesondere auf die Art und Weise Auswirkungen, wie diese Form von Anschriften in der Simulation behandelt werden muss. Die »action« Anschrift sollte in allen Fällen Verwendung finden, bei denen es nicht gewünscht ist, dass ein Ausdruck nur maximal einmal ausgeführt wird. Dadurch entsteht ebenfalls der Zusammenhang, dass die mögliche Bindung erst zum Zeitpunkt des Feuerns ausgewertet werden kann.

Ein Realweltbeispiel für einen praktischen Einsatz wäre die Belastung der Kreditkarte eines Kunden mithilfe eines Drittanbieters.

Guardprädikate Referenznetze basieren im Speziellen auch auf dem Formalismus der gefärbten Netze. Daher unterstützen sie Guardprädikate als Methodik um unterschiedliches Schaltverhalten je nach Markentyp umsetzen zu können. Der übliche Weg besteht darin sogenannte »guards« zu verwenden, welche anhand eines Prädikats prüfen, ob mit der gegebenen Markierung ein Feuern der Transition möglich ist. Syntaktisch wird das Schlüsselwort »guard« verwendet, gefolgt von einem Java Ausdruck, welcher letzten Endes einen booleschen Wert liefert.

Netzinstanzerzeugung Ein Bezeichner für die zu erzeugende Referenz gefolgt von einem Doppelpunkt, dem Keyword »new« und dem Namen eines statischen Netzes kann eingesetzt werden, um eine neue Netzinstanz zu erzeugen. Die Syntax ist dabei ähnlich zu der eines Downlinks, akzeptiert außer dem statischen Netz keine Parameter, verzichtet jedoch auf Klammern. Ein Beispiel für die Syntax lautet: `n:new example_net`

Tupel Neben einfachen Ausdrücken können mit eckigen Klammern und Kommata getrennt Tupel angegeben werden. Ein Tupel ist dabei eine Ansammlung mehrerer Elemente, die ihrerseits wiederum Tupel sein können. Eine spezielle Bedeutung kommt dem null-elementigen Tupel [] zu: Dieses wird wie eine klassische schwarze Marke aus einem P/T-Netz behandelt und verwendet.

Spezielle Kantenarten Eine weitere Eigenheit der Anschriftensprache liegt in den zusätzlichen Kantentypen. Kanten, die gänzlich auf Pfeilspitzen verzichten, tragen den Namen »Testkanten«. Testkanten haben die semantische Bedeutung eine Marke einer Transition zum Feuern zur Verfügung zu stellen, diese dabei aber nicht zu verbrauchen. Darüber hinaus wird die Marke für den Feuervorgang einer schwächeren Reservierung unterzogen, welche erlaubt, dass weitere Testkanten eine zusätzliche Reservierung auf die Marke anmelden können. Insbesondere können mehrere Transitionen (oder sogar die gleiche Transition) dieselbe Marke in mehreren Feuervorgängen nebenläufig verwenden, wenn alle beteiligten Transitionen die Marke über eine Testkante bereitgestellt bekommen. Der übliche Einsatz von Testkanten umfasst den lesenden Zugriff auf Informationen.

Ähnlich den Testkanten existieren »rückgerichtete Kanten«, welche auf beiden Seiten eine Pfeilspitze aufweisen. Während ihr Verhalten auf den ersten Blick denen der Testkanten gleicht, können Sie jedoch beim Feuern der Transition nicht mehrfach gebunden werden. Intuitiv entspricht dies dem Verhalten die Marke zu entfernen und nach dem Feuern wieder in den Platz zu legen. Rückgerichtete Kanten sollten für die Modellierung von exklusiv genutzten Ressourcen oder physikalischen Gegenständen eingesetzt werden.

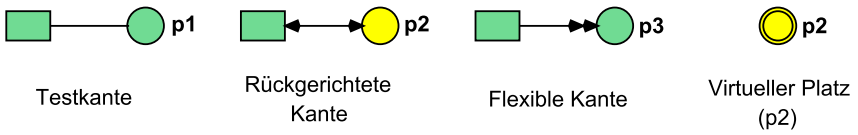

Abbildung 2.8.: Spezielle Kantentypen und ein virtueller Platz

»Flexible Kanten« weisen an einem Ende zwei schwarze Pfeilspitzen auf und können dafür verwendet werden Sammlungen und Arrays in ihre Bestandteile zu entpacken bzw. solche zu verpacken.

Darüber hinaus existieren zwei weitere Kantentypen, die jedoch nicht mit True-Concurrency-Semantik auswertbar sind, sondern nur mit einer sequentiellen Simulation des Netzes ausgewertet werden können. Dabei handelt es sich zum einen um die »Clear-Kanten«, welche nur von Platz zu Transition verfügbar sind und alle Marken aus dem Platz abziehen, sowie die »Inhibitor-Kanten«, auch bekannt als Nulltestkanten[26]. Eine Transition mit Inhibitor-Kante kann nur aktiviert sein (notwendige Bedingung), wenn der angeschlossene Platz keine Marken enthält. Beide Kantentypen lassen sich nur schwer mit den Forschungsfragen und skalierender, verteilter Simulation kombinieren und sind daher hier nur am Rande erwähnt.

Eine Übersicht aller relevanter Kantentypen findet sich in Abbildung 2.8.

Virtuelle Plätze Virtuelle Plätze sind ein rein syntaktisches Element und haben keine weiteren Eigenschaften bezogen auf den Formalismus. Sie werden mit einem doppelten Außenring als Platz dargestellt und stellen lediglich einen Platz an anderer Stelle dar und dienen der Übersichtlichkeit von Netzen. Zwischen virtuellen Plätzen und synchronen Kanälen besteht keinerlei Verwandschaft. Während unangepasst keine Möglichkeit existiert Rückschlüsse auf den abgebildeten Platz eines virtuellen Platzes zu ziehen, bietet es sich jedoch an eine eindeutige Farbcodierung zu verwenden. Ein Beispiel findet sich ebenfalls in Abbildung 2.8. Platz $p2$ wird in gelb und rechts als virtueller Platz erneut abgebildet.

2.8.3. Beispiele zu Referenznetzen

Mit der Einführung des in Renew eingesetzten Formalismus kann nun ein Beispiel für ein Referenznetz, wie es im Rahmen der Arbeit Verwendung findet, gegeben werden.

[26] Als Exkurs stellt die Einführung von Inhibitor-Kanten in P/T-Netze das Schwellenelement zur Turing-Mächtigkeit dar.

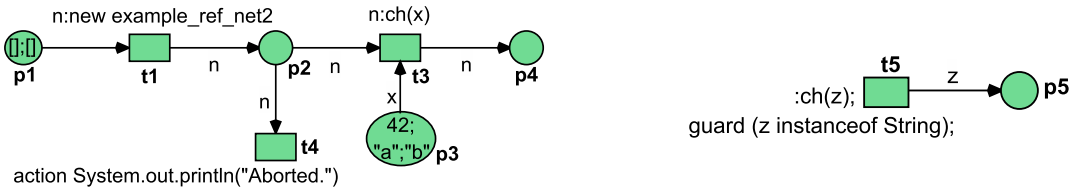

Abbildung 2.9.: Ein Beispiel für ein Referenznetz

Beispiel 2.24. Das in diesem Beispiel verwendete Referenznetz findet sich in Abbildung 2.9. Dieses umfasst zwei statische Netze, die mit den Bezeichnern *example_ref_net1* (kurz »das erste Netz«) bzw. *example_ref_net2* (kurz »das zweite Netz«) versehen sind. Wird mit einer Netzinstanz vom ersten Netz begonnen, so befinden sich initial zwei schwarze Marken (Dargestellt durch *[];[]*) im Platz $p1$, sowie drei Marken in Platz $p3$. Dadurch ist in dieser Netzinstanz die Transition $t1$ aktiviert und kann feuern. Durch die spezielle Anschrift der Transition $t1$ (»n: new ...«) entsteht jeweils eine neue Netzinstanz des zweiten Netzes. Die Referenz wird durch die Variable n ihrerseits referenziert und so im Platz $p2$ abgelegt. Die Transition $t5$ in der neu erzeugten Netzinstanz vom zweiten Netz hat offensichtlich keinen Vorbereich. Durch die Anschrift *:ch(z)* wird jedoch ausgedrückt, dass die Transition den Uplink eines synchronen Kanals *ch* darstellt. Da Uplink-Transitionen nicht autonom schalten können, kann die Transition nur durch das Feuern einer Transition schalten, die einen Downlink auf den Kanal *ch* hält und gleichzeitig eine Referenz auf die Netzinstanz, deren Teil die Transition $t5$ ist.

Eine solche Bedingung ist in Transition $t3$ gegeben, welche durch das Feuern von $t3$ und $t5$ eine Unifikation innerhalb des synchronen Kanals auslöst. Der Kanal beinhaltet einen Parameter, welcher zum Informationsaustausch genutzt werden kann. Auf der Seite des ersten Netzes ist dieser als x referenziert, während er beim zweiten Netz den Bezeichner z trägt. Im Vorbereich von $t3$ existieren zwei Zeichenketten-Marken und eine Integer-Marke, welche an x gebunden werden können. Für die Unifikation muss zusätzlich das Guardprädikat (z instanceof String) erfüllt sein. Somit gelingt die Unifikation und eine der beiden Zeichenketten-Marken wird in $p5$ in der Netzinstanz des zweiten Netzes abgelegt.

Alternativ hätte an dieser Stelle auch $t4$ feuern können. Da $t4$ keine neue Referenz auf die Netzinstanz hinter n generiert, geht die Referenz verloren. Existieren keine Referenzen mehr auf die Netzinstanz (auch aus Java Klassen und der GUI), schalten keine Prozesse mehr in der Netzinstanz und sind sonst auch keinerlei Aktivitäten mehr vorhanden in der Netzinstanz, wird sie vom Java Garbage Collector entsorgt. $t4$ besitzt eine action-Anschrift, welche sicherstellt, dass die Operation (Logzeile auf der Konsole erzeugen) erst beim Feuern von $t4$ ausgeführt wird und nicht etwa schon bei der Bindungssuche.

Nebenläufig zu allem bisherigen Verhalten hatte $t1$ die Möglichkeit (erneut) zu schalten und nachfolgend (erneut) $t3$ oder $t4$. Nachdem $t1$ und $t3/t5$ bzw. $t4$ je zwei mal geschaltet haben, ist keine Transition mehr aktiviert. In erreichbaren Zuständen, in denen keine Transition mehr schalten kann, existieren bis zu drei Netzinstanzen: Eine vom ersten und zwei vom zweiten Netz. Für bis zu zwei[27] der beiden Netzinstanzen des zweiten Netzes liegen Referenzen im Platz $p4$. In nur einer Netzinstanz des zweiten Netzes liegt die Marke »a« im Platz $p5$ respektive die Marke »b«. Die Marke »42« kann niemals in einer Netzinstanz des zweiten Netzes auftreten, da dies durch das Guardprädikat verhindert wird.

2.8.4. Algorithmen

Der folgende Abschnitt gibt eine Zusammenfassung der wesentlichen Algorithmen und Datenstrukturen, die im Zusammenhang zu Referenznetzen existieren. Die Inhalte richten sich nach Olaf Kummers Dissertationsschrift (Kummer, 2002, Kapitel 14). In der Originalquelle ist der Aufbau in fünf Bereiche gegliedert, die jeweiligen Teilalgorithmen entsprechen. Auf einige dieser Anteile wird nachfolgend detaillierter eingegangen. Der Anspruch liegt dabei nicht auf Vollständigkeit, sondern darauf ein ausreichendes Verständnis der Inhalte für den Rahmen dieser Arbeit zu vermitteln. Für weiterführende Details sei auf die Quelle (Kummer, 2002, Kapitel 14) verwiesen.

- Ein Unifikationsalgorithmus. Da in Petrinetzformalismen in der Regel keine Vorgabe für die Richtung des Informationsflusses gemacht wird und lediglich mögliche Handlungen an den Transitions- und Kanteninschriften zu finden sind, sind Zuweisungen und Parameterübergaben ungeeignet. Obwohl eine volle Unifikation wesentlich schwieriger zu implementieren ist, bildet sie doch das genaue Verhalten des Netzes weit besser ab. Daher wurde in den Originalalgorithmen ein Unifikationsanteil integriert. Intuitiv belegt die Unifikation die Variablen bei einer Aktivierung einer Transition mit möglichen passenden Marken oder mit Ergebnissen von Berechnungen aus Marken.
- Datenstrukturen zur Speicherung von Referenznetzen. Petrinetze im Allgemeinen und Referenznetze im Speziellen sind neben ihren statischen Eigenschaften äußerst zustandsbehaftet. Während in klassischen Petrinetzen der aktuelle Zustand des Netzes mit Hilfe der aktuellen Markierung ablesen lässt, sind Referenznetze diesbezüglich weniger intuitiv.

[27] Für jeden Schaltvorgang von $t4$ wird jeweils die letzte Referenz auf die jeweilige Netzinstanz vom zweiten Netz entfernt. Zusätzlich ist darin keine Transition aktiviert. In einem solchen Fall entfernt der Java Garbage Collector die Netzinstanz vollständig aus dem Speicher.

- Suchraumhandhabung zur Prüfung auf Aktiviertheit von Transitionen. Petrinetztheorie basiert auf Nebenläufigkeit, daher wäre es ungünstig, wenn nach jedem Feuern für jede Transition erneut geprüft werden würde, ob diese aktiviert ist. Glücklicherweise kann jedoch der Suchraum deutlich eingeschränkt werden, da eine nicht aktivierte Transition, deren Vorbereich sich nicht ändert, nicht plötzlich aktiviert sein kann. Die in diesem Zusammenhang notwendigen Teilalgorithmen werden im Unterabschnitt Suchraumreduzierung behandelt.
- Bindungssuche einer einzelnen potentiell aktivierten Transition. Sobald eine potentiell aktivierte Transition entdeckt wurde, muss die Prüfung erfolgen, ob diese wirklich aktiviert ist. Die Bindungssuche spielt eine zentrale Rolle in den Algorithmen und wird zur Simulationszeit des Netzes sehr oft ausgeführt werden. Daher ist es zentral die Bindungssuche zu optimieren um starke Verzögerungen zu vermeiden.
- Feuern einer Transition mit Verbrauch und Erzeugung von Marken. Sofern die Bindungssuche eine aktivierte Transition gefunden hat, können die Marken ihres Nachbereichs berechnet werden. Ein besonderes Augenmerk ist hierbei auch auf mögliche Konfliktsituationen zu richten, da andere Feuervorgänge mit überschneidendem Vorbereich nebenläufig versuchen könnten gleiche Marken zu konsumieren.

Datenstrukturen

Zunächst gilt es die statische Netzstruktur abzulegen. Die damalige Designentscheidung durch (Kummer, 2002) beinhaltete, dass für kein (statisches) Netz Code generiert wird, sondern dass das Verhalten in der Datenstruktur abgelegt wird. Neben den naheliegenden Klassen *Place* und *Transition* werden Kanten als spezielle Transitionsanschrift aufgefasst. Statische Netze enthalten daher nur diese drei Arten von Entitäten, wobei diverse Subklassen der Transitionsanschrift existieren, wie etwa für Kanten, Guards, Uplinks, Downlinks und Netzinstanzerzeugung. Besonderes Interesse genießen außerdem die Ausdruck- und Action Transitionsanschriften, da diese auszuwertende Ausrücke repräsentieren.

Bindungssuche

Ein elementarer Bestandteil der Simulationsalgorithmen von Renew ist der Suchalgorithmus, welcher auch als Bindungssuche bezeichnet wird. Innerhalb der Simulationsalgorithmen kommt der Bindungssuche die Aufgabe zuteil potenziell aktivierte Transitionen zu finden und für ein mögliches Feuern vorzubereiten. Dabei müssen unterschiedliche Elemente des Referenznetzes wie Guards, action-

Anschriften und weitere unterschiedlich behandelt werden. So ist es beispielsweise notwendig bereits vor dem Feuern eine gewisse Auswertung der Variablen und Belegung für die Unifikation vorzunehmen, um beurteilen zu können, ob die Unifikation überhaupt möglich ist.

Der Aufbau des Suchalgorithmus umfasst die drei Hauptbestandteile: Searcher, Binder und Finder.

Searcher Die Aufgabe des Searchers besteht darin den gesamten Suchvorgang zu kontrollieren. Diese Kontrolle umfasst unter anderem die Referenzierung aller benötigten Informationen, sowie die Kontrolle zugehöriger Binder- und einem Finder-Objekt. Ein Searcher beauftragt im Normalfall mehrere Binder Objekte und ein einzelnes Finder Objekt.

Binder Binder Objekte haben entsprechend ihres Namens die Aufgabe Variablenbindungen innerhalb einer Unifikation vorzunehmen. Dazu kann ein Searcher Objekt beim Binder eine Information einholen, wie aufwändig ein Durchtesten aller aktuell möglichen Bindungen vermutlich ist. Die Binder Objekte werden sukzessive vom Searcher Objekt aufgerufen, um über Informationen über alle möglichen Bindungen zu sammeln. Dabei wird stets demjenigen Binder Objekt der Vortritt gewährt, welches den vermutlich geringsten Aufwand aufweist eine Bindung zu finden.

Finder Ein Finder Objekt kann eine mögliche Lösung aufnehmen und sie zur Weiterverarbeitung bereitstellen. Bei dieser Weiterverarbeitung handelt es sich in aller Regel um ein möglicherweise folgenden Schaltvorgang.

Feuern von Transitionen

Das Feuern von Transitionen umfasst mehrere Schritte. Zunächst müssen die gefundenen Marken entsprechend des Finder Objektes gesperrt werden. Nachfolgend können alle action-Inschriften ausgeführt werden.

Es wird angenommen, dass action-Inschriften nicht auf technischer Ebene fehlschlagen können. Implementationsseitig bedeutet dies, dass entsprechende Fehlerbehandlungen in der Implementation der Inschrift vorhanden sein müssen und stets ein erwartbares Ergebnis geliefert wird. Insbesondere unbehandelte Ausnahmen dürfen daher hier nicht auftreten.

Nachdem alle Berechnungen durchgeführt wurden, werden die erzeugten Marken in den Nachbereich der Transition abgelegt und der Feuervorgang endet erfolgreich. Sind die Marken, welche vom Finder-Objekt referenziert werden zwischenzeitlich durch eine andere Transition konsumiert worden, werden keine action-Inschriften ausgeführt, der Feuervorgang wird abgebrochen und ggf. wird ein neuer Suchprozess angestrengt.

2.8.5. Plugin-System

Während der initiale Entwurf lediglich durch Formalismenwahl zur Kompilierzeit modulithischen Charakter hatte, zeichnete sich mit hinzukommender Funktionalität ab, dass eine Entkoppelung der Codebasis von Renew unausweichlich wird. Maßgeblich zur Entkoppelung beigetragen haben die Arbeiten von Duvigneau und anderen, welche umfassend in der zugehörigen Literatur (Duvigneau, 2009) beschrieben sind. Auf Basis der Komponentenorientierten Softwareentwicklung entstand das Plugin-System für Renew. Definitionen zu Softwarekomponenten finden sich beispielsweise in den Arbeiten (Sametinger, 1997), (Griffel, 1998), (Councill und Heineman, 2001) und (Szyperski, Gruntz und Murer, 2002). Für eine genauere Unterscheidung bietet es sich an die Aufstellung in (Duvigneau, 2009, Seite 44ff.) zu betrachten.

Ein maßgeblicher Vorteil der komponentenorientierten Softwareentwicklung besteht darin mit einem vergleichsweise kompakten und damit leichter zu wartenden Kernsystem arbeiten zu können, während das finale System nutzerspezifisch um die gewünschte individuelle Funktionalität erweitert werden kann. Ein weiterer Vorteil entsteht durch starke Kapselung, welche ebenfalls perspektivisch für eine intendierte Verteilung des Systems interessant werden kann. Eine Herausforderung besteht darin die Komponenten so dynamisch zu erzeugen, dass Plugins leicht ins System integriert werden können, ohne dass Code der Basis verändert werden müsste. Dabei sind einige Architekturmuster wie beispielsweise das Beobachter-Muster[28] hilfreich.

Zur Zeit der Veröffentlichung der Arbeit Duvigneaus war die Beschreibung von Plugin-Systemen aus wissenschaftlicher Sicht weniger fundiert, obgleich Plugin-Systeme vielfachen Einsatz in der Praxis finden und fanden. Daher entstand mit der Arbeit (Duvigneau, 2009) eine konzeptuelle Modellierung von Plugin-Systemen. Ferner half das Plugin-System von Renew als Fallstudie bei der Beschreibung von Softwarearchitekturen durch die inhärente Nebenläufigkeit und Modellierungsmächtigkeit von Referenznetzen, welche bei vielen Architekturbeschreibungskonzepten normalerweise fehlen oder wenig ausgeprägt sind. Die Modellierung durch Referenznetze wurde abgewogen gegen andere mögliche Kandidaten, wie beispielsweise die Netscape Plugin-Architektur, Mozilla Extensions, Eclipse Plugins und weiteren. Letztendlich ergab sich eine Überlegenheit der Petrinetz-Modellierung gegenüber den Vergleichssystemen.

Aus der Sichtweise von Renew ergab sich mit Einführung des Plugin-Systems eine völlig neue und im Kern unterschiedliche Version von Renew: Renew 2.0. Dabei ist zentrales Element in Renew der (Plugin-)Loader. Plugins umfassen den Simulator, welcher die Simulationsalgorithmen kapselt, die graphische Oberfläche, sowie diverse weitere darauf aufbauende Plugins.

[28] Beobachter-Muster vgl. beispielsweise (Gamma u. a., 1994)

Über die Jahre kamen eine Vielzahl zusätzlicher Plugins zu Renew hinzu, welche umfassende neue Funktionalitäten einführten. Diese Funktionalitäten erweitern teils andere Plugins oder sind weitestgehend eigenständig. Trotz der komponentenorientierten Architektur von Renew 2.0 ist das Deployment jedoch weniger dynamisch als es zunächst erscheint. Durch die Loader-Funktionalitäten ist es möglich zur Laufzeit Plugins zu laden, jedoch existiert (seitens Java) keine Möglichkeit geladene Klassen wieder zu entladen. Abhilfe schafft hier die Entwicklung in Richtung des Modulkonzeptes, welches mit Java 9 unter dem Namen »Project Jigsaw« eingeführt wurde. Die Modularisierung und die Transition in Richtung Renew 4.x wird gesondert in Abschnitt 2.8.9 adressiert.

2.8.6. Persistenz

Wie bei jeder Simulation, ist es auch bei der Simulation von Referenznetzen von Interesse die Simulation persistent pausieren zu können, um sie zu einem späteren Zeitpunkt oder in einer weiteren Wiederholung erneut aufzunehmen. Auch der Transfer einzelner Netzinstanzen und deren aktueller Zustand kann wünschenswert sein. Für einfachere Formalismen von Petrinetzen ist die Implementation von Persistenz auch mehr eine Fleißarbeit, als eine solche, deren Lösung weitere Überlegung erfordert.

Die Implementation von Referenznetzen auf Basis der Programmiersprache Java, wie sie in Renew umgesetzt ist, basiert jedoch auf dem Java Objektsystem. Objekte können nicht triviale Verbindungen wie beispielsweise Zyklen untereinander auf Basis ihrer Referenzen besitzen. Daher kann nicht jedes einzelne Java Objekt problemfrei in eine persistierbare Repräsentation überführt werden.

Java bietet für die Serialisierung von Javaobjekten das Interface *Serializable* an, welches aber nicht standardmäßig implementiert werden muss. Während es theoretisch denkbar ist, Objekte nur auf Basis ihrer Referenzen untereinander und vollumfänglich zu sichern, wäre diese Methode nicht sehr feingranular. Sie würde bedeuten, dass im Zweifel nur das gesamte laufende Ökosystem im Ganzen gesichert werden könnte.

Trotz dieser Hürden in der Umsetzung wurde bereits in der Veröffentlichung (Jacob, 2001) die Persistierung von Referenznetzsimulationen in einer Datenbank beschrieben. Dabei wurde nach jedem Feuervorgang der neue Zustand in die Datenbank geschrieben, ungeachtet dessen, ob dieser Zustand gegebenenfalls auch deterministisch erneut berechenbar wäre. Die hierdurch erreichte Persistenz kann zwar durch Konsistenz überzeugen, verursacht jedoch auch ein erhebliches Ausbremsen des Simulationssystems.

2.8.7. Netzcompiler und Schattennetze

Die Darstellung von Petrinetzen in ihrer grafischen Form ist insbesondere für menschliche Betrachter von Nutzen. Wenn jedoch wie im Fall von Referenznetzen das Modell ebenfalls eine direkte Ausführbarkeit bereitstellt, muss eine Mechanik für die Übersetzung von Modell zu ausführbarem Code existieren. Renew realisiert dies durch den Einsatz eines *Netzcompilers*.

Der Netzcompiler interpretiert das Netz unter der Angabe eines bestimmten Formalismus, wie beispielsweise Referenz- (Java-) oder P/T-Netze. Dabei wird aus den grafischen Netzen eine andere Repräsentation berechnet, welche lediglich die für die Ausführung relevanten Informationen beinhaltet. So werden beispielsweise Informationen zur räumlichen Anordnung von Transitionen und Plätzen oder auch Farben nicht übernommen, da diese keine Information bezüglich der Ausführung tragen.

Das resultierende Datenformat wird als *Schattennetz* (engl.: *shadow net*) bezeichnet. Dabei können Schattennetze bereits kompiliert oder aber noch unkompiliert sein. Renew stellt die Möglichkeit bereit, mehrere Schattennetze in einem Verbund zu persistieren, dieser Verbund wird als *Schattennetzsystem* (engl.: *shadow net system*) bezeichnet. Während herkömmliche Renew Netze die Dateiendung ».rnw« tragen, tragen Schattennetzsysteme die Endung ».sns«.

Besonderes Augenmerk im Rahmen der Arbeit genießen persistierte und kompilierte Schattennetzsysteme, da durch sie auf einfache Weise Ausführungsinformationen an Renew Simulatoren übermittelt werden können; auch dann, wenn diese Simulatoren keine grafische Oberfläche besitzen.

2.8.8. Verteilte Simulation

Eine grundlegende Eigenschaft, die jedes skalierbare System aufweisen muss, ist die Fähigkeit in einer verteilten Umgebung lauffähig zu sein. Über die Jahre wurden für Renew viele verschiedene Ansätze der Verteilung überlegt, diskutiert und implementiert. Eine wesentliche Schwierigkeit dabei besteht darin, dass die Bindungssuche um potenziell aktivierte Transitionen finden zu können, im Allgemeinen ein globales Wissen über das System besitzen muss, welches im verteilten Fall nicht gewährleistet werden kann. Dennoch wurden die verschiedenen Ansätze der Verteilung mit verschiedenen Einschränkungen versehen, sodass die Ausführung zumindest in bestimmten Situationen gut abgebildet werden kann. Dabei sollen an dieser Stelle die beiden wichtigsten verschiedenen Ansätze zur verteilten Ausführung von Renew betrachtet werden: das Distribute-Plugin, sowie die Ausführung von Renew in OpenShift und RenewGrass.

Distribute-Plugin

Das Distribute-Plugin entstammt der Abschlussarbeit von Michael Simon (M. Simon, 2014) und wurde in der zugehörigen Veröffentlichung (M. Simon und Moldt, 2016) publiziert. Das Distribute-Plugin erweitert den Renew Simulator um den Distribute-Formalismus, welcher im Kern eine Erweiterung der klassischen Bindungssuche und des Feueralgorithmus darstellt. Es basiert auf der Remote Method Invocation – kurz RMI – und deren Implementation in der Programmiersprache Java: JavaRMI. Um die Implikationen aus diesem Umstand nachvollziehen zu können, ist es notwendig eine kurze Einführung in die Besonderheiten von JavaRMI zu geben, welches im weiteren Verlauf des Abschnittes geschehen soll. Darüber hinaus fügt das Distribute-Plugin einige statische Methoden zum Verwalten von entfernten Netzinstanzen hinzu sowie zum Registrieren lokaler Netzinstanzen. Ferner erweitert es die Syntax des Referenznetz-Formalismus, wie er in Renew umgesetzt ist, indem spezielle Parameter für den Aufruf verteilter synchroner Kanäle zugefügt werden, sowie für den bidirektionalen Datenaustausch ohne Unifikation. Bei diesen Änderungen handelt es sich um eine konservative Erweiterung des Referenznetzformalismus.

Eingriff in den Suchalgorithmus
Wie eingehend beschrieben greift das Distribute-Plugin in erster Linie in den Suchalgorithmus als Bestandteil der Simulationsalgorithmen ein. Dabei werden beim Erkennen eines verteilten Downlinks zunächst die angesprochene Netzinstanz und die jeweiligen Parameter ausgewertet. Darauf folgt ein Einfrieren der hiesigen lokalen Suche und ein Senden der Parameter an die anvisierte Netzinstanz. Bei der angesprochenen Netzinstanz wird sodann versucht eine valide Bindung für einen Uplink, welcher von dem betreffenden synchronen Kanal angesprochen werden kann, zu finden. Sobald dies abgeschlossen ist, wird auch die entfernte dort lokale Suche eingefroren und die Ergebniswerte werden zurück an die Transitionsinstanz übermittelt, welche den Aufruf getätigt hat. Dort wird dann die lokale Bindungssuche weiter fortgesetzt. Wenn auf diesem Wege eine Bindung gefunden werden kann, wird eine der verfügbaren lokalen Bindungen gefeuert. Als Besonderheit gilt es noch zu erwähnen, dass die Bindungssuche bei der entfernten Netzinstanz ebenfalls erneute Channels ansprechen und öffnen kann. Somit kann ein rekursives Verhalten in den entfernten Aufrufen entstehen.

Eingriff in den Feueralgorithmus
Das Ziel beim Anpassen des Feueralgorithmus besteht darin, eine gefundene Bindung global atomar zu feuern. Dabei setzt das Distribute-Plugin ein Sperrverfahren auf der Basis einer Totalordnung auf den IDs der Simulationen und Platzinstanzen ein. Global werden die Simulationen anhand dieser Totalordnung abgearbeitet und innerhalb einer Simulation anhand der Totalordnung auf den IDs der jeweiligen Platzinstanzen. Dabei gilt der Grundsatz, dass falls ein einziger

Feuervorgang an einer Stelle fehlschlägt, alle Feuervorgänge global gemeinsam fehlschlagen müssen. Die Umsetzung entspricht somit eingeschränkt einem klassischen verteilten Commitprotokoll.

Statische Methoden von Distribute
Die zentralen statischen Methoden, welche vom Distribute-Plugin bereitgestellt werden, umfassen das Registrieren einer Netzinstanz unter einem bestimmten Namen, dass Holen einer Netzinstanz mithilfe eines bestimmten Namens, sowie das Erzeugen einer Distribute-Referenz aus der aktuell aktiven Netzinstanz zum Einsatz in der Registrierung. Dies geschieht unter der Grundannahme, dass eine zugehörige externe Registry bereits mit dem Start des RENEW Simulators vorliegt und ebenfalls statisch angesprochen werden kann. Das Distribute-Plugin unterstützt keinen Wechsel der Registry im laufenden Betrieb. Die oben genannten Befehle können aufgerufen werden mit den Anschriften:

```
DistributePlugin.getRegistry().getNetInstance("Name");
DistributePlugin.getRegistry().registerNetInstance("Name",
   <Distr.-Ref.>);
DistributePlugin.wrap(<lokale-Ref.>);
```

Anschriftensprache von Distribute
Ein grundlegender Unterschied zwischen der Bindungssuche in Distribute und der klassischen lokalen Bindungssuche besteht darin, dass – wie zuvor beschrieben – keine echte Unifikation mit den entfernten Daten erfolgt, sondern lediglich ein Roundtrip. Dadurch entsteht die Situation, dass Parameter nicht in beliebiger Tiefe unifiziert werden können und sich somit grundlegend anders verhalten, als ihr lokales Pendant in synchronen Kanälen. Um dieser Besonderheit gerecht zu werden, verwenden verteilte Kanäle eine spezielle Syntax: Anstatt eines Doppelpunktes (:) wird ein Ausrufezeichen (!) im Downlink eingesetzt. Darüber hinaus muss kenntlich gemacht werden, welche Variablen beim initiierenden Downlink vor dem Roundtrip ausgewertet werden können und welche noch Informationen vom entfernten Synchronisationspartner benötigen. Dazu verwendet das Distribute-Plugin die Konvention Variablen, die bereits vor der Kommunikation ausgewertet werden sollen, identisch zu synchronen Kanälen in Klammern hinter dem Kanalnamen zu notieren. Variablen, welche erst nach der Kommunikation beim Downlink zur Verfügung stehen, werden im Anschluss an den Aufruf mittels »->« gefolgt vom Variablennamen notiert. Auf der anderen Seite der Kommunikation – beim Uplink – existieren ebenfalls solche Konventionen. Variablen, die lokal gebunden werden können, werden in den Klammern analog zu klassischen synchronen Kanälen notiert. Dabei spielt es im Falle des Uplinks keine Rolle, ob die Variable von einem entfernten Downlink oder lokal gebunden werden muss. Soll eine Variable zur weiteren Verarbeitung, bzw. zur entfernten Bindung beim entfernten Downlink bereitgestellt werden, so kommt hier die analoge Syntax »<-« zum Einsatz.

Zur Übersicht sind die Syntaxerweiterungen durch das Distribute-Plugin noch einmal mit Beispielen in Tabelle 2.1 aufgeführt.

Ausdruck	**lokal geb.**	**entfernt gebunden**	**Erläuterung**
:ch(x)	x	x (verteilt)	Uplink. Als klassischer synchroner Kanal mit lokaler Unifikation. Als verteilter Uplink mit entfernter Bindungsvorgabe durch den Downlink.
ref:ch(x)	x	-	Downlink eines klass. synchronen Kanals mit Netzinstanz ref.
ref!ch(x)	x	-	Distribute-Downlink eines verteilten Kanals mit entfernter Netzinstanz ref
ref!ch(x) ->y	x	y (Uplink)	Distribute-Downlink mit lokaler und entfernter Bindung.
:ch(x) <- y	y	x (Downlink)	Distribute-Uplink. x wird vom initiierenden Downlink vorgegeben, y wird lokal unifiziert und im Rückweg dem Downlink zur Verfügung gestellt.

Tabelle 2.1.: Syntaxerweiterungen durch das Distribute-Plugin

Beispielnetz zum Distribute-Plugin

In Abbildung 2.10 findet sich eine typische Beispielanwendung für einen Einsatz des Distribute-Plugins. Dabei ist die Aufgabe der Beispielanwendung nicht weiter spezifiziert, es ist lediglich die abstrakte Struktur gegeben. In einer Komponente werden Arbeitspakete gesammelt, welche dann durch Worker-Netze abgearbeitet (»doWork()«) werden können.

Es sind zwei Netze abgebildet, welche auf mindestens zwei voneinander entfernten Knoten ausgeführt werden. Gestartet werden mindestens zwei Simulationen mit einer einzelnen Netzinstanz des »Distributor«-Netzes, bzw. des »Worker«-Netzes. Um Konflikte zu vermeiden, bietet es sich an lediglich eine Simulation mit dem »Distributor« Netz zu starten und alle weiteren mit »Worker«-Netzen.

Alle als weiß dargestellten Plätze dienen der Vorbereitung und Initialisierung von Distribute. $t1$ registriert die Netzinstanz des Distributors in der Registry. Die Folge $t4 \rightarrow t5$ schaltet genau so lange, bis ein Distributor in der Registry verfügbar wird.

Nach dem Setup kann $t6$ schalten. $t6$ enthält keine lokal zu bindenden Variablen, sodass direkt der verteilte Kanal verfolgt werden kann, welcher einen zugehörigen Uplink in der Transition $t2$ findet. x wird an eine Marke aus dem Platz $p3$ gebunden, in diesem Beispiel $4, 2$ oder 7. Gleichzeitig muss in Platz $p2$ eine Marke zumindest vorhanden sein. Somit kann die Bindung erst nach Registrierung der Netzinstanz durch $t1$ erfolgen. Nach erfolgreicher Bindung von x, wird die betreffende Bindung an $t6$ zurückgesandt, wo das gelieferte x lokal an die zufällig gleichnamige Variable x gebunden wird. $t6/t2$ kann nun feuern.

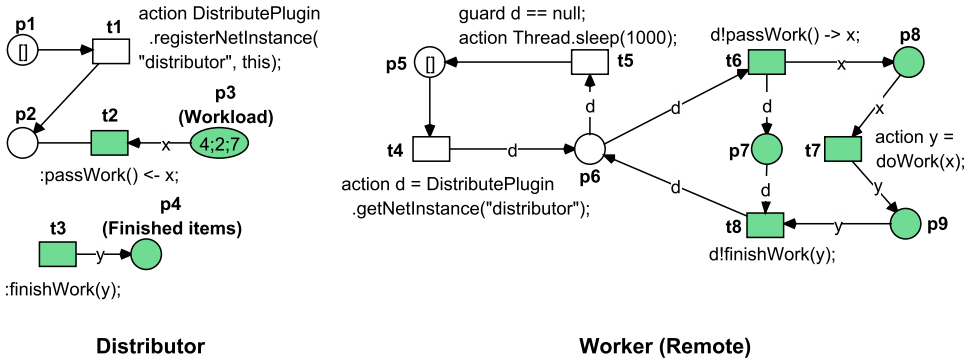

Abbildung 2.10.: Typische Beispielanwendung des Distribute-Plugins

Die Abarbeitung durch den Worker erfolgt nun über die Folge $p8 \rightarrow t7 \rightarrow p9$, während in $p7$ die Referenz auf den Distributor zwischengespeichert wird. Nebenläufig zum gesamten Ablauf können weitere Worker Arbeitspakete aus dem Distributor mit der zuvor genannten Methodik abziehen.

Sobald der Auftrag erledigt ist und das Ergebnis vorliegt, kann $t8$ nach einer Bindung suchen. Da y lokal vor jeglicher Kommunikation gebunden werden soll, wird es innerhalb der Klammern notiert. In der entfernten Netzinstanz weist $t3$ den korrekten Kanalbezeichner und die passende Parameterzahl auf. Da y bereits gebunden wurde, ist es fix vorhanden beim ersten Betrachten von $t3$ und kann an das dortige y gebunden werden. $t8/t3$ feuern und legen das Ergebnis in $p4$ ab. In der Worker Netzinstanz liegt nun wieder eine Referenz auf den Distributor in $p6$ und der Worker ist bereit das nächste Arbeitspaket zu empfangen.

JavaRMI und technische Umsetzung

Das Distribute-Plugin basiert technologisch auf JavaRMI. JavaRMI ist die Java Implementation des Remote Method Invocation Protokolls durch Sun Microsystems. Da die technischen Rahmenbedingungen durch diesen Umstand für einige technische Aspekte der Arbeit maßgeblich sind, werden sie an dieser Stelle grundlegend dargestellt.

Im Zentrum der Implementation steht der Einsatz einer Registry, welche das Auffinden von entfernten Routinen koordiniert. In einer Java-Anwendung, welche JavaRMI einsetzt, können neben klassischen Objekten sog. Objekt-Stubs existieren. Objekt-Stubs weisen die gleiche Schnittstelle auf, wie ein herkömmliches Objekt, jedoch liegen die zugehörigen Objekte auf einem entfernten Knoten und die Kommunikation wird durch JavaRMI realisiert. Das hat den Vorteil, dass die Implementation komplett transparent für den sonstigen Java Code stattfinden kann. Zusätzlich müssen allerdings alle Objekte, die als Parameter übergeben werden können, serialisierbar sein, sprich das »Serializable« Interface implementieren.

Bei der Kommunikation wird zunächst die JavaRMI Registry auf einem vorgegebenen TCP-Port adressiert um eine aktive Implementation eines Objekt-Stubs ausfindig zu machen. Daraufhin teilt die Registry dem Aufrufenden im Erfolgsfall die Adresse des implementierenden Knoten zusammen mit einem TCP-Port des Zielrechners mit. Der Knoten meldet sich beim neu gefundenen entfernten Knoten, welcher ihm einen zufälligen TCP-Port für alle weitere Kommunikation mitteilt. Der aufrufende Knoten wählt seinerseits ebenfalls einen zufälligen TCP-Port und teilt diesen mit. Die Kommunikation zwischen beiden läuft nun gänzlich ohne Registry über die gewählten TCP-Ports ab.

Folglich muss in einer normalen Ausführung des Distribute-Plugins darauf geachtet werden, dass die Registry bereitgestellt wird und dass keine Einschränkungen durch Firewalls bei der Konnektivität bezüglich freigegebener Ports besteht.

Ein weiterer Fallstrick entsteht beim Einsatz des Distribute-Plugins durch die Implementation der Bindungssuche. Liegen in einem Platz mehrere Marken vor und werden verschiedene mögliche Bindungen auf der entfernten Instanz durchprobiert, kann es unter Umständen zum Versand sehr vieler Marken kommen. Eine Netzinstanz mit einem Platz, der sehr viele Marken enthält, kann so die Simulation stark ausbremsen bis hin zum Totalausfall. Daher sollte nach Möglichkeit vor Distribute-synchronen Kanälen ein komplementärer Platz verwendet werden, um die Anzahl der Marken im Vorbereich der Transition zu limitieren.

Inter Cloud Nets (ICNets) und die Drop-Engine

Im Kern dieses Verteilungsansatzes stehen die Arbeiten (S. Bendoukha, 2013; S. Bendoukha, H. Bendoukha und Moldt, 2015a,b) und die zugehörige Dissertation (S. Bendoukha, 2017). Betrachtet wurde insbesondere Modellierung in Inter-Cloud-Anwendungen[29] mit Hilfe des Agentenparadigmas und Prozessen. Dabei liegt der Fokus auf der Bereitstellung von Modellierungswerkzeugen für Inter-Cloud Umgebungen mit besonderem Fokus auf Workflowlösungen. Weil die Ausrichtung der Arbeiten anders als in dieser Arbeit ist, werden im Folgenden die Schlüsselkonzepte nur kurz erläutert in Hinblick auf die Nutzbarkeit im Kontext von skalierenden verteilten Referenznetzsimulationen. Diese Konzepte umfassen RenewGrass als Einstieg, das entfernte Ausführen von Renew, die Cloud-Task-Transition, die Inter Cloud Nets sowie die Drop-Engine.

RenewGrass
RenewGrass ist ein Plugin für Renew, welches zum Erzeugen von Bildverarbeitungs-Workflows eingesetzt werden kann. Es basiert auf dem namensgebenden Geographic Resource Analysis Support System (Grass) Geographic Information

[29] Inter-Cloud-Anwendungen beziehen sich auf kooperierende Cloud Umgebungen, die ihrerseits aber jeweils autonom agieren. Siehe auch »Cloud of clouds«

System (GIS). In (S. Bendoukha, 2017) wird ein Beispiel mit Satellitenbildern vorgestellt, bei dem aus Ausgangsbildern der »Normalized Differences Vegetation Index« (NDVI) und der »Enhanced Vegetation Index« (EVI) berechnet wird. Da das Grundsystem Grass GIS in C entworfen wurde und Renew auf der Sprache Java basiert, liefert das RenewGrass Plugin eine Adapterschnittstelle zwischen den beiden Werkzeugen.

Ausführung von entfernten Renew Instanzen
Eine wesentliche Neuerung in der Arbeit (S. Bendoukha, 2017) besteht darin Renew Instanzen auf entfernten Maschinen auszuführen und mit ihnen zu kommunizieren. Dieser Ansatz ist eine Entwicklung orthogonal zu den Neuerungen durch (M. Simon und Moldt, 2016), da dort lediglich die Koppelung von mehreren Simulationen zu einer großen Simulation thematisiert wurde, nicht jedoch, wie solch eine Simulation ins Leben gerufen wird. Die Ausführung basierte technisch zunächst auf VirtualBox[30] und wurde später zu Vagrant[31] auf OpenStack[32] gewechselt. VirtualBox ist eine frei einsetzbare Lösung für virtuelle Maschinen von Oracle, Vagrant ein Tool zur automatisierten Konfiguration von virtuellen Maschinen und Rechnersystemen und OpenStack ein frei verfügbarer Cloud Plattform Provider, welcher ähnliche Funktionalitäten wie die kommerziellen Angebote Amazon AWS[33], Microsoft Azure[34] und Google Cloud[35] bietet. Die Lösungen werden allesamt durch ein einfaches Interface gekapselt und in Isolation ausgeführt.

ICNets und Cloud-Task-Transition
Cloud-Task-Transitionen (CTTs) sind von Workflow-Transitionen (vgl. (Jacob u. a., 2002) sowie (Wagner, 2018)) abgeleitet. Sie beschreiben einen Cloud-basierten Arbeitsschritt, der während der Ausführung fehlschlagen kann. Entgegen der für Transitionen üblichen Semantik können CTTs daher auch Marken zurück in ihre Vorbereiche legen, die zuvor zum Feuern abgezogen wurden. Jedoch kann eine CTT nur all die von ihr gebundenen Marken in ihren Vorbereich an die entsprechenden Stellen zurücklegen oder im Erfolgsfall die entsprechenden Ergebnismarken in ihrem Nachbereich platzieren.

Zu den Inputs einer CTT zählen Servicebeschreibung(en), Quality of Service (QoS) Anforderung(en), sowie (Daten-)Input(s) und eine Referenz auf Cloud-Ressourcen bzw. Cloud-Service-Providers. Beim Feuern der CTT werden im Hintergrund entsprechende Cloud-Service-Providers angesprochen und der benötigte Service mit den Daten ausgeführt. Sobald eine Antwort vom Cloud-Service-

[30] https://www.virtualbox.org/
[31] https://www.vagrantup.com/
[32] https://www.openstack.org/
[33] https://aws.amazon.com
[34] https://azure.microsoft.com/
[35] https://cloud.google.com/, alle URLs dieser Seite wurden am 15.10.2020 abgerufen

Provider eintrifft, werden die Ergebnisse in den Nachbereich platziert bzw. im Fehlerfall die Marken zurück in den Vorbereich gelegt.

Inter-Cloud-Nets (ICNets) sind für die Workflow-basierte Nutzung mehrerer Cloud Umgebungen vorgesehen. Neben den üblichen Komponenten eines gefärbten Petrinetzes weisen sie spezielle Transitionen und Plätze für je Daten- und Berechnungsoperationen auf, sowie Mengen für daten- und berechnungsorientierte Ressourcen und eine Menge an verschiedenen Cloud Umgebungen.

ICNets können als Erweiterung der CTTs für Inter-Cloud Umgebungen gesehen werden.

ICWorkflow
ICWorkflow ist ein weiteres Plugin für Renew und dient dem Zweck Inter-Cloud Workflows zu beschreiben und steuern. Die dabei gelösten Probleme umfassen Authentifizierung, Zugriffe, Deployments, die Ausführung von Workflows und Persistenzsteuerung. ICWorkflow dient als Grundlage der weiteren Integration der vorgestellten Komponenten.

Drop-Engine
Als letzter Punkt in der Arbeit (S. Bendoukha, 2017) werden die zuvor genannten Konzepte zusammengefasst und miteinander integriert. Die Gesamtheit der Integration von CTTs, ICNets, ICWorkflow und RenewGrass wird als Drop-Engine bezeichnet.

Die Architektur der Drop-Engine umfasst die komplette Strecke über den lokal ausgeführten Workflow, ein Cloud Portal, welches Workflows entgegen nimmt, die Integration mit einem Cloud Provider, sowie die Integration mit einem Cloud Broker, einem Cloud Repository und der Bereitstellung automatisch konfigurierter VM-basierter Cloud-Instanzen von Renew.

Als Beispiel wurde in (S. Bendoukha, 2017) RenewGrass als Cloud Anwendung in der Drop-Engine umgesetzt.

Renew Remoting

Schon früh in der Entwicklung von Renew bestand der Wunsch entfernt auf einen Renew-Simulator zugreifen zu können. Auf dieser Basis wurde Renew Remote geschaffen (Kummer, Wienberg, Duvigneau, Köhler u. a., 2003).

Renew Remote führt dabei eine Abstraktionsebene zwischen Simulator und grafischer Oberfläche ein, welche auf Java RMI basiert. In der klassischen Ausführung (lokaler Simulationskern und lokale Benutzeroberfläche) werden hierbei lokale Methoden aufgerufen. Das Nutzererlebnis bleibt in diesem Fall unverändert zum vollständig lokalen Fall ohne Java RMI. Falls gewünscht können an deren

Stelle jedoch auch entfernte Aufrufe an einen Renew Simulator abgesetzt werden. Die Anbindung erfolgt in einer vollständigen Ankoppelung an einen entfernten Simulator, sodass alle Informationen von und Interaktionen mit dem entfernten Simulator stattfinden anstatt dem lokalen.

Während die Lösung auch dem Bereich der Verteilung entstammt, ist sie doch anders gelagert als die Drop-Engine und das Distribute-Plugin. Sie befindet sich insbesondere auf der Ebene des Simulators und ist nicht innerhalb der Simulation verfügbar. Darüber hinaus ist nur eine eins zu eins Abbildung zwischen zwei Instanzen von Renew möglich, was auch unmittelbar erfordert, dass beide Hostsysteme in der Lage sein müssen Renew auszuführen. Ferner wird die aktive manuelle Mitwirkung eines Benutzers, welcher die Verbindung aufbaut, vorausgesetzt.

2.8.9. Modularisierung

Die Modularisierung von Renew erfolgte im Rahmen der Arbeiten (Daschkewitsch, 2019) und (Janneck, 2021). Dabei wird insbesondere der Aspekt der Hierarchie zwischen Modulen/Plugins wieder forciert und in die Implementation eingebettet. Es kommt das Modulsystem aus dem Java »Jigsaw« Projekt[36] zum Einsatz, welches in Java ab Version 9 allgemein verfügbar ist und insbesondere auch das JPMS (Java Platform Module System) beinhaltet. Durch dessen Einsatz kann eine deutlich bessere Kapselung erreicht werden, da Funktionalität auch auf Ebene der Programmiersprache gekapselt wird und nur über definierte Interfaces ansprechbar ist. Modulabhängigkeiten müssen darüber hinaus als gerichteter azyklischer Graph beschrieben werden und können nicht mehr beliebig aufgebaut sein. Der Einsatz von sog. *Modullayern* ermöglicht die Durchsetzung des Modulkonzepts auch für zur Laufzeit der Anwendung nachgeladenen Code.

Mit dem Einsatz von Modulen entsteht Version 4.0 von Renew. Zum Zeitpunkt der Erstellung dieser Arbeit besteht ein Pre-release von Renew 4.0. Auf der Basis von Modulen ist für die weitere Entwicklung eine Trennung der einzelnen Komponenten in verschiedene Repositories vorgesehen. Auf diese Weise können nachfolgende Weiterentwicklungen von Renew sehr flexibel konfiguriert werden und die Abhängigkeitsauflösung und -bereitstellung in den Buildprozess integriert werden.

[36] Siehe für weitere Informationen zu Jigsaw und Java Modulen beispielsweise: https://openjdk.java.net/projects/jigsaw/ - Zuletzt abgerufen am 05.01.2022

2.9. Zusammenfassung - Grundlagen

Die grundlegenden Konzepte, Methoden und bestehende Projekte, die zur Bearbeitung der Forschungsfragen eingesetzt werden können, wurden vorgestellt. Dabei fiel besonderes Augenmerk auf den Formalismus der Petrinetze, da diese sowohl ein mächtiges Werkzeug zur Modellierung verteilter und echt nebenläufiger Systeme darstellen, zugleich aber auch hervorragend mit Softwaresystemen nach der Agentenmetapher synergieren. Die hierarchischen Netze-in-Netzen und Referenznetze können den Unterbau für skalierende verteile Simulationen bilden. Ein zusätzlicher Vorteil von Referenznetzen basiert auf der Möglichkeit der nativen Einbettung des Modells in Java-Code mit dem Simulator Renew. Daneben wurden unter anderem die Begriffe der *Softwarearchitektur*, *Skalierbarkeit* und *Features* definiert sowie generelle Einführungen zu den Themen der Softwareagenten und insbesondere Paose, Entwicklungsmodalitäten und Deploymentformen gegeben.

3. Stand der Forschung

Mit den im Grundlagenkapitel eingeführten Definitionen und Begrifflichkeiten ist es nun möglich, bestehende Forschungsarbeiten in Hinblick auf die gestellten Forschungsfragen zu untersuchen. Dieser Prozess soll in vier Schritten erfolgen, welche sich gleichermaßen über dieses Kapitel und das Kapitel 4 zur Anforderungsanalyse verteilen. Zunächst wird die grobe Thematik der Arbeit abgesteckt. Zu diesem Zweck werden einzelne Unterthemen genannt und die zugehörigen Inhalte nach dem Verständnis dieser Arbeit beschrieben.

Anschließend wird im Rahmen dieses Kapitels ein Überblick über bestehende Architekturen und Arbeiten gegeben und diese bezüglich der Relevanz der Themen dieser Arbeit in den jeweiligen Untersuchungen eingeordnet. Auf dieser Basis kann bereits eine Vorauswahl von interessanten Ergebnissen getroffen werden, welche unter Umständen bereits Teile der Forschungsfragen beantworten können. Im zweiten Schritt in Kapitel 4 zur Anforderungsanalyse werden die aus den Forschungsfragen weiter zu Anforderungen konkretisiert und es erfolgt eine Evaluation der passenden bestehenden Architekturen im Kontext der Anforderungen. Auf dieser Basis erfolgt dann eine Auswahl eines Ausgangspunktes für den Hauptteil der Arbeit mit den besten Voraussetzungen, sodass bereits möglichst viele Anteile der Anforderungen erfüllt werden können.

3.1. Eingrenzung des Themas

Um eine sinnvolle Auswahl aus den großen Mengen publizierter Forschungsergebnisse treffen zu können, soll zunächst eine Eingrenzung des Themas erfolgen. Diese Basis wurde sowohl als Ausgangspunkt für Recherchen eingesetzt als auch für eine Evaluation der einzelnen individuellen Arbeiten und Forschungsfeldern bezüglich der Relevanz für das Thema der Arbeit. Die Aspekte ergeben sich im Wesentlichen aus den Ausführungen der Einleitung und insbesondere aus den gestellten Forschungsfragen.

Die einzelnen Aspekte müssen dabei nicht immer alle restlos erfüllt sein, damit eine Publikation für die Arbeit von Interesse ist. Je mehr Aspekte erfüllt sind, desto höher ist jedoch die Relevanz der entsprechenden Publikation. Arbeiten, die lediglich ein oder zwei Aspekte ansprechen, werden in der Betrachtung unter

ihrem Forschungsfeld zusammengefasst und dienen an verschiedenen Stellen in der Arbeit zur Abgrenzung zu bereits gelösten Fragen.

Zunächst wird durch die Arbeit ein Fokus auf die *Nebenläufigkeit* der beschriebenen Interaktionen gelegt. Dabei handelt es sich nicht um eine beliebige Form der Parallelität, welche beispielsweise durch eine Taktung verschiedener Komponenten eines verteilten Systems realisiert werden kann, sondern um eine echte Nebenläufigkeit. Diese zeichnet sich dadurch aus, dass verschiedene Ereignisse lediglich mit einer partiellen Ordnung bezüglich ihres zeitlichen Auftretens zueinander beschrieben werden können. Dies bedeutet insbesondere, dass zeitlich unvergleichbare Ereignisse explizit zugelassen sind.

Bereits zu Beginn wurde motiviert, dass die agentenorientierte Softwareentwicklung ein hervorragender Ansatzpunkt für die Beschreibung der Interaktionen zwischen Entitäten ist. Aus diesem Grund sollte die Thematik der *Agentenorientiertheit* und der *Interaktion zwischen Agenten* Beachtung finden.

Neben dem Einsatz der agentenorientierten Softwareentwicklung wurde motiviert, dass der Einsatz geeigneter Modelle insbesondere bei der Beschreibung eines Realweltsystems Vorteile mit sich bringt. Dabei soll die Annahme eines impliziten Modells nicht genügen, sondern es sollte eine explizite *Modellierung* in einer Modellierungssprache erfolgen.

Eine häufig vertretene Annahme bei der Konstruktion von Agentensystemen ist die Annahme eines verteilten Systems. Dabei soll es zunächst genügen, wenn das Agentensystem ein *verteiltes System* beschreibt. Im weiteren Verlauf der Arbeit wird diese Bedingung weiter eingeschränkt, sodass auch eine entsprechende Simulation des verteilten Systems selbst als verteiltes System realisiert werden können muss.

Durch die Einführung der Plattformidee ergibt sich zwangsläufig auch eine Unterscheidung zwischen *lokaler Kommunikation und entfernter (verteilter) Kommunikation*. Eine explizite Beachtung dieser Unterscheidung kann hilfreich sein, um realitätsnah Systeme beschreiben zu können. Darüber hinaus ist die aktive Plattformverwaltung durch Agenten ein zentraler Aspekt der Forschungsfragen. Diese starke Einschränkung bereits in dieser frühen Phase durchzusetzen, könnte jedoch diverse interessante Forschungsergebnisse von vornherein ausschließen. Deswegen soll an dieser Stelle nur eine Begrenzung auf eine *beliebige Form der Plattformverwaltung* erfolgen.

3.2. Agentensimulationssysteme und Skalierbarkeit

Der zentrale Inhalt der Forschungsfragen und damit dieser Arbeit betrifft die Konstruktion skalierbarer Agentensysteme. Aus diesem Grund liegt es nahe, die bestehenden Arbeiten und Ergebnisse in genau diesem Bereich für eine Betrachtung des Stands der Forschung heranzuziehen. Dazu werden in diesem Abschnitt verschiedene Agentensysteme vorgestellt, welche in ihrem Rahmen entweder direkt Skalierbarkeitsfragen adressieren oder aber ausreichend groß und bekannt sind, dass sie in einer derartigen Analyse nicht fehlen sollten.

Das allgemeine Forschungsfeld der Agententechnik ist groß und diverse Konferenzen existieren. Die prominenteste Konferenz hierbei ist die AAMAS (International Joint Conference on Autonomous Agents and Multiagent Systems). Sonstige Konferenzen umfassen die PAAMS (Practical Applications of Agents and Multiagent Systems), PRIMA (International Conference on Principles of Practice in Multi-Agent Systems), EUMAS (European Workshop on Multi-Agent Systems) und weitere.

3.2.1. JADE (Bellifemine et al.)

(Bellifemine, Poggi und Rimassa, 2000) und (Bellifemine, Poggi und Rimassa, 1999) stellen mit JADE (Java Agent DEvelopment Framework) ein Java-basiertes Agentenframework vor. Die Gesamtkonstruktion ist dabei eher allgemein gehalten und auf Erweiterbarkeit ausgelegt. JADE setzt bereits die Anforderungen der FIPA um und erlaubt eine verteilte Ausführung des Agentensystems. Es existieren zahlreiche Erweiterungen, von denen einige im Kontext der Arbeit interessante, später separat vorgestellt werden. JADE ist eine Open Source Software. Die Projektwebsite war zum Zeitpunkt des Verfassens (November 2021) über mehrere Wochen nicht erreichbar, archivierte Kopien der Projektwebsite existieren[1]. Die letzte bekannte Version von JADE ist Version 4.5 vom Juni 2017. Eine offizielle Einstellung des Projekts ist nicht bekannt und erscheint aufgrund der weiten Verbreitung von JADE als unwahrscheinlich.

Während JADE als agentenbasiertes verteiltes System konzipiert ist, spielen (formale) Modelle als Basis keine Rolle. Ebenso adressiert JADE keine (echte) Nebenläufigkeit, keine Ausnutzung bzw. Differenzierung von lokalen und verteilten Operationen, sowie keine Form der Plattformverwaltung.

[1] https://web.archive.org/web/20211022044205/https://jade.tilab.com/ - Zuletzt abgerufen am 14.11.2021

3.2.2. Elastic JADE (Siddiqui et al.)

(Siddiqui u. a., 2012) untersuchen die Verschiebung eines JADE Deployments in eine Amazon EC2 Cloudumgebung. Dabei kommen Autoskalierungsmechaniken durch die Plattform zum Einsatz. Skalierungsmechaniken selbst werden somit nicht durch die Publikation adressiert, ebenso wird keine Architektur für die Ausführung in der Cloud beschrieben. Analog zu JADE wird durch diese Arbeit die Nebenläufigkeit nicht explizit adressiert. Ähnliches gilt für die Unterscheidung lokaler und verteilter Aspekte. Durch den Einsatz der Amazon Cloudtechnik kann argumentiert werden, dass zumindest eine grundlegende Plattformverwaltung beschrieben wird. Diese entstammt jedoch nicht der Publikation und steht in keiner direkten Beziehung zu den simulierten Agenten.

3.2.3. Hsieh

Der Artikel (Hsieh, 2018) (»Design of scalable agent-based reconfigurable manufacturing systems with Petri nets«) ist von Interesse, da er viele der für die Arbeit relevanten Themen aufgreift: Skalierbarkeit, Agentenorientiertheit und Petrinetze. Die Untersuchungen umfassen die Modellierung von konfigurierbaren Produktionssystemen mit petrinetzbasierten Agenten. Die Skalierbarkeit bezieht sich dabei jedoch auf die Durchsatzfähigkeit des beschriebenen Produktionssystems, nicht jedoch auf das Agentensystem selbst. Eine Unterscheidung zwischen lokaler Verarbeitung und entfernter Verarbeitung wird nicht explizit adressiert. Die Untersuchungen und Modellierung als Petrinetz wird in ein Optimierungsproblem umkonstruiert, welches durch die Produktionssteuerung kontinuierlich gelöst wird. Ein (aktives) Management von Plattformen ist an keiner Stelle Gegenstand der Untersuchung. Eine prototypische Umsetzung in JADE wird diskutiert.

Insgesamt präsentiert sich die Arbeit als Anwendungspapier für petrinetzbasierte Agentenmodellierung. Ein perspektivisches Anknüpfen für die Konstruktion einer Architektur zur Beantwortung der Forschungsfragen dieser Arbeit gestaltet sich daher als schwierig.

3.2.4. cloneMAP (Dähling, Happ et al.)

cloneMAP (Dähling, Razik und Monti, 2021) beschreibt eine zeitgemäße Implementation eines Cloud-basierten Multiagentensystems. Das System kommt auch bei anderen Arbeiten der Autoren zum Einsatz (Happ, Dähling und Monti, 2020). Die zentralen technischen Stützpfeiler des Aufbaus umfassen die Containerisierungslösung Docker, den Containerorchestrator Kubernetes[2] und die ver-

[2]Eine detailliertere Einführung von Kubernetes erfolgt später in Abschnitt 14.1.3

teilte Datenbank Cassandra. Die Architektur entspricht im Wesentlichen den FIPA Vorgaben und der Implementation in JADE.

Die zentrale Idee bei dem System besteht im Einsatz der zentralisierten und in sich skalierbaren verteilten Datenbank Cassandra. Alle Berechnungseinheiten selbst sind dabei zustandslos. Die Umsetzung entspricht im Kern den Überlegungen des Prototypen der replizierten Netzdatenbank, wie er später in Abschnitt 14.2.1 beschrieben werden wird. Grundlegende Konzepte hierzu wurden vom Autor dieser Arbeit ebenfalls in (Röwekamp, Feldmann u. a., 2019) vorgestellt.

Skalierbarkeitsfragen werden adressiert, jedoch wird die Skalierbarkeit auf bestehende automatische Skalierbarkeitslösungen abgewälzt und erscheint transparent für die Agenten. Nebenläufigkeitsaspekte werden ebenfalls nicht explizit adressiert. Durch den Einsatz der zentralen Datenbank besteht ebenfalls keine Unterscheidung lokaler und verteilter Prozessanteile.

3.2.5. Pawlaszczyk et al.

Pawlaszczyk betrachtet im Wesentlichen Skalierbarkeitsfragen in getakteten Multiagentenanwendungen (Timm und Pawlaszczyk, 2005; Pawlaszczyk und Timm, 2006; Petsch, Pawlaszczyk und Schorcht, 2007; Pawlaszczyk und Strassburger, 2009). Die zugehörige Dissertation Pawlaszczyks beschreibt einen »*parallele[n] Simulator für die Simulation nachrichtenbasierter Agentenmodelle [...], der zur Untersuchung der Leistungsfähigkeit verteilter optimistischer Simulation genutzt werden kann*« (Pawlaszczyk, 2009, S.285). Der namenlose Simulator setzt auf der JADE Plattform auf und adressiert vor allem Synchronisationsfragen. Die Neuerung durch Pawlaszczyk umfasst dabei eine Begrenzung optimistischer Abläufe innerhalb der Simulation. Die Modellierung echter Nebenläufigkeit wird nicht adressiert, sondern sogar explizit eine Taktung vorausgesetzt. Skalierbarkeitsfragen reduzieren sich auf statisch zu Beginn der Simulation bereitgestellte Agentenmengen. Eine Beschreibung der Agentenplattform als aktive Einheit oder die Erzeugung von Plattformen ist nicht Teil der Ergebnisse.

3.2.6. MMAS2L (Murakami et al.)

(Murakami u. a., 2018) stellen die zweischichtige Architektur MMAS2L vor, um Agentensysteme verteilt auszuführen. Dabei existieren sogenannte Micro- und Macroagenten. Macroagenten verbleiben auf ihrem physikalischen Knoten und koordinieren Microagenten. Macroagenten müssen manuell auf physikalischen Knoten installiert werden. Die Lösung adressiert die verteilte Ausführung von interagierenden Agenten, arbeitet jedoch nicht modellbasiert und thematisiert keine explizite Nebenläufigkeit (in Abgrenzung zur Parallelität). Durch das Konzept der

Macroagenten werden Plattformen adressiert, jedoch erfolgt keine Kontrolle der Plattformen, da diese händisch bereitgestellt werden müssen. Weitere verwandte Arbeiten der Autoren umfassen die Artikel (Lhaksmana, Murakami und Ishida, 2018) und (D. Lin, Murakami und Ishida, 2018).

3.2.7. DSEJAMON (Bosse et al.)

Die DSEJAMON Architektur (Bosse, 2021) beschriebt ein verteiltes Agentensystem auf der Basis lose gekoppelter virtueller Maschinen. Technisch basiert es auf der JavaScript Agent Machine (»JAM«) des Autors (Bosse und Engel, 2019). Der Fokus wird auf die Unterscheidung von verteilten und lokalen Aspekten, sowie die Mobilität der Agenten und den Einsatz von JavaScript gelegt. Auf diese Weise soll eine hohe Flexibilität erreicht werden.

Das System setzt eine zentrale Taktung ein, adressiert Nebenläufigkeit somit explizit nicht. Auch eine aktive Plattformverwaltung wird nicht diskutiert, insbesondere nicht unter der Einflussnahme der Agenten. Die Konzeption erfolgt in direkter Übersetzung in Programmcode, sodass keine zwischengelagerte Modellierung zum Einsatz kommt.

3.2.8. Weitere erwähnenswerte Arbeiten

Weitere Arbeiten, welche sich thematisch in der Umgebung des Forschungskontextes bewegen, jedoch keine umfassenden Lösungsansätze für die Forschungsfragen darstellen, werden an dieser Stelle aufgegriffen.

Jason (Bordini, Hübner, et al.) und LightJason (Müller et al.)

Jason (Bordini, Hübner und Vieira, 2005) ist ein Interpreter einer erweiterten Fassung der Sprache »AgentSpeak(L)«. Der Fokus liegt hierbei auf der Interpretation der Sprache und weniger auf dem gesamten Deployment. Für eine Umsetzung müssen entsprechende Umgebungen selbst implementiert werden. Die aktuellste Fassung von Jason wurde 2021 veröffentlicht und ist über die Projektwebsite[3] verfügbar.

Durch diese Eigenschaften ist Jason nicht alleinig für den intendierten Einsatzzweck im Rahmen der Arbeit geeignet. Es existiert jedoch eine Neuinterpretation und -implementation unter dem Namen LightJason (Aschermann u. a., 2018), welche von Jason inspiriert wurde und mit hoher Skalierbarkeit und Nebenläu-

[3] http://jason.sourceforge.net/wp/ - Zuletzt abgerufen am 25.11.2021

figkeit wirbt, sodass eine entsprechende Evaluation angebracht ist. Die Website des Projekts[4] führt letzte Neuigkeiten und Publikationen aus dem Jahr 2018 auf.

Während die Skalierbarkeit einen der zentralen Aspekte darstellt, stellen Publikationen, Projektwebsite, Tutorialvideos und weitere Quellen der Autoren keine Informationen bereit, wie eine skalierbare Architektur umgesetzt wurde. Einzelne Architekturdiagramme legen jedoch nahe, dass keine Implementation als verteiltes System erfolgte. Der Fokus des Projektes liegt ähnlich wie der von Jason auf dem Einsatz der Sprache »AgentSpeak(L++)«. Parallele Operationen müssen vom Modellierer explizit angegeben werden. Eine Modellbasis und das Deployment als verteiltes System fehlen und damit ebenfalls alle Betrachtungen von lokaler und verteilter Ausführung sowie Plattformverwaltung.

Casquero et al.

Die Arbeit (Casquero u. a., 2019) betrachtet verteiltes Scheduling in Kubernetes auf Basis eines Multiagentensystems und JADE (Java Agent Development Framework) für Fog-in-the-Loop (FIL) Anwendungen. Die Arbeit umfasst damit Überschneidungen mit Skalierbarkeitsfragen, agentenorientierter Softwareentwicklung und lokalen und verteilten Systemen. Der Fokus liegt dabei jedoch auf dem Einsatz eines Agentensystems für die Verteilung von containerisierter Software und nicht bei der Skalierung des Agentensystems selbst. Aus diesem Grund passt die Arbeit nicht zum hier intendierten Einsatzzweck.

Gracia et al.

Ähnlich wie (Casquero u. a., 2019) stellt die Arbeit (Gracia, Rana u. a., 2016) einen Scheduler für Kubernetes Cluster auf der Basis von Referenznetzen vor. Die Nutzung modellbasierter nebenläufiger Abläufe liefert hierbei Vorteile gegenüber dem mitgelieferten Scheduler von Kubernetes. Die Umsetzung betrifft keine Agenten und die Steuerung erfolgt unabhängig von der im Cluster ausgeführten Software. Da die Rückkoppelung aus der Anwendung nicht modelliert ist, scheidet die Arbeit als mögliche Umsetzung der Forschungsfragen aus. (Gracia, Arronategui u. a., 2019) setzen Petrinetz-Modelle ein, um die gemeinsame Ressourcennutzung zweier Anwendungen auf demselben physikalischen Knoten innerhalb eines Kubernetes Clusters zu verbessern.

[4] https://lightjason.org/ - Zuletzt abgerufen am 24.11.2021

Agentcities und OpenNet

»*Agentcities*« (Constantinescu, Willmott und Dale, 2003) war ein EU-gefördertes Projekt für die Vernetzung verschiedener FIPA-konformer Agentenplattformen. Das System sah einen zentralen Verzeichnisdienst vor, an welchem sich Agenten und Plattformen anmelden konnten. Aufgrund der Zentralisierung stieß das Projekt jedoch an Grenzen und ging in das Nachfolgeprojekt »*OpenNet*« über. Die letzten Einträge auf den Websites von Agentcities stammen aus dem Jahr 2004[5] vor der Abschaltung der Projektwebsite im Oktober 2007.

Der Nachfolger *OpenNet* sollte mit seiner Architektur in erster Linie Skalierbarkeit adressieren. Im Gegensatz zu Agentcities erhielt das Projekt keine EU-Förderung mehr. Großflächige Beteiligung an dem Projekt erfolgte nicht (Reese, 2009, S.40). Der letzte bekannte Ausschnitt der Projektwebsite[6] stammt aus dem Jahr 2009. Die Skalierbarkeit in OpenNet adressierte im Wesentlichen die Verzeichnisdienste.

Beide Architekturen stellen nur die Verwaltungsinfrastruktur bereit und nehmen an, dass manuell neue Plattformen global eingebunden werden. Somit nehmen sie keine eigene Skalierung vor. Da zusätzlich beide Projekte eingestellt wurden, erfolgt im Rahmen dieser Arbeit keine weitere Betrachtung. Sie stellen jedoch große Agentenprojekte dar, welche auch den Skalierbarkeitsaspekt adressierten und sollten daher an dieser Stelle genannt werden.

3.3. Relevante Arbeiten an der Universität Hamburg

Neben den beschriebenen relevanten internationalen Arbeiten lohnt sich ein Blick auf die Veröffentlichungen von Arbeitsgruppen im Kontext der Universität. Der Fokus liegt dabei nach wie vor auf solchen Arbeiten, welche zu den Forschungsfragen ähnliche Thematiken adressieren.

3.3.1. Mulan (Rölke, Köhler, Moldt et al.)

Mulan ist eine Referenzarchitektur[7] für die Modellierung von Multiagentensystemen mit Petrinetzen nach der Spezifikation der Foundation for Intelligent Physical Agents (FIPA). Da sich in diesem Kapitel und in Kapitel 4 ergeben wird,

[5] https://web.archive.org/web/20071012220347/http://www.agentcities.org:80/ - Zuletzt abgerufen am 30.11.2021

[6] https://web.archive.org/web/20090223073923/http://x-opennet.org/ - Zuletzt abgerufen am 30.11.2021

[7] vgl. zum Begriff »Referenzarchitektur« beispielsweise (Lilienthal, 2008)

dass Mulan als Ausgangspunkt für die Konzeptualisierung der Architektur im Rahmen der Arbeit eingesetzt werden wird, soll eine deutlich detaillierte Beschreibung der Architektur im Vergleich zu den anderen zitierten Arbeiten bereits an dieser Stelle erfolgen.

Mulan wurde erstmals zur Jahrtausendwende von Rölke, Köhler und Moldt beschrieben (Rölke, 1999; Köhler, Moldt und Rölke, 2001) und ist Gegenstand der Dissertation (Rölke, 2004). Einen umfangreichen Einstieg bietet beispielsweise die Arbeit (Cabac, 2010). Eine generelle Einführung in die agentenorientierte Programmierung mit Mulan ist in (Cabac, Haustermann und Mosteller, 2018) zu finden.

Eine Erweiterung von Mulan umfasst die Concurrent Agent Platform Architecture Capa (vgl. (Duvigneau, Moldt und Rölke, 2003)), welche Kommunikation zwischen Agenten nach FIPA-Standard ermöglicht. Aktuelle Arbeiten im Kontext sind beispielsweise die Erweiterung der Settler Anwendung auf der Basis von Mulan (Beese, 2021) sowie die Modularisierung (Janneck, 2021) der Mulan Implementation.

Im Kontext dieser Arbeit wird sich die grundlegende Architektur, welche in Mulan zum Einsatz kommt, als zentraler Stützpfeiler erweisen. Daher wird in dieser Einführung auf diese ein besonderer Fokus gelegt, während die Implementationsdetails auf niedrigerer Ebene in geringerer Tiefe behandelt werden. Die Mulan-Architektur basiert in erster Linie auf der Einheitentheorie, welche bereits in Abschnitt 2.5.3 eingeführt wurde. Die letzte Neuerung der Implementation von Mulan umfasst die Einführung von Modularisierung im Jahr 2021/2022 insbesondere im Rahmen des im Wintersemester durchgeführten Lehrprojekts am Arbeitsbereich ART an der Universität Hamburg.

Die Architektur von Mulan

Mulan verwendet als Grundlage den Referenznetzformalismus. Durch den Einsatz von Petrinetzen kann zunächst ein grobes Modell der Funktionalität geschaffen werden, welches dann sukzessive und iterativ verfeinert werden kann. Mulan sieht im Wesentlichen vier Ebenen vor: Infrastruktur, Plattform, Agent und Protokoll. Jede dieser Ebenen ist eine Einheit im Sinne der Einheitentheorie und spezifiziert sich durch ihre Interaktionsmöglichkeiten zwischen Inkarnationen der Ebene, ihrer übergeordneten Ebene und den ihr untergeordneten Elemente. Dabei sind die Interaktionen oberhalb der Infrastruktur und unterhalb der Protokolle nur im Ansatz beschrieben. Abbildung 3.1 zeigt eine grafische Darstellung der Ebenen der Mulan-Architektur

Infrastruktur (Ebene 1) Der Begriff der Infrastruktur ist in der Mulan -Architektur sehr generell gehalten. Es existiert neben den einzelnen Plattformen

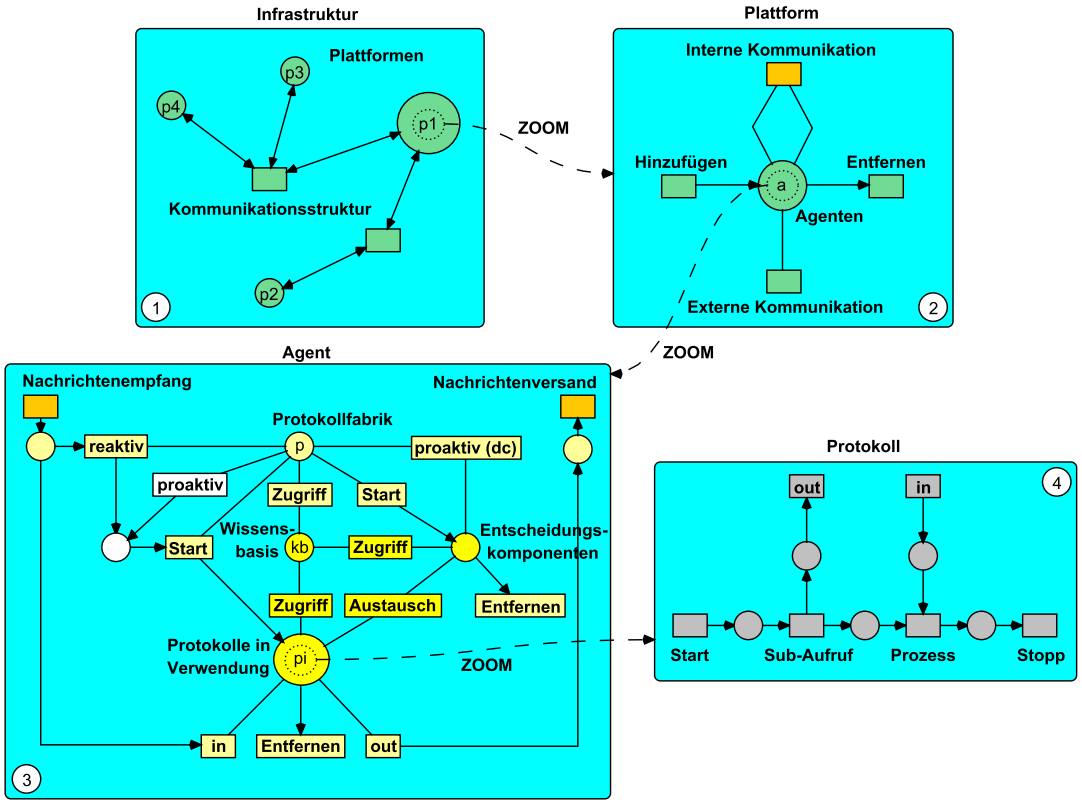

Abbildung 3.1.: Die Architektur von MULAN nach (KÖHLER, MOLDT und RÖLKE, 2001, Seite 227), jeodch mit ausgetauschtem Agentenmodell inklusive Entscheidungskomponenten nach (CABAC, DÖRGES, DUVIGNEAU, REESE u. a., 2007, Seite 148)

auf der Infrastruktur, welche durch einzelne Plätze dargestellt werden, eine Kommunikationsstruktur, welche durch Transitionen symbolisiert wird. Dabei können einzelne Plattformen miteinander in Kommunikation treten, dargestellt durch rückgerichtete Kanten innerhalb der Infrastrukturdarstellung. Die intendierte Interpretation umfasst reale Orte und Plattformen bzw. physikalische Computer. Da die Infrastrukturebene von MULAN die höchste betrachtete Einheit ist, wird keine übergeordnete Einheit beschrieben. Ebenso wird keine direkte Existenz weiterer Infrastrukturen angenommen. Sie definiert sich damit primär durch die Bereitstellung von der Kommunikationsstruktur für ihre untergeordneten Plattformen und die statische Existenz eben dieser.

Plattform (Ebene 2) Die Plattform bettet sich in eine Infrastruktur ein. Sie beherbergt Agenten und ermöglicht diesen die Kommunikation mit anderen Plattformen und realisiert so die Interaktion mit ihrer übergeordneten Einheit. Da bei solch einer Kommunikation ebenso andere Plattformen eingebunden sind, besteht somit eine Kommunikation mit anderen Plattformen. Plattformen beschränken sich auf diese Form der Kommunikation untereinander. Darüber hinaus ist eine Plattform in der Lage, Agenten zu erzeugen und zu entfernen sowie diesen interne Kommunikation zu ermöglichen. So interagiert die Plattformen mit ihren untergeordneten Agenten.

Agent (Ebene 3) Die Agenten in MULAN sind – wenig überraschend – die am weitesten ausdefinierten Einheiten. Sie sind stets in eine Plattform eingebettet und können über diese mit anderen Agenten kommunizieren, indem sie Nachrichten empfangen oder versenden. Agenten besitzen eine Wissensbasis und eine Reihe an Protokollen, welche ihr Verhalten bestimmen. Darüber hinaus sind sie in der Lage, reaktiv (Transition **re** in der Abbildung) und proaktiv (Transition **pro** in der Abbildung) zu handeln. Protokolle, welche aktuell ausgeführt werden, können Nachrichten aufnehmen oder versenden. Nicht mehr benötigte verwendete Protokolle können bei Bedarf entfernt werden. Durch diese Eigenschaften definiert der Agent die Kommunikation mit seinen Protokollen und damit sein gesamtes Verhalten. In der Literatur sind die Eigenschaften der Wissensbasis, Initialisierung, Interaktion mit Protokollen und weiteren Elementen detailliert beschrieben. Wie eingehend erläutert, soll in dieser Arbeit jedoch eine weniger tiefgreifende Vorstellung dieser unteren Bestandteile genügen. Für weitere Erläuterungen sei daher auf die entsprechenden Literaturquellen (KÖHLER, MOLDT und RÖLKE, 2001; RÖLKE, 2004; CABAC, 2010) verwiesen.

Protokoll (Ebene 4) Ein Protokoll bettet sich in einen Agenten ein und beschreibt einen isolierten Ablauf. In der Abbildung ist eine beispielhafte Struktur des Protokolls aufgeführt. Sie können durch die als **in** und **out** gekennzeichneten Transitionen Nachrichten absenden oder auf den Eingang

bestimmter Nachrichten warten. Unterhalb der Protokolle sind Subprotokolle eingebettet, welche durch das Protokoll angesteuert werden können. Wie zuvor erwähnt, ist diese Beziehung nicht mehr detailliert definiert. Protokolle werden von den zugehörigen Agenten für die Verwendung gestartet und auch wieder entfernt.

Capa

Das Akronym Capa steht für *Concurrent Agent Platform Architecture* und bezeichnet die FIPA-konforme Implementation der Mulan-Architektur auf der Basis von Referenznetzen. Im Vordergrund steht dabei die Bereitstellung einer speziellen Plattform, bzw. eines Plattformagenten. Capa wurde in (Duvigneau, 2002; Duvigneau, Moldt und Rölke, 2002) vorgestellt.

Nennenswerte Abwandlungen von der Mulan Referenzarchitektur umfassen unter anderem die Umsetzung von FIPA-konformen Agenten-Lebenszyklen anstatt einfacher Erzeugung und Destruktion durch die Plattform. Darüber hinaus können mittels Capa andere FIPA-konforme Plattformen angeschlossen werden, wodurch insgesamt ein verteiltes System realisiert werden kann. In der Capa Plattform realisierte Bestandteile umfassen alle der in Abschnitt 2.4.2 genannten FIPA Komponenten.

Für diese Arbeit relevante Eigenschaften ergeben sich aus Capa im Rahmen der Grundidee Agentenplattformen selbst wie Agenten zu behandeln. Dieser Zusammenhang kann sich ebenfalls als Implikation aus der Einheitentheorie ergeben. Diese Idee wird im Rahmen des Konstruktionsansatzes am Ende der Anforderungsanalyse in Abschnitt 4.4 noch einmal detaillierter aufgegriffen.

Verzeichnisdienste in Agentcities und OpenNet

Im Laufe der Zeit wurden für die Capa Plattform Anbindungen an die mittlerweile eingestellten Großprojekte Agentcities (Reese, 2003) sowie an den Nachfolger OpenNet (Laka, 2007) implementiert. Diese Untersuchungen betreffen dabei im Kern die Aspekte der Verzeichnisdienste in den beiden Projekten und deren Abbildung auf Mulan/Capa Agenten.

WebGateway

Eine für den Rahmen dieser Arbeit relevante Vorarbeit ist die Einführung webbasierter Agentendienste im Kontext von Mulan in (Betz, 2011). Der Fokus liegt hierbei auf der Integration von Webservice-basierten Technologien mit Agenten-

technik. Das zentrale Ergebnis der Arbeit ist die Beschreibung einer entsprechenden Architektur und die Umsetzung des »Mulan WebGateways«.

Die grundlegenden Überlegungen sind hierbei nützlich für die spätere Umsetzung, die Arbeit adressiert jedoch im Kern die Kommunikationsform und -syntax von Agenten. In späteren Untersuchungen wird ersichtlich werden, dass eine Realisierung einer skalierbaren Agentensimulation im Sinne der Forschungsfragen eine Adressierung in einem größeren Rahmen erfordert, als es durch die Einschränkung auf nur die Kommunikationsebene der Agenten möglich wäre.

Insgesamt liefert die Arbeit somit konzeptuell eine wichtige Vorarbeit bezüglich der Agentenkommunikation, welche es ermöglicht, detailliertere Betrachtungen der Übersetzung von FIPA-konformen Nachrichten auf Webtechnologien zu überspringen.

3.3.2. Jadex (Pokahr, Braubach et al.)

Active Components Jadex entstand zunächst als BDI (Belief Desire Intention) Erweiterung für das JADE Framework (Braubach, Lamersdorf und Pokahr, 2003; Pokahr, Braubach und Lamersdorf, 2005), ist jedoch mittlerweile ein eigenständiges Projekt (Pokahr und Braubach, 2009). Das Projekt wird in den Dissertationen der beiden Hauptautoren (Braubach, 2007; Pokahr, 2007) ebenfalls beschrieben. Die wesentlichen Eckpunkte umfassen hierbei den Einsatz eines zentralisierten Taktgebers und die Unterstützung von sogenannten »virtuellen Umgebungen«. Diese virtuellen Umgebungen weisen Ähnlichkeiten zum Plattformverständnis dieser Arbeit auf, jedoch erfordert die Konstruktion die Bereitstellung durch einen Entwickler (Braubach und Pokahr, 2013). Die zum Zeitpunkt des Verfassens aktuelle Version 4.0.250 von Jadex ist auf der Projektwebsite[8] verfügbar.

3.3.3. Orestes/Baqend (Gessert, Wingerath, Ritter et al.)

Orestes und das dazugehörige kommerzielle Pendent Baqend[9] beschreiben eine Architektur für horizontal skalierbare Datenbanksysteme. Die wichtigsten Literaturquellen umfassen neben zahlreichen Papieren die Veröffentlichungen (Wingerath, 2019), (Gessert, 2019) sowie (Gessert, Wingerath und Ritter, 2020). Während Agententechnik keine zentrale Rolle in den Betrachtungen spielt, sind jedoch die Skalierbarkeitsaspekte interessant im Kontext dieser Arbeit.

[8] https://www.activecomponents.org/ - Zuletzt abgerufen am 08.11.2021
[9] https://www.baqend.com/ - Zuletzt abgerufen am 23.11.2021

Orestes realisiert eine Plattformverwaltung durch lastbasierte Skalierung von containerisierten Anwendungsteilen und setzt damit den Autoskalierungsansatz um. Die Beschreibung der Architektur ist nah an Webtechnologien gehalten und überspringt daher die explizite Modellierung durch ein Modell auf Formalismusbasis wie beispielsweise Petrinetze. Ebenso ist die echte Nebenläufigkeit nicht im Fokus der Untersuchungen, Caching fördert eine grundlegende Unterscheidung zwischen lokalen und verteilen Operationen.

3.3.4. MARS (Glake, Weyl, et al.)

(Glake, Panse u. a., 2021) beschreiben Datenmanagement in heterogenen Multiagentenumgebungen im Simulationssystem MARS (»Multi-Agent Research and Simulation«) (Glake, Weyl u. a., 2017; Weyl u. a., 2019). Der Fokus liegt hierbei auf der Datenhaltung und der Abbildung räumlich-zeitlicher (»spatiotemporal«) Eigenschaften im System, wie sie beispielsweise in der Simulation von Verkehrssystemen relevant sind. Das System setzt hierzu analog zu anderen Arbeiten der Autoren (Glake, Kiehn u. a., 2021) einen sogenannten »Polystore« ein, eine Integration verschiedener Datenbanktypen mit den jeweiligen spezifischen Anfragesprachen.

MARS ist als cloud-native Anwendung entworfen und nutzt bestehende Skalierungsmöglichkeiten (Glake, Ritter und Clemen, 2020). Die Betrachtungen gründen auf der Basis von diskreten »Ticks« (Zeitschritten) und schließen somit echte Nebenläufigkeit zugunsten von getakteter Parallelität aus. Konsistenz im System wird stets nach einem abgelaufenen Tick hergestellt. Während die Architektur des Datenmodells ausführlich beschrieben ist, existiert keine Beschreibung des Gesamtsystems und des Deployments in einer Modellierungssprache, welche als Basis für eine Implementation eingesetzt wurde.

3.3.5. Weitere Arbeiten an der Universität Hamburg

In diesem Abschnitt werden noch einmal die wichtigsten Vorarbeiten nach Autoren sortiert knapp adressiert. Die Referenzen werden zum überwiegenden Teil im Verlauf der Arbeit an den für sie relevanten Stellen erneut aufgegriffen. Dieser Abschnitt soll dem Zweck dienen eine Vorstellung der Bedeutung der Forschungsgebiete der jeweiligen Autoren im Kontext der Arbeit zu vermitteln.

Die zitierten Arbeiten werden nicht streng chronologisch aufgeführt, da sich die Abläufe mitunter thematisch überschneiden. Stattdessen wird sich jeder Absatz auf eine Thematik beziehen und mit der/den zentralen Arbeit(en) beginnen. Gegebenenfalls folgen dann Vor- und Nacharbeiten zu dem Thema in chronologischer Reihenfolge.

Hauschildt

Hauschildt beschreibt eine der frühen ersten verteilten Implementationen von (gefärbten) Petrinetzen (Hauschildt, 1987). Die Hauptneuerung besteht in der Modellierung von Plätzen als Prozesse, welche den konfliktbehafteten Zugriff auf Marken durch mehrere Transitionen intern auflösen. Die Arbeit stammt aus einer Zeit vor allen sonstigen in dieser Arbeit betrachteten Lösungen und Vorarbeiten und bildet für viele dieser ihrerseits eine Vorarbeit.

Valk

Die für diese Arbeit zentralen Konzepte liefert Valk in der Form von Objektnetzen, speziellen höheren Petrinetzen und »Netzen-in-Netzen«, bei denen Marken selbst wiederum Petrinetze sein können (Valk, 1991, 1995, 1998). Dabei legt Valk den Grundstein für die Abbildung hierarchischer Zusammenhänge in Systemen durch höhere Petrinetze und der Idee von Netzinstanzen.

Kummer

Der zentrale Beitrag Kummers umfasst die formale und umfassende Erschaffung und Beschreibung von Referenznetzen (Kummer, 2002). Zugehörige Vorarbeiten umfassen die Arbeiten (Kummer, Moldt und Wienberg, 1999; Kummer, 2000, 2001), in denen unter anderem formale Ergebnisse wie die Unentscheidbarkeit einiger Fragestellungen im Kontext von Referenz- und Objektnetzen adressiert werden. Referenznetze vereinen die Konzepte synchroner Kanäle und der Objektnetze. Sie wurden detailliert in Abschnitt 2.3 eingeführt.

Der Simulator Renew wird in (Kummer, Moldt und Wienberg, 1998; Kummer und Wienberg, 1999a; Kummer, 2002; Kummer, Wienberg, Duvigneau, Schumacher u. a., 2004) beschrieben, sowie dessen aktuellste Fassung in (Kummer, Wienberg, Duvigneau, Cabac u. a., 2020b).

Moldt

Moldt liefert vielfältige Vorbedingungen für verschiedene Konzepte, welche in der Arbeit aufgegriffen werden. (Moldt, 1996) beschreibt den Einsatz höherer Petrinetze zur Spezifikation von Systemen, bei dem der Einsatz des Netzformalismusses in den Gesamtprozess der Softwareentwicklung eingeordnet wird. Neben der Monografie existieren zu dem Ansatz unter anderem auch die weiteren Veröffentlichungen (Becker und Moldt, 1993a,b), (Maier und Moldt, 1997) und (Maier und Moldt, 2001). In letzterer Arbeit wird ebenfalls der Einsatz syn-

chroner Kanäle zur Kommunikation nach (S. Christensen und Hansen, 1994) adressiert.

In (Moldt, 2005) wird die Einheitentheorie beschrieben, welche zuvor bereits indirekt in Mulan (Köhler, Moldt und Rölke, 2001) genutzt wurde. Als Vorarbeit wurde die Anwendung von gefärbten Petrinetz-Modellen auf Agentensysteme in (Moldt und Wienberg, 1997) beschrieben. Moldt liefert somit elementare Anteile zu den Konzepten der Referenznetze, der Multiagentenarchitektur Mulan, der Einheitentheorie und weiteren Aufbauten auf diesen Konzepten von den 1990er-Jahren bis heute.

Weitere Arbeiten umfassen die Entwicklung von Workflownetzen auf der Basis von Netzen-in-Netzen (Aalst u. a., 1999), die Beschreibung von Webservices durch Mulan (Moldt, Ortmann und Offermann, 2004) und (Moldt, Offermann und Ortmann, 2005), die Integration von Agenten und einer Plugin-Architektur (Cabac, Duvigneau, Moldt und Rölke, 2005, 2006; Cabac, Duvigneau, Moldt und Schleinzer, 2007; Schleinzer u. a., 2008) sowie die Integration Agenten und Workflows (Moldt, Quenum u. a., 2010),(Wagner, Quenum u. a., 2012) und (Wagner und Moldt, 2015).

Köhler

Köhler liefert wichtige Erkenntnisse zu Objektnetzen (Köhler, 2004; Köhler-Bussmeier, 2014).

Weitere Arbeiten behandeln die Konstruktion von Multiagenten- sowie Multiorganisationssystemen insbesondere unter einer sozialwissenschaftlichen Perspektive und mit Fokus auf reflexive Selbstorganisation (Köhler-Bussmeier, 2009). Nennenswerte Vorarbeiten sind hierbei (Köhler, Moldt und Rölke, 2001) sowie (Köhler, 2007) und (Köhler-Bussmeier und Wester-Ebbinghaus, 2009b) in denen unter anderem mit »Sonar« eine formale Grundlage für die Beschreibung von Organisationsmodellen auf Agentenbasis geliefert wird.

Reese

Reese lieferte erste Erkenntnisse zur Integration von Agenten und Workflows unter Adressierung von heterogenen Systemen und Prozessmodellierung (Reese, 2009). Die Arbeit liefert im Wesentlichen Anknüpfungspunkte für Wagner und damit indirekt Vorarbeiten für diese Arbeit. Die Vorarbeiten zur Monografie Reeses umfassen (Reese, Ortmann u. a., 2005),(Reese, Markwardt u. a., 2006),(Reese, Wester-Ebbinghaus u. a., 2007) und (Reese, Wester-Ebbinghaus u. a., 2008).

Wester-Ebbinghaus

Wester-Ebbinghaus stellt als wesentliches Ergebnis die Multiorganisationsarchitektur Organ (Wester-Ebbinghaus, 2010) vor. Weitere Arbeiten in diesem Kontext umfassen unter anderem (Wester-Ebbinghaus und Moldt, 2008b), (Wester-Ebbinghaus und Moldt, 2008a) sowie (Köhler-Bussmeier und Wester-Ebbinghaus, 2009a). Organ ist oberhalb der Mulan-Plattformen verortet und daher durchaus von Interesse im Kontext der Arbeit, da ein zentraler Aspekt der Forschungsfragen der Einfluss durch Agenten auf Plattformen behandelt. Da die Überlegungen auf Mulan aufsetzen, sind sie nicht als separater Abschnitt innerhalb dieses Kapitels aufgeführt.

Cabac

Cabac erweitert Agenten im Sinne des Mulan-Modells mit der zugehörigen Arbeit (Cabac, 2010). Zentrale Ergebnisse umfassen den Einsatz von Agenteninteraktionsprotokollen (Cabac, Moldt und Rölke, 2003), die Generierung von Netzkomponenten aus diesen (Cabac, 2003), sowie die anteilige Einführung von Entscheidungskomponenten (Decision Components »DCs«) (Cabac, Duvigneau, Reese u. a., 2007), welche im Kern kommunikationsfreie domänenspezifische Verhaltensmuster von Agenten implementieren.

Bendoukha

Die Arbeiten Bendoukhas adressieren das Management verschiedener Cloudlösungen auf der Basis von Petrinetzen (S. Bendoukha, 2017). Relevante Vorarbeiten umfassen (S. Bendoukha und Wagner, 2012),(S. Bendoukha, 2013; S. Bendoukha, Moldt und Wagner, 2013) und (S. Bendoukha, H. Bendoukha und Moldt, 2015a,b). Aufgrund der Adressierung von Aspekten des Cloudcomputings sind die Arbeiten Bendoukhas wichtige Vorarbeiten für diese Arbeit. Abschnitt 2.8.8 gibt ebenfalls eine detailliertere Übersicht zu den Ergebnissen.

Wagner

Wagner beschreibt integrierte Einheiten als konzeptionelle Kombination aus Agenten (Struktur) und Workflows (Verhalten) als Basis für die Spezifikation von Systemen (Wagner, 2018). Für diese Arbeit ist insbesondere die Verbindung von Agenten und Workflows im Bezug auf die Kommunikation von Interesse (Wagner, 2009, 2012)(Wagner und Moldt, 2015). Weitere Vorarbeiten zur Monografie umfassen unter anderem (Wagner und Cabac, 2013),(Wagner,

Schmitz und Moldt, 2016) und (Wagner, Moldt und Köhler-Bussmeier, 2016).

Simon

Simon liefert mit (M. Simon, 2014) und (M. Simon und Moldt, 2016) eine (begrenzte) Erweiterung der Algorithmen für die Simulation synchroner Kanäle auf verteilte Fälle. Die Arbeit wird als Vorbedingung eine wichtige Rolle im Verlauf der Arbeit einnehmen und die Ergebnisse sind detailliert in Abschnitt 2.8.8 beschrieben. Aktuelle Forschungsarbeiten umfassen die Beschreibung funktionaler Sprachanschriften in höheren Petrinetzformalismen (M. Simon und Moldt, 2018; M. Simon, Moldt u. a., 2019).

Dreschler-Fischer

Dreschler-Fischer liefert im Wesentlichen Vorarbeiten bezüglich der Szenenanalysekomponenten der Arbeit und adressiert in vielen Arbeiten das Korrespondenzproblem der Bildverarbeitung[10]. Erste Arbeiten umfassen die volumetrische Beschreibung starrer Körper aus Bildfolgen (Dreschler, 1981). Nach weiteren Arbeiten zum Korrespondenzproblem (Leonie Dreschler-Fischer und Triendl, 1985) (Bartsch, Leonie Dreschler-Fischer und Schröder, 1986) folgten in jüngerer Zeit Untersuchungen anhand von Satellitenbildern und -bildfolgen (Martin Gade, Fiedler und Leonie Dreschler-Fischer, 2003) (Seppke, L. Dreschler-Fischer und Hübbe, 2010) (Seppke, L. Dreschler-Fischer, Heiming u. a., 2010) (Seppke, M. Gade und L. Dreschler-Fischer, 2010). Erwähnenswert ist in diesem Kontext auch die betreute Dissertation von Seppke (Seppke, 2013), welche Betrachtungen zum Korrespondenzproblem in Satellitenbildern behandelt.

3.4. Angrenzende Forschungsbereiche

Neben den direkt relevanten Architekturen und Autoren werden an dieser Stelle noch angrenzende Forschungsfelder aufgezeigt. Diese können interessante Sichtweise oder Gedankenanstöße zu den behandelten Thematiken liefern. Die Einträge hier sind somit nicht als vollwertige Lösungsmöglichkeiten für die Beantwortung der Forschungsfragen zu verstehen.

[10] Das Korrespondenzproblem beschreibt das fundamentale Problem der Zuordnung von verschiedenen Punkten in verschiedenen Bildern zu identischen Punkten/Objekten in der Realwelt.

3.4.1. Verteilte diskrete Eventsimulation

Eine diskrete Eventsimulation bezeichnet eine Simulation, welche durch Zustandsübergänge zu bestimmten diskreten Zeitpunkten geprägt ist. Außerhalb dieser Übergänge ist das System in seinen Eigenschaften unverändert. Ein derartiges System könnte beispielsweise durch Petrinetze und die darin enthaltenen Transitionen modelliert werden. Anders als Petrinetze sind diskrete Eventsimulationen klassischerweise sequenziell aufgebaut. Als Erweiterungen existieren daher parallele und verteilte diskrete Eventsimulationen. Zur diskreten Eventsimulation existieren verschiedene Konferenzen, wie beispielsweise die Konferenz ACM SIGSIM PADS (»Principles of Advanced Discrete Simulation«).

Verteilte Diskrete Eventsimulationen können auch für die Simulation von Agentensystemen eingesetzt werden, weshalb der Bereich an dieser Stelle erwähnenswert ist. Aktuelle Beiträge zu dem Thema umfassen (Clemen u. a., 2021) im Kontext des Bereits vorgestellten Simulationsframeworks MARS, sowie auszugs- und beispielsweise (Tan u. a., 2021), (Zia, Farooq und Ferscha, 2021) und (Chen u. a., 2020).

3.4.2. GALS Systeme

Das Akronym GALS steht für »Globally Asynchronous, Locally synchronous« (Global asynchron und lokal synchron) und ist primär im Kontext Hardwarenaher Forschung zu finden. Die überwiegende Menge an Ergebnissen zu dem Thema lässt sich dem Bereich der »Systems on Chip« (SoC) zuordnen. Obwohl das hier betrachtete Thema der Arbeit inhärent anders aufgestellt ist und sich auf architektureller Ebene bezogen auf Software beschäftigt, bieten diverse Ergebnisse aus dem Bereich der GALS-Systeme gute Anknüpfungspunkte, sodass der Bereich in jedem Fall Erwähnung finden soll.

Aktuelle Arbeiten umfassen exemplarisch die Interkonnektivität in GALS Systemen (Bertozzi u. a., 2021), die Optimierung des Energieverbrauchs (Weber u. a., 2020) und die Dissertation (Marsso, 2019) zum modellbasierten Testen in GALS-Systemen.

3.4.3. Infrastrukturmanagement

Insbesondere im Kontext der Plattformverwaltung ist ein Blick auf aktuelle Erkenntnisse aus dem Bereich des Infrastrukturmanagements hilfreich. Diese Untersuchungen beziehen sich zum überwiegenden Anteil auf Autoskalierungsansätze in verwalteten Rechenclustern. Auf verschiedene Weisen werden Anstrengungen unternommen, die gegenwärtige und zukünftige Auslastung konkreter physikali-

scher Systeme zu approximieren und auf dieser Basis eine Lastbalancierung durchzuführen. Die Untersuchungen haben zum überwiegenden Teil keinen Agentenkontext. (Shah und Dubaria, 2019) gibt im Bereich des Infrastrukturmanagements beispielsweise einen simplen Einstieg in die Anwendungsentwicklung mit den Technologien Docker, Kubernetes und der kommerziellen Google Cloud. Einzelne, spezielle und nah an der Thematik dieser Arbeit verortete Publikationen wurden auch bereits in Abschnitt 3.2.8 adressiert.

Verschiedene Arbeiten können hier Inspiration für diese Arbeit liefern. (Taherizadeh und Grobelnik, 2020) betrachten Schlüsselfaktoren um Autoscaler für Kubernetes Umgebungen zu designen. (Zhong und Buyya, 2020) stellen einen Kubernetes Scheduler für heterogene Ressourcen vor, welcher versucht die initiale Containerplatzierung, das Herunterfahren von VMs und das Autoscaling zu optimieren. Interessant ist hierbei der Fokus auf Heterogenität. (Balla, C. Simon und Maliosz, 2020) betrachten einen weiteren automatisierten Skalierungsalgorithmus namens Libra für Kubernetes. (Fu u. a., 2019) betrachten einen Scheduler für Kubernetes, welcher versucht Ressourcennutzung in der Zukunft vorherzusagen.

(Rattihalli u. a., 2019) beschreiben einen Umzugsservice in Kubernetes, welcher tatsächliche Ressourcennutzung überwacht und somit mehr Container auf einem physikalischen Rechner unterbringt. Dieser Umzugsservice könnte sich als nützlich für einen etwaigen Umzug von Plattformen erweisen.

Jedoch existieren auch anwendungsorientiertere Arbeiten. Exemplarisch beschreiben (Jin u. a., 2019) die Steuerung von CNC[11]-Systemen mittels gefärbten Petrinetzen. Eingesetzt wird ein Kubernetes Cluster aus CNC-Steuerungen in Docker Containern.

3.4.4. Cloud-Fog-Computing

Primär im Kontext von Internet of Things (IoT) Anwendungen verortet sich das Cloud-Fog- bzw. auch Fog-Cloud-Computing. Fog-Computing bezeichnet die Vorverarbeitung von Daten nah an den Datenquellen in einem dezentralen, aber physikalisch nahen Rechnernetz. Metaphorisch senkt sich die Wolke (Cloud) herab und liegt näher an den Datenquellen (Boden) und wird somit zum Nebel (Fog). Werden zusätzlich Clouddatenspeicher und/oder -services eingebunden wird vom Cloud-Fog-Computing gesprochen. Eingesetzt wird das Paradigma vor allem in der Industrie, bei der Standorte große Datenmengen generieren, aber ein direkter Upload in eine Cloud aufgrund mangelnder Infrastruktur nicht möglich ist. Ande-

[11] Computergestütze Steuerungstechnik. Wörtlich: »Computerized Numerical Control« (dt.: Rechnergestützte numerische Steuerung)

re Anwendungsfelder wie beispielsweise Medizintechnik werden aber auch durch die Forschung adressiert. Das Forschungsfeld ist vergleichsweise jung.

Im Kontext der Arbeit ist der Aspekt der gezielten Trennung von lokaler und entfernter Verarbeitung von Interesse. Das Forschungsfeld ist aktiv und aktuelle Arbeiten umfassen beispielsweise die Optimierung des Energieverbrauchs (Arshed und Ahmed, 2021) (Aldossary und Alharbi, 2021), medizinische Anwendungen für die Steuerung von medizinischen Instrumenten bei Operationen (Sedaghat und Jahangir, 2021), Sicherheitsaspekte (X. Zhang und Si, 2021) (Rangisetti, Dwivedi und Singh, 2021) und weitere. Auch einige agentenbasierte Verfahren existieren (Bulla und Birje, 2021) (Mutlag u. a., 2021). (Fellir u. a., 2020) beschrieben ein agentenbasiertes Schedulingverfahren in Cloud-Fog-Anwendungen. Weitere Überschneidungen existieren beispielsweise in (Castro-Jul, Redondo u. a., 2019) und (Castro-Jul, Conan u. a., 2017). Die Autoren betrachten diskrete, eventbasierte Lokalisierungsfragen mit Sensornetzwerken und entwerfen eine Fog-Computing Architektur dafür.

3.4.5. Agentenbasierte Szenenanalyse

Die Synergie zwischen agentenorientierter Softwareentwicklung und der Szenenanalyse wurde bereits aufgezeigt. Das Forschungsfeld ist klein im Vergleich zur Gesamtheit des Forschungsbereichs der Bildverarbeitung und Mustererkennung mit jährlichen Publikationen im deutlich vierstelligen Bereich allein auf den größten Konferenzen CVPR (»Computer Vision and Pattern Recognition«), ICCV (»International Conference on Computer Vision«) und ICPR (»International Conference on Pattern Recognition«). Die adressierten Fragestellungen umfassen meist die Modellierung einzelner Agenten für die Erfüllung verschiedener Erkennungsaufgaben. Die Simulation und größere Multiagentensysteme sind seltener Teil der Betrachtungen.

Ein häufiges Anwendungsfeld sind Straßenverkehrsanwendungen. Exemplarisch adressieren (Suo u. a., 2021) die Modellierung von Agenten für Multiagentensysteme für die Straßenverkehrssimulation. (Kim u. a., 2021) behandeln ebenfalls Straßenverkehrsfragen, insbesondere zukünftiges Verhalten einzelner Agenten in Bezug auf die Wahl der Fahrspur. Es existieren jedoch auch allgemeinere Arbeiten, wie beispielsweise (X. Lin, Li und Yu, 2021), in der die Autoren das Problem der Objektfindung in einer Umgebung auf der Basis eines Agenten beschreiben. (Štuikys u. a., 2016) beschreiben eine Agenten- und Roboterbasierte Lernumgebung.

Hilfreich im Kontext der Arbeit sind die Arbeiten insofern, als darauf Bezug genommen werden kann im Kontext der Erzeugung von Agenten aus Szenen und Bildfolgen.

3.5. Übersicht der betrachteten Forschungsarbeiten

Abschließend folgt in diesem Abschnitt eine Einordnung der wichtigsten Arbeiten in die Thematiken dieser Arbeit. Zur Illustration und Übersicht dient Tabelle 3.1. Dabei finden sich in den Spalten die verschiedenen Thematiken, welche bereits in Abschnitt 3.1 vorgestellt wurden. Die Zeilen im oberen Teil der Tabelle beziehen sich jeweils auf eine konkrete vorgestellte Lösung bzw. Architektur, die im unteren Teil auf angrenzende Forschungsbereiche. Während der untere Teil zur Einordnung und Information dienen soll, werden im Folgenden aus dem oberen Teil der Tabelle potentielle Kandidaten ausgewählt, deren Eignung zur Beantwortung der Forschungsfragen detaillierter evaluiert werden soll.

Die Einträge zur aktuellsten (Weiter)entwicklung beziehen sich jeweils auf die neuste Entwicklung im Kontext der Arbeit. Dies kann durch wissenschaftliche Veröffentlichungen erfolgt sein, jedoch auch durch Implementationen, Wartungspatches, offiziellen Ankündigungen oder Vergleichbarem. Die Metrik ist intendiert, um die Lebendigkeit des jeweiligen Projekts einzuschätzen. Alle referenzierten Forschungsbereiche sind aktive Forschungsfelder und erhalten daher die Jahreszahl 2022, welche dem Verfassungsdatum dieser Arbeit entspricht. Die Klammern suggerieren, dass die Metrik im Falle der Forschungsfelder im Gegensatz zu den Einzelarbeiten bei dieser speziellen Auswahl weniger Aussagekraft besitzt.

Die Tabelle ist dabei innerhalb der beiden Abschnitte nach ihren Vorkommen im Text dieser Arbeit sortiert. Diese Sortierung wiederum erfolgte auf der Basis verschiedener Überlegungen, wie Vorarbeitsstatus für andere Arbeiten (insb. JADE), dem ersten Eindruck der Eignung für die Beantwortung der Forschungsfragen und ferner der Aktualität der Arbeit. Die Einträge der Tabelle weisen dabei ein »-« auf, falls diese Thematik nicht adressiert wird, »∼«, falls die Arbeit bzw. das Forschungsgebiet das Thema zwar anspricht, sich jedoch (meist) nicht umfassend damit auseinandersetzt und »✓«, falls das Thema (üblicherweise) aufgegriffen und bearbeitet wird.

Auswahl der weiter zu untersuchenden Arbeiten

Bei der Betrachtung von Tabelle 3.1 ist direkt ersichtlich, dass keine der vorgestellten Lösungen alle Thematiken umfassend adressiert. Dies muss nicht zwangsläufig problematisch sein, da es potentiell denkbar ist, dass eine Arbeit einen Aspekt nebenher erfüllt, ohne ihn explizit zu adressieren. Die höchsten Übereinstimmungen weist Mulan auf, weshalb es in jedem Fall für weitere Untersuchungen betrachtet werden sollte.

Die wichtigsten Komponenten der Forschungsfragen umfassen die Interaktion mit Plattformen seitens der Agenten, daher sollte die Auswahl an weiteren Arbeiten

Forschungsarbeit	*Aktuellste (Weiter)entwicklung*	*Agententechnik*	*Interaktion von Agenten*	*Verteiltes System*	*(Aktive) Plattformverwaltung*	*Echte Nebenläufigkeit*	*Lokal vs. verteilt*	*Modellbasiert*
JADE	2017	✓	✓	✓	-	-	-	-
Elastic JADE	2012	✓	✓	✓	∼	-	-	-
(Hsieh)	2018	✓	∼	✓	-	∼	-	✓
cloneMAP	2021	✓	✓	✓	✓	-	-	-
DSEJAMON	2021	✓	✓	✓	-	-	✓	-
(Pawlaszczyk)	2009	✓	✓	✓	-	-	-	-
MMAS2L	2018	✓	✓	✓	∼	-	∼	-
LightJason	2018	✓	✓	-	-	-	-	-
Jadex	2021	✓	✓	✓	∼	-	-	-
Orestes	2021	-	-	✓	✓	-	∼	-
MARS	2021	✓	✓	✓	∼	-	-	-
Mulan	2022	✓	✓	✓	∼	✓	∼	✓
Vert. diskrete Eventsim.	(2022)	∼	∼	✓	-	∼	-	-
GALS Systeme	(2022)	-	-	✓	-	∼	✓	-
Infrastrukturmanagement	(2022)	-	-	✓	✓	∼	-	-
Cloud-Fog-Computing	(2022)	∼	∼	✓	-	-	✓	∼
Agentenb. Szenenanalyse	(2022)	✓	∼	∼	-	-	-	∼

Tabelle 3.1.: Übersicht der betrachteten Lösungen bezogen auf die thematische Eingrenzung. (✓: betrachtet, ∼: angeschnitten, - : nicht thematisiert) (Stand: Januar 2022)

mit Fokus auf diesen Aspekt erfolgen. Direkt erfüllt wird dieser Aspekt nur durch cloneMAP und Orestes. Da Orestes jedoch allgemein gehalten ist und ein Agentenkontext erst nachgeliefert werden müsste, soll nur cloneMAP in die nähere Betrachtung aufgenommen werden.

Elastic JADE, Jadex und MARS sprechend den Aspekt der Plattformverwaltung zwar grob an, gehen jedoch nicht darüber hinaus. Lediglich MMAS2L liefert zusätzlich noch Aspekte der Adressierung von verteilten und lokalen Anteilen der Algorithmen und soll daher ebenfalls in die Betrachtung aufgenommen werden.

Die verbleibenden Ansätze adressieren noch weniger Thematiken, sodass zusammenfassend für weitere Untersuchungen MMAS2L, cloneMAP und Mulan verbleiben.

3.6. Zusammenfassung - Stand der Forschung

Dieses Kapitel behandelte eine Übersicht der für die Beantwortung der Forschungsfragen potenziell infrage kommenden Arbeiten sowie angrenzende Forschungsbereiche. Konkret wurden die Arbeiten bzw. Projekte JADE, Elastic JADE, Pawlaszczyk, Hsieh, DSEJAMON, LightJason, MMAS2L, cloneMAP, Mulan, Jadex, Orestes und MARS in Bezug auf die Adressierung der Thematiken dieser Arbeit evaluiert. Letzten Endes ergab sich, dass die Ansätze MMAS2L, cloneMAP und Mulan weiter betrachtet werden sollten. Angrenzende Forschungsfelder umfassen die verteilte diskrete Eventsimulation, sog. GALS Systeme, Infrastrukturmanagement, Cloud-Fog-Computing und die agentenbasierte Szenenanalyse. Als weiterer Aspekt erfolgte eine Kurzvorstellung der Autoren an der Universität Hamburg, deren Arbeiten im Laufe dieser Arbeit Erwähnung fanden und finden werden. Konkret wurden dabei Hauschildt, Valk, Kummer, Moldt, Köhler, Reese, Wester-Ebbinghaus, Cabac, Bendoukha, Wagner, Simon und Dreschler-Fischer vorgestellt.

4. Anforderungsanalyse

Dieses Kapitel widmet sich der Anforderungsanalyse an eine skalierende Simulation von verteilt agierenden Agenten. Dabei wird ausgehend von den vorangegangenen Kapiteln zu Grundlagen und dem Stand der Forschung systematisch aufgezeigt, welche konkreten Punkte durch die bereits in der Literatur vorhandenen Ansätze und Lösungen nicht zufriedenstellend abgedeckt werden können. Ziel des Kapitels ist es, sowohl die zu erledigenden Aufgaben zu spezifizieren als auch eine Grundlage für die spätere Evaluation der Ergebnisse der Arbeit zu schaffen.

Alle in diesem Kapitel formulierten Anforderungen spielen sich innerhalb der Eingrenzung des Themas ab, wie es in Abschnitt 3.1 beschrieben wurde. Aus diesem Grund werden alle dort genannten Thematiken als gegeben vorausgesetzt und nicht beispielsweise eine explizite Anforderung bezüglich der echten Nebenläufigkeit des Systems oder dem Einsatz von agentenorientierter Softwareentwicklung formuliert.

Zunächst zeigt eine Vision des Gesamtsystems den Rahmen der Überlegungen in dieser Arbeit auf. Dazu wird der Umfang grob umrissen sowie die notwendigen zu betrachtenden Bestandteile herausgearbeitet. Auf der Basis dieser Bestandteile werden sodann Anforderungen an solch eine Architektur formuliert. Im folgenden Schritt erfolgt eine Auswertung bestehender Konzepte und Lösungen im Kontext der Anforderungen.

Abschließend führt der letzte Teil des Kapitels einen Plan auf, um die zuvor identifizierten Lücken zu schließen. Das Ziel dabei ist die Ebnung des Weges zur Schaffung einer Architektur, wie sie in den Forschungsfragen in Abschnitt 1.1 antizipiert wurde. Dabei werden nur die ersten beiden Forschungsfragen adressiert werden, da die dritte das Anwendungsgebiet der Szenenanalyse erfordert und daher im Aufbau der Arbeit weniger abstrakt behandelt werden muss. Die dritte Frage wird gesondert im Kontext der bis dahin vorliegenden Ergebnisse in Kapitel 13 behandelt werden.

4.1. Vision

Im Zentrum der Betrachtung stehen die Forschungsfragen, welche in Abschnitt 1.1 formuliert wurden. Die Konstruktion einer Architektur, die sowohl die Simulati-

on von Agenten auf Plattformen zulässt als auch eine Skalierung der Plattformen selbst unterstützt, ist kein einfaches Unterfangen. Aus diesem Grund soll zunächst die Vision des finalen Systems präsentiert werden. Dabei handelt es sich um eine in einer idealen Welt so existierende Architektur. Darauf aufbauend werden die bisher vorhandenen architekturellen Lösungen diskutiert und eine Lösung erarbeitet.

Im idealisierten System existieren Agenten auf logischen und/oder physikalischen Plattformen bzw. in Multiagentensystemen. Dabei existieren beliebig viele verschiedene Plattformen, mit denen die Agenten interagieren können. Interaktion bezieht sich dabei neben der Kommunikation und Kooperation mit anderen Agenten beispielsweise auch auf Anfragen bezüglich des Zustands der Plattform, wie Ressourcenauslastung, Einschätzung der Funktionstüchtigkeit der Plattform, spezifische Funktionalitäten der Plattform und weitere.

Plattformen können dabei im laufenden Betrieb hinzugefügt und wieder abgeschaltet werden. Die Plattform ist damit eine Art flüchtiger Ort, von der bei Bedarf neue Exemplare geschaffen werden können. Dabei sollten für die Bearbeitung einer aktuellen Aufgabe stets genug Plattformen zur Verfügung stehen. Agenten sollten in der Lage sein, die Erzeugung und Zerstörung neuer Plattformen anzufragen.

Jede Plattform kann eine Reihe individueller Funktionalitäten bereitstellen, auf die Agenten bei Bedarf zugreifen können. Dabei kann es sich um allgemeine Funktionalitäten handeln, die beispielsweise besonders gut auf der jeweiligen Plattform ausgeführt werden können, denkbar sind aber auch sehr spezifische Funktionalitäten, wie die Ansteuerung spezieller lokaler Gegebenheiten, mit denen die Plattform ausgestattet ist. Plattformen sind dabei effektiv heterogene Umgebungen, welche sich durch die Zusammenstellung ihrer Funktionalitäten auszeichnen. Ist eine spezielle Funktionalität in sich nachlieferbar, sollte auch dies möglich sein. Da dies etwas abstrakt klingt, kann als realweltliche Interpretation beispielsweise eine Tafel oder ein Whiteboard in einem Raum genannt werden. Die Interaktion der Kommunikationspartner ist auch ohne Tafel im Raum möglich, eine Tafel kann jedoch nachgerüstet werden und ist dann Teil des Raums (Teil der Plattform). Der Raum stellt dann die Funktionalität, reversibel Notizen an der Wand niederschreiben zu können, bereit.

Zuletzt besteht die Möglichkeit für Agenten, die Plattform zu wechseln. Dabei müssen insbesondere die Protokolle und Wissensbasis des Agenten auf die neue Plattform übertragen werden. Bei Bedarf kann ein Log der Aktionen eines Agenten angefordert werden. Unabhängig von der lokalen Plattform kann ein Agent auch eine Nachricht an einen beliebigen anderen Agenten auf einer beliebigen anderen Plattform versenden. Dabei erfolgt die Auflösung der konkreten Koordinaten des Kommunikationspartners innerhalb der Plattformlandschaft automatisiert. Bei intensiver Kommunikation bzw. Kooperation ist ein Plattformwechsel

und lokaler Austausch jedoch der plattformübergreifenden Kommunikation vorzuziehen. Die plattformübergreifende Kommunikation ist jedoch im Allgemeinen als notwendig zu erachten, um überhaupt Kooperationen mit Agenten anderer Plattformen einleiten zu können.

Aus Sicht des Nutzers des Agentensystems besteht somit die Möglichkeit, heterogene Systemlandschaften zusammenzuschließen, anstatt auf gleichförmige Rechnersysteme angewiesen sein zu müssen. Dies wird im Wesentlichen durch die dynamische Bereitstellung von Funktionalitäten gewährleistet. Darüber hinaus können Nutzer auch die Rolle eines Agenten selbst einnehmen und mit einzelnen Plattformen interagieren.

So können gezielt manuelle Eingriffe erfolgen, um das Gesamtsystem im Bedarfsfall anpassen zu können. Darüber hinaus können aggregiert Statusinformationen eingesehen werden. Durch die Skalierbarkeit der Plattformmenge wird die Berechnung weitaus größerer und komplexerer Simulationen möglich, als dies mit nur einer Plattform oder einer statischen Menge an Plattformen möglich wäre.

4.2. Definition der Anforderungen

Auf der Basis der Vision der Möglichkeiten und des Umfangs des Systems können nun entsprechende Anforderungen an die Architektur und damit auch an diese Arbeit abgeleitet werden. Diese Anforderungen dienen der Erleichterung der Bewertung bestehender Lösungen im Kontext der Forschungsfragen sowie der späteren Evaluation der Arbeit im selben Kontext. Die Anforderungen sind dabei in verschiedene Kategorien unterteilt und mit entsprechenden Kürzeln versehen. Die Kürzel dienen der einfacheren Handhabung und Referenzierung beispielsweise in Tabellen.

Zunächst werden Interaktionsanforderungen betrachtet, welche die Interaktionsmöglichkeiten aus Sicht von Agenten im System adressieren. Interaktionsanforderungen erhalten als Kürzel »**I**«. Neben den Interaktionsanforderungen spielen strukturelle Anforderungen an die Architektur eine zentrale Rolle. Diese sind im darauf folgenden Abschnitt dargelegt und tragen das Kürzel »**S**«. Die Modifikation von Strukturen und Einheiten der Architektur zur Laufzeit stellt eine der zentralen Forderungen an die Architektur dar. Die konkrete Ausgestaltung dieser dynamischen Modifikationen ist im Abschnitt zu dynamischen Anforderungen mit dem Kürzel »**D**« festgehalten. Letztendlich erfolgte die einleitende Motivation zur Betrachtung der Forschungsfragen mit der Anwendungsdomäne der Szenenanalyse. Die Architektur sollte grundlegend dafür einsetzbar sein. Da es sich hierbei um einen separaten Aspekt handelt, erhält diese Anforderung ihre eigene Kategorie: Anforderungen bezüglich der Anwendungsdomäne mit dem Kürzel »**A**«.

4.2.1. Interaktionsanforderungen

Interaktionsanforderungen betreffen die speziellen, den Agenten des Systems möglichen, Aktionen, um Einfluss aufeinander und ihre Umgebung zu nehmen. Hierbei sind drei Anforderungen aus der Vision zu extrahieren.

Anforderung I1

Agenten können direkten Einfluss auf die Erzeugung und Vernichtung von Plattformen nehmen.

Ein entsprechendes System sollte in der Lage sein, den Agenten Möglichkeiten einzuräumen, sich selbst neue Plattformen, auf denen sie miteinander interagieren können, zu schaffen. Agenten benötigen dafür entweder ein eigenes Verständnis für die Gesamtmenge an Plattformen oder aber es muss eine (ggf. logische) Instanz über der Ebene der Plattformen existieren, an welche derartige Anforderungen geschickt werden können. Da es sich bei dieser Anforderung um eines der zentralen Elemente der Forschungsfragen handelt, kommt diesem Punkt eine hohe Wichtigkeit zu. Diese Anforderung wird im Verlauf der Arbeit auch als »skalierende Agentensimulation« referenziert werden.

Anforderung I2

Agenten können mit Plattformen in Interaktion treten.

Neben der Interaktion der Agenten untereinander sollte es für Agenten auch möglich sein, mit Plattformen in Kontakt zu treten und auf diese Einfluss zu nehmen. Analog dazu, wie realweltliche Entitäten ihre Kommunikationsräume im Rahmen der Möglichkeiten formen können, sollten die in der Simulation eingesetzten Plattformen ebenfalls eine solche Interaktion mit Agenten ermöglichen. Auch hierbei handelt es sich um ein wichtiges Kriterium, da die Interaktion mit Plattformen ebenfalls in den zentralen durch die Forschungsfragen adressierten Fragestellungen liegt.

Anforderung I3

Agenten können über Plattformgrenzen hinweg kooperieren und kommunizieren.

Basierend auf der angestrebten Möglichkeit, dass Agenten Einfluss auf die Erzeugung von Plattformen besitzen, erscheint es logisch, dass Agenten auch mit

Agenten auf anderen Plattformen zumindest grundlegend in Kontakt treten können, um eine Interaktion zu initiieren. Wird diese Anforderung nicht umgesetzt, würde dies im angestrebten Rahmen zwangsläufig bewirken, dass Agenten nur entweder mit lokalen anderen Agenten kooperieren können, welche zufällig auf derselben Plattform zugegen sind oder aber, dass Kommunikationspartner vorgegeben werden. Beide Aspekte sind nachteilig für das Leitbild der Arbeit. Die erste Variante würde keine gezielte Kooperation möglich machen und die zweite Variante Agenten in ihrer Autonomie einschränken. Diese Anforderung ist daher ebenfalls als eine der wichtigen Anforderung zu bewerten.

4.2.2. Strukturelle Anforderungen

Strukturelle Anforderungen betreffen den Aufbau bezüglich Agenten und Plattform der Architektur. Sie beinhalten weiterhin Annahmen bezüglich der Struktur der abgebildeten Zusammenhänge. Zwei Kriterien sollen hierbei benannt und besonders beachtet werden.

Anforderung S1

Die hierarchische Struktur zwischen Agenten und Plattformen wird abgebildet.

Während Agenten und Plattformen sowohl autonome Einheiten sind und darüber hinaus Agenten und Plattformen in Interaktion treten können sollen, betten sich nach dem Verständnis dieser Arbeit entlang der Einheitentheorie Agenten in Plattformen ein. Diese inhärente Hierarchie zwischen Agentenplattformen sollte durch eine entsprechende Lösung abgebildet werden. Während diese Anforderung für das Verständnis der Arbeit wichtig ist, wäre es jedoch denkbar, diese Anforderung unterzuordnen, sofern eine Lösung alle anderen Punkte hervorragend erfüllt. Vielmehr wird durch die Argumentationsstruktur dieser Arbeit nahegelegt, dass eine entsprechende Lösung eine solche Hierarchie beinhalten sollte. Weist eine Lösung wie eben beschrieben diesen Punkt nicht auf, so sollte die Begründung seiner Abwesenheit stichhaltig dargelegt sein.

Anforderung S2

Plattformen können heterogen sein.

Eine möglichst realitätsnahe Abbildung interagierender Entitäten führt zwangsläufig dazu, eine heterogene Plattformlandschaft anzunehmen. Während andersherum bei der klassischen Konstruktion von Systemen im Idealfall eine homogene

Umgebung angestrebt wird, ist dies bei der beschriebenen Abbildung im Allgemeinen nicht haltbar oder realistisch. Eine Lösung sollte daher eine gewisse Form von Heterogenität unter den Plattformen unterstützen und Möglichkeiten aufzeigen, wie diese Plattformen in einer entsprechenden Simulation umgesetzt werden können. Auch hierbei handelt es sich um einen wichtigen Punkt, da seine Abwesenheit eine zu starke Vereinfachung der Problemstellung darstellt. Gerade der Umgang mit heterogenen Systemen bringt stets eine erhöhte Komplexität mit sich.

4.2.3. Dynamische Anforderungen

Dynamische Anforderungen betreffen alle Anpassungen und Änderungen zur Laufzeit einer Realisierung der Architektur. Um dynamische Änderungen zulassen zu können, muss die Architektur diese Umstände in besonderem Maße vorsehen. Bei dieser Kategorie sind drei wesentliche Kriterien zu nennen.

Anforderung D1

Die Funktionalitäten von Plattformen können dynamisch angepasst werden.

Wie bereits in der Vision motiviert, sind Funktionalitäten einer Plattform nicht zwangsläufig unveränderlich. Während diese auch ein Ausdruck der Heterogenität der Plattformlandschaft sind, ist es möglich, manche der Funktionalitäten dynamisch anzupassen. Aus diesem Grund sollten durch eine Lösung bereitgestellte Plattformen eine Möglichkeit zur Erweiterung und Änderung der Funktionalitäten zur Laufzeit anbieten. Die Anforderung ist insofern als wichtig zu erachten, als die angenommene Heterogenität der Umgebung zumindest eine Konfigurierbarkeit von Plattformen erfordert. Sofern neue Plattformen von Agenten dynamisch erschaffen werden können, wie es Anforderung **I1** fordert, können nicht für jede denkbare Möglichkeit spezifische Plattform vorgehalten werden. Somit ist die dynamische Modifizierbarkeit von Plattformen eine notwendige Eigenschaft, die jede geeignete Lösung aufweisen muss.

Anforderung D2

Agenten können Plattformen wechseln.

Um neu erzeugte Plattformen nutzen zu können, ist es erforderlich, Agenten einen Plattformumzug zu ermöglichen. Nur auf diese Weise können neu erschaffene Plattformen effizient genutzt werden. Eine denkbare andere Umsetzung wäre die

automatisierte Umverteilung von Agenten auf geeignete Plattformen, um so neue Plattformen entsprechend zu bevölkern. Diese Umsetzung würde aber dem Grundgedanken der Autonomie von Agenten widersprechen und sollte daher nicht von einer Lösung angestrebt werden. Die Anforderung ist insofern wichtig, als das Fehlen ihrer Umsetzung die weitgehende Einschränkung des Nutzens der Architektur mit sich ziehen würde.

Anforderung D3

Der Nachrichtenversand erfolgt für die Agenten auf technischer Ebene transparent.

Agenten können sich gegenseitig bekannt sein. Dies lässt sich durch Aliase bzw. allgemeiner Identifikatoren ausdrücken. In einem dynamischen, verteilten System, bei dem Plattformen erzeugt, manipuliert und vernichtet werden können, kann die Adressierung anderer Agenten weitaus problematischer sein als im statischen lokalen Fall. Dennoch sollte die technische Auflösung dieser Adressierung nicht Teil der Verantwortung des jeweiligen Agenten sein. Vielmehr sollte die Architektur einen entsprechenden Verzeichnisdienst bzw. eine andere Lösung für eine für Agenten transparent erfolgende Lokalisation und Referenzierung anderer Agenten anbieten. Diese Anforderung ist nicht zwingend umzusetzen, erleichtert die Modellierung entsprechender Systeme jedoch ungemein.

4.2.4. Anforderungen durch die Anwendungsdomäne

Bestimmte Anwendungsdomänen erfordern bestimmte Umstände, Beachtungen seitens der Architektur und mitunter Spezialbehandlungen. Während die Architektur gezielt unabhängig von der Anwendungsdomäne und möglichst allgemein entwickelt werden soll, ist es insbesondere in der Evaluation der Ergebnisse der Arbeit hilfreich, hier ein Kriterium zu formulieren.

Anforderung A1

Die Architektur sollte mindestens für einen Einsatz in der Anwendungsdomäne der Szenenanalyse geeignet sein.

Während die entsprechende Lösung selbstverständlich für allgemeine Interaktion und somit verschiedene Anwendungsdomänen einsetzbar sein soll, wurde doch in der Einleitung motiviert, dass die Szenenanalyse ein hervorragendes Hilfsmittel dazu darstellt, eine Abbildung von realer Welt zu Modell bzw. Hypothese zu

ermöglichen. Die spezielle Anwendungsdomäne der Szenenanalyse nimmt daher eine gesonderte Stellung im Rahmen dieser Arbeit ein. Eine Lösung sollte entsprechende Kompatibilität zumindest nachvollziehbar motivieren. Diese Anforderung ist im Kontext der gesamten Arbeit von Wichtigkeit, stellt aber wie dargelegt nur einen möglichen Aspekt einer zu konstruieren Lösung dar.

4.3. Evaluation bestehender Lösungen

Dieser Abschnitt adressiert die Evaluation der vielversprechendsten bestehenden Lösungen aus Kapitel 3. Das Ziel dabei ist die Evaluation, ob bereits bestehende konzeptuelle (und ggf. implementierte) Architekturen die im vergangenen Abschnitt aufgeführten Anforderungen unter Umständen bereits abbilden können. Sollte sich dies nicht bewahrheiten, dient dieser Abschnitt sogleich der Begründung der Konstruktion einer neuen, den Anforderungen entsprechenden Architektur. Ferner wird die Architektur mit den besten Voraussetzungen die zentrale Vorarbeit bilden, auf deren Basis weitere Überlegungen angestrengt werden sollen. Durch die zu starken Abweichungen von der thematischen Verortung der Arbeit sollen die Ansätze JADE, Elastic JADE, Pawlaszczyk, DSEJAMON, LightJason, Jadex, Orestes und MARS an dieser Stelle nicht weiter betrachtet werden. Ebenfalls wird der Ansatz durch Hsieh nicht weiter betrachtet, da, wie bereits in Abschnitt 3.2.3 beschrieben, Anknüpfungspunkte im Rahmen der Arbeit fehlen.

4.3.1. MMAS2L

Die Architektur MMAS2L (Murakami u. a., 2018) umfasst die Beschreibung eines Agentensystems auf der Basis von zwei Ebenen. Der Ansatz ist nicht modellgetrieben. Während Plattformen beschrieben werden, ist eine automatische Skalierung sowie die Interaktion von Agenten mit der Plattform nicht vorgesehen. Dahingegen wird Kooperation zwischen Agenten auf unterschiedlichen Plattformen vorgesehen, die Beschreibung von Hierarchie beschränkt sich jedoch auf die rudimentären Ebenen von Agenten und Plattformen. Eine mögliche Heterogenität und Anpassbarkeit der Plattform(agenten) wird ebenso nicht thematisiert wie die Möglichkeit von Agenten Plattformen zu wechseln, weswegen die Annahme getroffen wird, dass diese Aspekte jeweils nicht unterstützt werden. Agenten IDs sorgen für einen transparenten Versand von Nachrichten. Eine Anwendungsdomäne wird im Kontext der Sprachverarbeitung illustriert.

4.3.2. cloneMAP

Der Ansatz cloneMAP (Happ, Dähling und Monti, 2020; Dähling, Razik und Monti, 2021) umfasst die cloud-native Implementation eines klassischen Multiagentensystems. Einzelne Strukturen werden dabei in gleichartigen Containern bereitgestellt und die gesamte Deploymentlandschaft entspricht einem klassischen Microservice-Deployment mit einem Verzeichnisdienst, verschiedenen zustandslosen Berechnungseinheiten und einer replizierten und geshardeten zentralisierten Datenbank sowie einem Containerorchestrator. Das Ziel der Architektur ist die komplette Abstraktion des Plattformbegriffs aus Sicht der Agenten. Folglich sind Agenten nicht in der Lage, direkt Einfluss auf die Plattformmenge zu nehmen oder mit Plattformen zu interagieren.

Im Fall von cloneMAP bedarf eine Kommunikation über Plattformgrenzen hinweg einer genauen Definition des Plattformbegriffs. Aus Sicht der Persistenz der Agentendaten handelt es sich dabei (auf logischer Ebene) nicht einmal um ein verteiltes System, da alle Daten in einem zentralisierten Lager vorgehalten werden. Mit einer technischen Sicht und Blick auf die Berechnungseinheiten kann argumentiert werden, dass Agenten, die auf verschiedenen Plattformen Berechnungen durchführen, kooperieren und kommunizieren können. Analog gilt dies für die Möglichkeit des Plattformwechsels von Agenten. In beiden Sichtweisen kann jedoch auch argumentiert werden, dass Agenten auf alle verfügbaren Plattformen umziehen können, sodass die Anforderung als erfüllt betrachtet werden kann.

Durch die Abstraktion der Plattformen wird kein Wert auf die hierarchische Struktur zwischen Agenten und Plattformen gelegt. Ebenso scheidet eine Heterogenität sowie die dynamische Anpassbarkeit von Plattform dadurch aus. Der Nachrichtenversand erfolgt durch die Lösung innerhalb des Microservice Deployments für Agenten transparent.

Abschließend sollte bei cloneMAP jedoch noch beachtet werden, dass es sich dabei um eines der wenigen betrachteten Agentensysteme handelt, welche eine Plattformskalierung überhaupt adressieren, wie in Abschnitt 3.2.4 diskutiert wurde.

4.3.3. Mulan und Capa

Mulan erfüllt bereits eine große Menge der Anforderungen. Die hierarchischen Zusammenhänge zwischen Agent und Plattform werden hervorragend abgebildet und auch der Umzug von Agenten wird beispielsweise in Capa adressiert. Der Versand von Nachrichten erfolgt für Agenten auf einer abstrakten Ebene und wird in der Implementation vom System übernommen. Das Verständnis von Plattformen als Agenten erlaubt zumindest die grundlegende Kommunikation zwischen Agen-

Architektur	I1	I2	I3	S1	S2	D1	D2	D3	A1
MMAS2L	-	-	✓	∼	-	-	-	✓	-
cloneMAP	-	-	∼	∼	-	-	✓	✓	-
Mulan	-	∼	✓	✓	∼	-	✓	✓	-

Tabelle 4.1.: Übersicht der betrachteten Lösungen und ihre Erfüllung der Anforderungen.

ten und Plattformen. Die Grenze mehrerer Plattformen ist in Mulan modelliert, jedoch als relativ abstrakt belassen, die Kommunikation über Plattformgrenzen ist jedoch konzeptuell möglich. Die Heterogenität von Plattformen ist abstrakt gehalten. Mindestens in der Capa Plattform ist durch den Einsatz verschiedener Plugins für den Simulator Renew ab Version 2.0 eine Abbildung von Heterogenität möglich. Nicht ausreichend adressiert wird durch Mulan jedoch die Möglichkeit von Agenten Einfluss auf die Erzeugung und Vernichtung von Plattformen zu nehmen, die dynamische Anpassung von Plattformen und Abbildungen im Kontext der Szenenanalyse. Der letzte Punkt wurde nicht in der Literatur diskutiert, sollte aber generell möglich sein. Die Erzeugung und Vernichtung von Plattformen ist konzeptionell durch den Ansatz Plattformen wie Agenten zu behandeln adressiert, die technische Umsetzung ist jedoch bisher weder im Kontext von Mulan noch Capa umfassend diskutiert worden.

4.3.4. Übersicht

Abschließend werden die jeweiligen Ergebnisse der Bewertung der bestehenden Lösungen in einer übersichtlichen Tabelle zusammengefasst. Dabei referenzieren die jeweiligen Spalten die Anforderungen, wie sie in Abschnitt 4.2 dieses Kapitels dargelegt wurden und die Zeilen die jeweiligen Lösungen.

Analog zu Tabelle 3.1 verwendet diese Tabelle die Symbole »-«, falls die betreffende Anforderung nicht erfüllt wird, »∼«, falls die Anforderung nur zum Teil erfüllt wird und »✓«, falls die Anforderung erfüllt wird.

Die jeweiligen Ansätze können in keinem der Fälle alle hergeleiteten Anforderungen erfüllen. Aus diesem Grund wird es nötig, im Rahmen dieser Arbeit selbstständig die Lücken zu schließen, um eine Antwort auf die Forschungsfragen formulieren zu können.

Allen Ansätzen gemein ist die Notwendigkeit für eine Implementation des Einflusses von Agenten auf Plattformerzeugung und -vernichtung, der dynamischen Anpassung von Plattformfunktionalitäten sowie einer Evaluation des Einsatzes in Szenenanalyseaufgaben. Lediglich die Mulan-Architektur stellt innerhalb der Auswahl bereits umfassende Lösungen für die Abbildung hierarchischer Struktu-

ren bereit und darüber hinaus rudimentäre Funktionen für die Interaktion zwischen Agenten und Plattformen sowie der Heterogenität von Plattformen. Mit einem erneuten Blick auf Tabelle 3.1 kann auch festgehalten werden, dass die MULAN-Architektur die größten Überschneidungen mit den Thematiken der Arbeit aufweist.

Aus diesem Grund kann an dieser Stelle eine wegweisende Entscheidung für den weiteren Verlauf dieser Arbeit gefällt werden:

Es muss eine neue Architektur für die Beantwortung der Forschungsfragen und Erfüllung der Anforderungen konzipiert werden. Dabei soll die MULAN-Architektur als konzeptuelle Basis dienen.

4.4. Schließen der Lücken und Konstruktionsansatz

Nach der Evaluation bestehender Lösungen wird ersichtlich, dass keine bestehende Lösung die Anforderungen umfassend erfüllen kann. Die größte Erfüllung der geforderten Aspekte der Architektur ist durch die MULAN-Architektur gegeben. Sie ist daher ein natürlicher Kandidat als Basis für die nun folgende Entwicklung einer entsprechenden Architektur.

In erster Linie sollte MULAN hierbei als Referenzarchitektur betrachtet werden. In MULAN existieren mehrere Plattformen, welche durch Kommunikationskanäle verbunden sind und Agenten können auch über Plattformgrenzen hinweg miteinander kommunizieren.

Dabei ist jedoch die Funktionsfähigkeit der in MULAN modellierten Plattformen sehr rudimentär. Agenten können erzeugt werden, sie können entfernt werden und interne und externe Kommunikation wird ermöglicht. Als oberste Ebene ist lediglich eine grob beschriebene Infrastruktur zu finden. Die Ausdefinition dieser Infrastruktur oder etwaiger darin existierender Verwaltungsentitäten adressiert MULAN nicht, sondern legt den Fokus im Wesentlichen auf die Ebene der Agenten und ihre Protokolle.

Folglich wäre es wünschenswert, eine weitere Ebene oberhalb der Plattformen beschreiben zu können. Diese Ebene könnte analog zum Verhalten der Plattform gegenüber den Agenten genutzt werden, um Plattformen zu erzeugen und zu zerstören.

Den Ansatz der Erweiterung der MULAN-Architektur durch darüber angesiedelte Ebenen verfolgt die Multiorganisations-Referenzarchitektur ORGAN (WESTER-EBBINGHAUS, 2010). Die kleinste in ORGAN betrachtete Einheit ist das Multiagentensystem, welches in der Definition von MULAN der Plattform entspricht. Darüber existieren Organisationen, Branchen und die Gesellschaft.

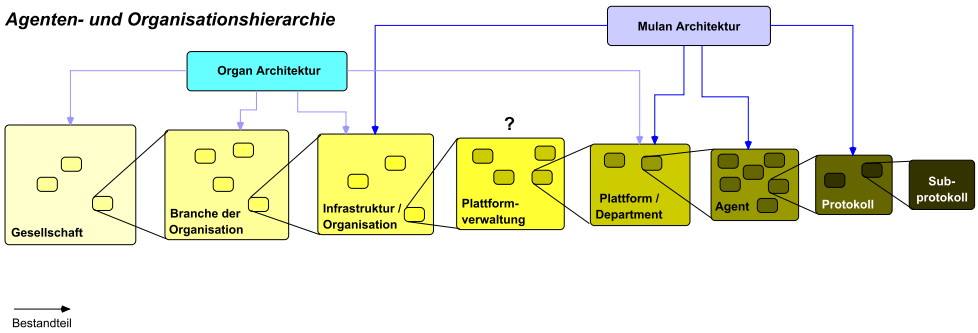

Abbildung 4.1.: Gesamthierarchie von MULAN und ORGAN

Während ORGAN die MULAN-Architektur in der Tat nach oben hin erweitert, wird die Ebene der Verwaltung von Plattformen übersprungen und indirekt durch die Organisation beschrieben. Die Organisation in ORGAN entspricht eher der abstrakten Infrastruktur in MULAN. Eine bildliche Darstellung dieser Adressierung und der Tatsache, dass beide Referenzarchitekturen eine etwaige Plattformverwaltung auslassen, findet sich in Abbildung 4.1.

Ein Schritt in die Richtung der Erschaffung komplexerer Plattformen für Agentensimulationen wurde mit der Plattform CAPA vollzogen. Das erklärte Ziel der Konstruktion dieser speziellen Plattform lag allerdings darin, eine Kompatibilität mit den Protokollen der FIPA herzustellen. Somit sollte die Interaktionsmöglichkeit zwischen Agenten auf Referenznetzbasis und beliebigen FIPA-konformen Agenten geschaffen werden. Die Plattform in CAPA adressiert somit ebenfalls einen Aspekt der Verteilung, fokussiert sich jedoch auf lokale Kompatibilitätsbetrachtungen und adressiert die übergeordnete Infrastruktur nach wie vor abstrakt.

Da ein erklärtes Ziel der Architektur das dynamische Erzeugen von Plattformen beinhaltet, muss und sollte zu diesem Zweck eine andere Art Plattform geschaffen werden. Darüber ist es naheliegend, wie beschrieben, wenn für das Erzeugen und Zerstören von Plattformen eine übergeordnete Einheit eingeführt wird. Diese Einheit soll im Weiteren, wie bereits angedeutet, als *Plattformmanagement* beschrieben werden.

4.4.1. Plattform als Agent

Eine Idee, welche bereits in (DUVIGNEAU, 2002) im Rahmen von CAPA zum Einsatz kommt und welche sich ebenfalls aus der Einheitentheorie ableiten lässt, kann auch für diesen Einsatzzweck Verwendung finden: Die Plattform selbst als Agenten zu betrachten. Die Idee umfasst im weiteren Sinne die MULAN-Hierarchie

auf beliebige Granularitäten abzubilden. Eine Verallgemeinerung dazu inklusive eines Managements wird in (Schleinzer, 2007) in Form des hierarchischen Capa beschrieben. Dabei wurden Plattformen wie Agenten behandelt und die Erzeugung einer Plattform konnte durch die Erzeugung eines Agenten modelliert werden. Die behandelte Tiefe hierbei genügt jedoch nicht für die Beantwortung der Forschungsfragen dieser Arbeit.

Auf der Grundlage aus (Schleinzer, 2007) wurde dann in (Duvigneau, 2009) das Plugin-System und die Plugin-Verwaltung für Renew konstruiert. Dennoch existieren an Plattformen und an die Verwaltung von Plattformen sehr unterschiedliche Anforderungen, welche im Folgenden im Rahmen dieses Kapitels herausgearbeitet werden. Daher kann an dieser Stelle eine weiteres Mal festgestellt werden: Die Anforderungen an Plattformen und an die Verwaltung von Plattformen können trotz Verschiebung der Granularitäten nur unzureichend mit dem Mulan-Modell beschrieben werden.

Dennoch soll zunächst als erste Heuristik die beschriebene Granularitätsverschiebung der Mulan-Architektur angewandt werden: Bei der Betrachtung der gewünschten Funktionalität Plattformen dynamisch erzeugen und zerstören zu können, liegt es also nahe, die Eigenschaften der abstrakten Mulan-(Agenten-) Plattform zu betrachten. Diese besitzt bereits genau die geforderten Eigenschaften bezogen auf Agenten anstatt auf Plattformen. Kann nun die Plattform ihrerseits als Agent interpretiert werden, kann die Konstruktion einer Mulan-Plattform als erste Referenzstruktur für die Erzeugung des Plattformmanagements dienen.

Insgesamt ergibt sich auf diese Art und Weise eine Verschiebung innerhalb der von Mulan vorgestellten Hierarchie: Aus Plattform wird Plattformmanagement, aus Agent wird Plattform, aus Protokoll wird Agent. In einer ersten Betrachtung könnte argumentiert werden, dass so die Idee eines Plattformmanagements in das Konzept der Mulan-Plattform kollabiert. Wie sich jedoch in den weiteren Ausführungen der Arbeit ergeben wird, ist der Aufbau von Plattform und Plattformmanagement nach konsequenter Herleitung der Funktionalitäten deutlich unterschiedlich, sodass die beiden Konzepte nicht austauschbar sind. Dennoch bietet die grundlegende Idee der Verschiebung der Mulan-Hierarchie einen zentralen Ansatz- und Ausgangspunkt. Die Frage nach der Notwendigkeit eines Plattformmanagements wird nach allen Betrachtungen abschließend erneut in Abschnitt 8.2 und Abschnitt 8.3 aufgegriffen.

Der Ansatz des verschobenen Mulan-Modells als zentraler *Ausgangsgedanke* wird bei allen weiteren Konzeptionen im Rahmen dieser Arbeit Anwendung finden. Darauf aufbauend werden die Aspekte der Architektur für skalierende Agentensimulationen Schritt für Schritt hergeleitet. Zusammenfassend wird somit die Erweiterung der Mulan und Organ Architektur wiederum erneut nach der Mulan-Architektur vollzogen. Bei der Idee aus der verschobenen Selbstanwendung von Mulan als Ausgangspunkt die Konstruktion eines Plattformmanagements

als fünfte Ebene der MULAN-*Architektur* herzuleiten, handelt es sich um eine der zentralen Neuerungen durch diese Arbeit.

Daher gilt es im Folgenden zu erörtern, welche Funktionalität die jeweiligen Bestandteile der erweiterten MULAN-Architektur konkret aufweisen müssen, damit sie ihre Aufgaben erfüllen können.

4.4.2. Plattformmanagement, Plattform und Agent

Um eine Abschätzung der Zusammenhänge zu ermöglichen, lohnt eine Betrachtung des verschobenen MULAN-Modells. Bei diesem wird aus der Plattform das Plattformmanagement, aus Agenten Plattformen und aus Protokollen Agenten. Dieses ist als erster Entwurf mit abgewandelten Beschriftungen in Abbildung 4.2 dargestellt. Für die Orientierung wird in den folgenden Ausführungen stets der Begriff *Infrastruktur*, *Plattformmanagement*, *Plattform* und *Agent* für die in Abbildung 4.2 dargestellt Architektur verwendet, während die Begriffe MULAN-Infrastruktur, MULAN-Plattform, MULAN-Agent und MULAN-Protokoll für die jeweiligen Einheiten der ursprünglichen MULAN-Architektur (vgl. Abbildung 3.1 im Grundlagenkapitel) verwendet werden.

Schlussendlich ist Ziel der Untersuchung eine Beschreibung einer abgewandelten MULAN-Architektur, welche in einer geordneten und gezielten Art und Weise ein Plattformmanagement einsetzt, um damit eine Skalierbarkeit der Agentensimulation zu unterstützen und zu ermöglichen. Als Ausgangspunkt folgt daher zunächst eine Analyse, was die Verschiebung der MULAN-Architektur konkret bedeuten würde. Die Betrachtung der einzelnen Elemente der Abbildung werden daher unter der Prämisse erfolgen, dass bestimmte Teile der originalen MULAN-Architektur und der verschobenen MULAN-Architektur (vgl. Abbildung 4.2) miteinander verschmolzen werden sollen.

Konkret bedeutet dies, dass beispielsweise die Plattform, welche hier mit dem Netz des MULAN-Agenten dargestellt wird, sowohl Funktionalitäten des MULAN-Agenten als auch Funktionalitäten der MULAN-Plattform in sich vereint. Analog gilt dies für MULAN-Protokoll und MULAN-Agent, sowie MULAN-Plattform und das neu zu erschaffene Plattformmanagement mit seinen zuvor definierten Anforderungen. Inhaltlich werden entlang der Einheitentheorie vor allem die Interaktionen mit den übergeordneten Einheiten, denen auf gleicher Ebene und den untergeordneten Einheiten adressiert.

Die Änderung in dem Infrastrukturanteil der MULAN-Architektur sind überschaubar. Anstatt MULAN-Plattformen werden nun Plattformmanagements durch eine abstrakte Kommunikationsstruktur voneinander getrennt. Die Konzepte sind und bleiben ähnlich und bedürfen daher keinem großen Integrationsaufwand.

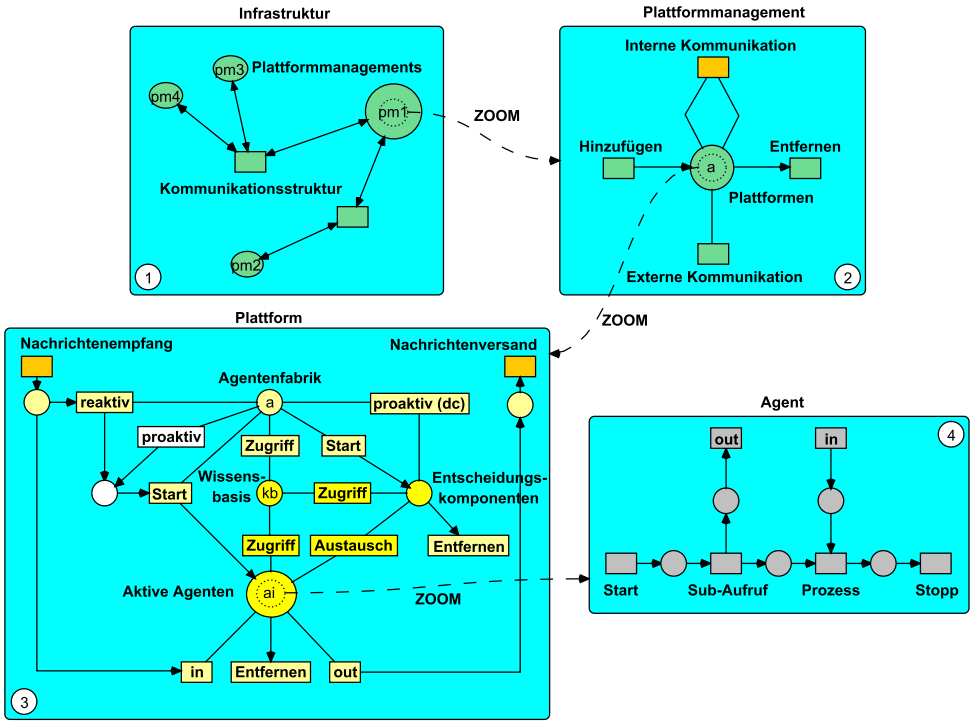

Abbildung 4.2.: Erste Idee: Einfache Verschiebung der MULAN-Architektur zur Abdeckung vom Plattformmanagement

Im Plattformmanagement existieren sowohl die Möglichkeiten des Hinzufügens als auch des Entfernens von Plattformen. Darüber hinaus besteht eine Schnittstelle zur externen Kommunikation mit Plattformen unter anderen Plattformmanagements. Je zwei Plattformen können auch in interne Kommunikation miteinander treten. Hierbei entsteht zusätzlich jetzt der Aspekt der Kommunikation zwischen Plattformen und der Kommunikation zwischen Agenten, da auch zwei Agenten, welche auf unterschiedlichen Plattformen beheimatet sind, Nachrichten austauschen können sollen. Der Aspekt der Trennung der Kommunikationsarten sollte somit im Plattformmanagement noch hinzugefügt werden. Darüber hinaus müssen Erzeugung und Destruktion von Plattformen weiter ausgeführt werden. Dazu zählt ebenfalls, wo die Entscheidung gefällt werden sollte, wann eine weitere Plattform generiert und wann eine Plattform abgebaut wird.

Wird das Modell der Plattform betrachtet, so fällt als erstes auf, dass das MULAN-Modell nach oben viele Details nicht weiter ausspezifiziert. Somit entstehen Agenten einfach und Funktionalitäten sind einfach gegeben, es fehlt eine Beschreibung, welche der Heterogenität der Plattformen Rechnung trägt. In der Verschiebung unterscheidet das Modell zwischen Agenten und aktiven Agenten. Die Plattform kann proaktiv Agenten aktivieren oder als Reaktion auf externe Einflüsse Agenten in den aktiven Zustand überführen. Dabei greift sie auf eine Wissensbasis zurück, welche bei dieser Entscheidung helfen kann. Durch die mit **in** und **out** markierten Transitionen können aktive Agenten an Nachrichtenübertragungen teilnehmen. Elemente, welche einer weiteren Untersuchung bedürfen, umfassen beispielsweise die Wissensbasis oder auch die Unterscheidung zwischen Agenten und aktiven Agenten und deren Erzeugung und Vernichtung.

Die Wissensbasis umfasst Informationen, welche die gesamte Plattform betreffen und hat damit eine spezielle Form im Gegensatz zur allgemein gehaltenen Struktur der Wissensbasis eines Agenten. Die Erzeugung und Zerstörung von Agenten ist hier im Gegensatz zur MULAN-Plattform deutlicher spezifiziert, indem Agenten als eine Art Templates vorgehalten werden können. Dennoch wäre die Limitation auf eine feste Menge an Agenten eine Einschränkung, welche von der MULAN-Plattform so nicht vorgesehen ist. Folglich sollte mindestens für die Nutzer des Systems eine Möglichkeit bestehen, Plattformen neue Agentendefinition zur Verfügung zu stellen.

Darüber hinaus sollte eine Plattform in der Lage sein, Auskünfte über ihren Zustand geben zu können. In einem herkömmlichen Agenten würde dies über den Nachrichtenausgang innerhalb eines Protokolls erfolgen. Da die Agenten innerhalb der Plattform jedoch allgemein gehalten sind, bedarf es eines Satzes spezialisierter Agenten, welche mit den Gegebenheiten der Plattform interagieren können und über diese Auskunft erteilen können.

Eine Schwierigkeit für die Modellierung von Interaktion zwischen aktiven Agenten besteht darin, dass in dem Modell der Plattform keine direkte interne Kommunikation zwischen Agenten vorgesehen ist. Diese muss also ebenfalls ergänzt werden.

Schlussendlich sollte noch der Aspekt der Heterogenität der Plattformen betrachtet werden. Wie zu Beginn der Arbeit erläutert, soll eine besondere Stärke der Architektur die Heterogenität der Plattformen umfassen. Folglich müssen Plattformen eine Menge an plattformspezifischer Funktionalität aufweisen, mit der Agenten interagieren bzw. welche diese nutzen können. Diese Menge an plattformspezifischer Funktionalität sollte implementationsseitig nachträglich bereitgestellt werden können, um dynamisch auf wechselnde Anforderungen reagieren zu können. Dennoch sollte ein genereller Update-Mechanismus im Plattformmanagement existieren, durch welchen das statische Grundmodell einer Plattform angepasst werden kann, um in späteren Simulationsläufen auf global geänderte Anforderungen reagieren zu können.

Zuletzt muss die Darstellung eines Agenten durch ein Protokollnetz erörtert werden. Insbesondere ist dabei zu beachten, dass der Informationsaustausch zwischen Agenten auch abstrahiert werden würde und *von Nachrichten zu Konversationen* werden würde. Dabei ist zu beachten, dass die Möglichkeiten des proaktiven Verhaltens sowie die Nebenläufigkeit durch die Darstellung als Prozess eingeschränkt werden. Mulan-Protokolle können im Agenten durch den Aufruf von Sub-Protokollen modelliert werden.

Für den Nachrichtenversand können nach wie vor die **in** und **out** Transitionen Verwendung finden, welche durch eine entsprechende Semantik erweitert werden müssen. Dabei entsteht jedoch nun das Problem, dass der Austausch zwischen Nachrichten in dieser Form nicht mehr auf eine physikalische Plattform beschränkt bleiben muss. Da die Mulan-Architektur jedoch die Verwendung von synchronen Kanälen vorsieht, muss ggf. auf eine Verteilungslösung für synchrone Kanäle zurückgegriffen werden. Weitere Überlegungen zu dieser Problematik erfolgen später auf einer niedrigeren Abstraktionsebene im Unterabschnitt 9.1.

Dennoch sollte für die plattformübergreifende Kommunikation zwischen Agenten eine geeignete Konzeptualisierung gefunden werden. Zu diesem Zweck kann auf die Arbeiten zur Integration zwischen Agenten und Workflows (Reese, 2009; Wagner, 2018) zurückgegriffen werden. Das dort beschriebene Ergebnis besagt, dass durch den Einsatz spezieller Einheiten Agenten- und Workflowverhalten integriert beschrieben werden kann. Daraus folgt, dass die Synchronisation durch ein Konstrukt beschrieben werden können muss, welches der Workflow-Semantik grundlegend entspricht. Insbesondere die in Workflow-Netzen verwendeten Task-Transition aus (Aalst u. a., 1999) und (Jacob, 2002) ist hierbei von Interesse. Das Finden und auch das Spezifizieren dieses Konstruktes ist ebenfalls Teil und Beitrag dieser Arbeit.

Ein weiterer Aspekt, welcher von den Untersuchungen im Rahmen der Mulan-Architektur adressiert wird, umfasst die Mobilität von Mulan-Agenten (Köhler und Rölke, 2002; Köhler, Moldt und Rölke, 2003; Köhler und Farwer, 2007; Cabac, Moldt, Wester-Ebbinghaus u. a., 2009). Mulan-Agenten sollten Mulan-Plattformen beliebig wechseln können. Konkret wird dies durch den Transfer, die Vernichtung und die Erzeugung der entsprechenden Mulan-Agenten realisiert, während die konkrete Ausgestaltung abstrakt gehalten wird. Ein Prozesskalkül, welches ebenfalls die Grundidee mobiler Prozesse beschreibt ist das π-Kalkül (Milner, Parrow und Walker, 1992a,b; Milner, 1999). Diese Funktionalität sollte konsequenterweise daher auch in der neuen Architektur so für Plattformen zur Verfügung stehen.

4.4.3. Leitmetaphern

Da zumindest einige Teile der Architektur weitestgehend neu konzipiert werden müssen, kann es hilfreich sein, eine *Leitmetapher* einzuführen. Die bekannte Leitmetapher »Everything is an object« (dt.: Alles ist ein Objekt) geht auf Alan Kay (Kay, 1993) zurück. Analog dazu wurde dieser Ansatz von verschiedenen Forschern aufgegriffen, so formulierten beispielsweise (Wooldridge, 2009) oder (Rölke, 2004) das Äquivalent dazu mit Agenten: »Alles ist ein Agent«. Daran anschließend formulierte (Wagner, 2018) zur Integration von Workflows und Agenten: »Alles ist eine integrierte Einheit«.

Es ist zu überlegen, ob eine ähnliche Leitmetapher sinnvoll im Kontext dieser Arbeit ist. In den weiteren Untersuchungen wird sich schnell zeigen, dass der Begriff der Skalierbarkeit häufig im Kontext von Webservices betrachtet und erörtert wird. Daher wäre es äußerst hilfreich, auf Technologien und Konzepte von Webservicelösungen zurückgreifen zu können, um die zu erschaffende Architektur und ihre spätere Implementation detailliert zu beschreiben. In (Moldt, Ortmann und Offermann, 2004) wurde bereits dargestellt, dass sich die Mulan-Architektur in der Tat eins zu eins auf die konzeptuellen Grundlagen von Webservices abbilden lässt. Da die Mulan-Architektur ebenfalls als Ausgangspunkt für die betrachteten Aspekte dient, ist so ein entsprechendes Bindeglied gefunden. Die Interpretation ist in Abbildung 4.3 dargestellt.

Während die Skalierbarkeit der Plattformlandschaft einen entscheidenden und den primären Stützpfeiler der Arbeit ausmacht, bestehen jedoch auch viele Aspekte der Agentenorientiertheit und solche von Workflows. Aus diesem Grund soll im Rahmen der Arbeit insbesondere bei der Konzeptualisierung des Plattformmanagements die Perspektive der Webservices eingenommen werden, um Ergebnisse aus diesem Bereich bestmöglich integrieren zu können. Auf die entsprechend der einleitenden Sätze äquivalente Formulierung bezüglich Webservices wird dahingegen aus eben genannten Gründen explizit verzichtet.

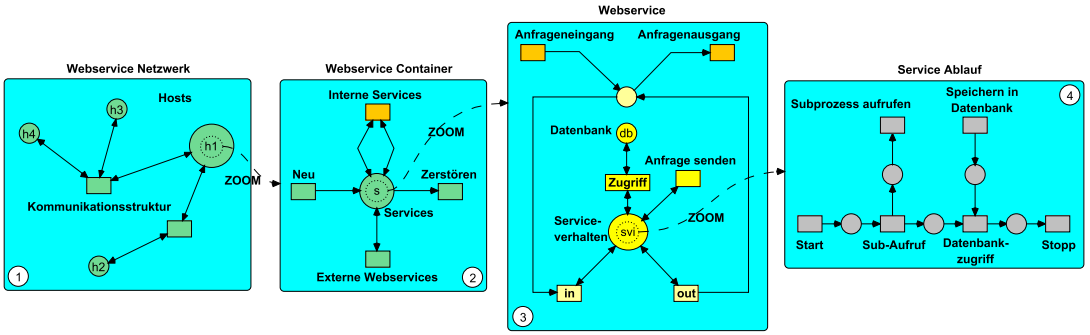

Abbildung 4.3.: Interpretation der MULAN-Architektur als Webservicestruktur. Entnommen, neu angeordnet und übersetzt aus (MOLDT, ORTMANN und OFFERMANN, 2004)

4.5. Aufbau und Herangehensweise im verbleibenden Teil der Arbeit

Im folgenden Hauptteil der Arbeit werden sowohl die Konzepte als auch die Realisierungen bezüglich Agentenkommunikation, Plattform sowie Plattformmanagement erarbeitet. Dabei umfasst der nächste Teil II die Erarbeitung eines abstrakten Konzepts zur Beschreibung eines entsprechend der Anforderungen entworfenen, skalierbaren Multiagentensystems. Der übernächste Teil III hingegen beschreibt das Konzept für eine mögliche Realisierung der Architektur. Im darauffolgenden Teil IV werden dann eine Reihe von Prototypen beschrieben, welche konkrete Technologien einsetzen, um das zuvor definierte Realisierungskonzept umzusetzen.

Die Reihenfolge der Betrachtungen folgt der Darstellung in Abbildung 4.4. Die Abbildung ist so zu lesen, dass bei der Erarbeitung der Konzepte der einzelnen Bestandteile zunächst von der kleinsten Einheit (Agenten) aus vorgegangen wird und danach die hierarchisch höher verorteten Bestandteile folgen. In der Erarbeitung der konkreten Realisierung der Bestandteile wird diese Reihenfolge umgekehrt.

Die Hintergründe zu dieser Darstellung ergeben sich wie folgt: In der Strukturierung der Konzepte definieren sich die Einheiten nach der Einheitentheorie über ihr Verhalten zur höher angeordneten Ebene, zu den gleichberechtigten Einheiten auf derselben Ebene und zu ihren untergeordneten Einheiten. Dabei sind untergeordnete Einheiten stets in der übergeordneten Einheit eingebettet. Somit muss sich eine Einheit den Gegebenheiten ihrer übergeordneten Einheit fügen. Da nun aber das Gesamtsystem entsprechend konzipiert werden soll, liegt es nahe, durch eine Betrachtung von der kleinsten Einheit aus nötige Schnittstellen an darüberliegende Ebenen zu definieren. So kann die übergeordnete Einheit unter

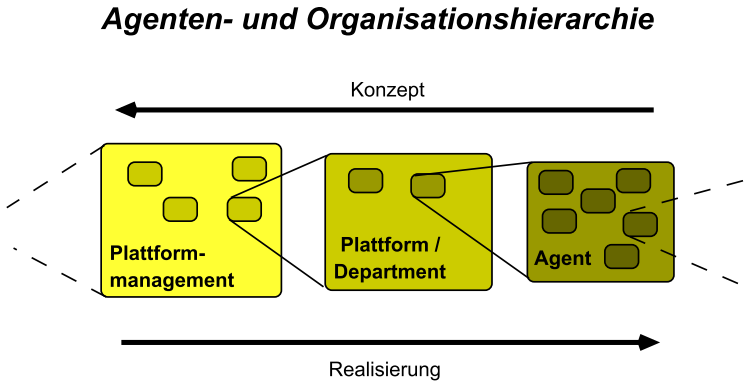

Abbildung 4.4.: Aufbau der Kapitel zur konzeptuellen Erarbeitung der Architektur und prototypischen Realisierung

Berücksichtigung der Bedürfnisse ihrer eingebetteten Einheit konzipiert werden und eine stimmige Gesamtarchitektur konstruiert werden.

Bei der Realisierung wird diese Reihenfolge umgekehrt. Da an dieser Stelle bereits die konzeptuellen Schnittstellen bekannt sind, können diese umgehend umgesetzt werden. Dadurch definiert sich eine konkrete Schnittstelle für die darunterliegenden Ebenen. In der Betrachtung der nächst niedrigeren Ebene kann dann der konkrete Konsum der angebotenen Schnittstelle modelliert werden.

4.6. Zusammenfassung - Anforderungsanalyse

In diesem Kapitel wurden die zentralen Anforderungen aufgeteilt in interaktionsbasierte, strukturelle, dynamische und Anwendungsdomänen-bezognene Anforderungen an das zu konstruierende System formuliert, um den Notwendigkeiten durch die Forschungsfragen gerecht zu werden. Beim Abgleich dieser Anforderungen mit den konkreten Systemen und Architekturen cloneMAP, MMAS2L und Mulan/ Capa ergab sich, dass keins der Systeme die Anforderungen erfüllen kann. Auf dieser Basis wurde durch eine Granularitätsverschiebung der Mulan-Architektur als Heuristik ein Grobentwurf für die Konstruktion einer weiteren Ebene im Mulan-Modell erstellt, welcher die Integration eines Plattformmanagements ermöglichen soll. Für die Realisierung des Plattformmanagements wurde motiviert, eine Webservice-basierte Perspektive einzunehmen. Abschließend wurde erläutert, dass in der Konzeptualisierung der Architektur die Bearbeitung von Agentenkommunikation bis zum Plattformmanagement erfolgt, während dies im Realisierungskonzept in umgekehrter Reihenfolge erfolgt.

Teil II.

Konzeption einer skalierenden Referenznetzsimulation

5. Plattformübergreifende Agentenkommunikation

In diesem Kapitel soll die generelle Kommunikationsform zwischen Agenten, die sich potentiell auf verschiedenen Plattformen befinden können, erörtert werden. Die Natur der Kommunikation hat direkte Auswirkungen auf die Anforderungen, die aus Sicht einzelner Agenten an die Plattform und das Plattformmanagement zu stellen sind. In Kapitel 4 wurde bereits motiviert, dass es bei einer Realisierung entlang der MULAN-Architektur hilfreich ist, Agentenkommunikation als Synchronisation darzustellen. Ferner wurde aufgezeigt, dass es hierzu ausreichend ist, die Semantik einer verteilten Task-Transition zu definieren.

Daher skizziert dieses Kapitel insbesondere eine mögliche Konkretisierung dieser verteilten Task-Transition sowie die zugehörigen Interaktionen, Rollen und die Anforderungen an Plattform und Plattformmanagement. Darüber hinaus werden in Vorbereitung auf die Konkretisierung in Kapitel 12 Möglichkeiten der Realisierung im Kontext von Webservices diskutiert.

5.1. Vorüberlegungen

Diese Arbeit adressiert insbesondere das Plattformmanagement und die Plattform selbst. Daher sind keine substantiellen Änderungen oder Neuerungen am Agentenmodell vorgesehen. Als ein Aspekt gilt es dennoch die Kommunikation zwischen Agenten auf den Plattformen zu betrachten, da diese fundamental mit Plattform und gegebenenfalls Plattformmanagement zusammenhängt. Dabei stellt sich die Frage, an welchen Stellen eine tiefergehende Betrachtung notwendig ist. Die FIPA stellt mit ihren Standards weitreichende Referenzwerke für die Kommunikation zwischen Agenten zur Verfügung. Im Kontext von Referenznetzen wurden diese bereits in der Arbeit (DUVIGNEAU, 2002) betrachtet. Die adressierte Arbeit beschreibt außerdem ein umfangreiches Nachrichtenmanagement und die Weiterleitung von entsprechenden Nachrichten an Agenten auf einer Plattform.

Dabei berücksichtigt der FIPA-Standard jedoch nicht die Modellierung des Nachrichtensystems in Hinblick auf plattformübergreifenden Einsatz, sondern lässt die Definitionen abstrakt. Ebenso adressiert das Nachrichtenmanagementsystem aus

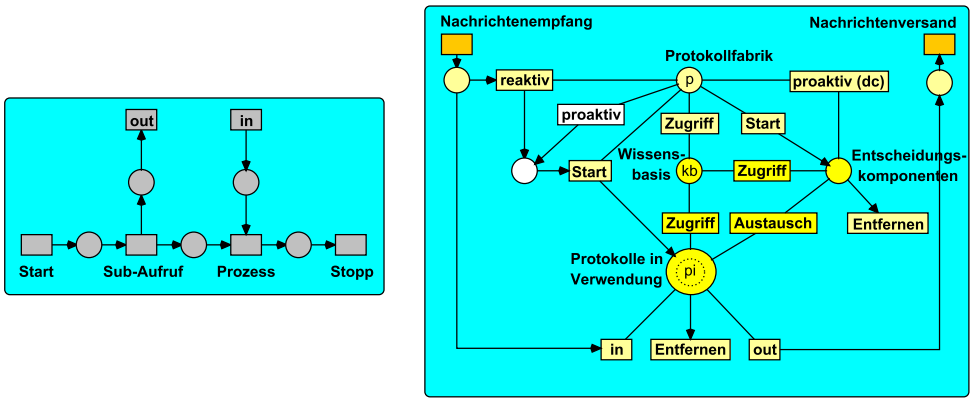

Abbildung 5.1.: Gesucht: Integration von Agent (links) und MULAN-Agent (rechts)

(DUVIGNEAU, 2002) den Einsatz innerhalb einer Plattform. Aus diesem Grund ist es notwendig, die referenznetzbasierte Kommunikation zwischen Agenten in Hinblick auf den plattformübergreifenden Einsatz zu betrachten.

Durch die Verschiebung der MULAN-Architektur ergibt sich eine Integration zwischen Agent und MULAN-Agent, wie in Abbildung 5.1 dargestellt. Bezogen auf die Kommunikation ergibt sich hierbei die Doppelung von nachrichtenbasierter Kommunikation und synchronisationsbasierter Kommunikation.

Dieser Zusammenhang ist auch abstrakt nachzuvollziehen: Aus der Darstellung einzelner Nachrichten wird die Darstellung ganzer Kommunikationen, welche durch Synchronisation abgebildet werden kann.

Da in der verschobenen Architektur der Agent auf die Ebene der Protokolle verschoben wird, wäre es für die Gesamtkonstruktion hilfreich, wenn auch so zwischen Agenten eine synchronisationsbasierte Kommunikation möglich wäre. Solange sich die Agenten auf ein und derselben Plattform befinden, können die zuvor genannten Technologien meist problemlos angewandt werden. Problematisch im verteilten Fall ist der Einsatz von Referenzsemantik. Eine einfache Lösung hierfür wäre die Annahme global eindeutiger Referenzen oder der Wechsel zum Einsatz von Namen anstelle von Referenzen. Dennoch ist hierbei eine globale Kontrolle nur schwer zu realisieren. In konkreten Realisierungen wäre innerhalb einer lokalen Einheit innerhalb der Anwendung die Installation einer lokalen Kontrollinstanz denkbar. Anstöße hierzu geben die Ideen aus dem Bereich der GALS (Global Asynchron, Lokal Synchron) Systeme, wie sie in Abschnitt 3.4.2 angesprochen wurden.

Wird die gleiche Situation über Plattformgrenzen hinweg betrachtet, gestaltet sie sich als schwieriger. Bei der Realisierung bieten sich direkt einige etablier-

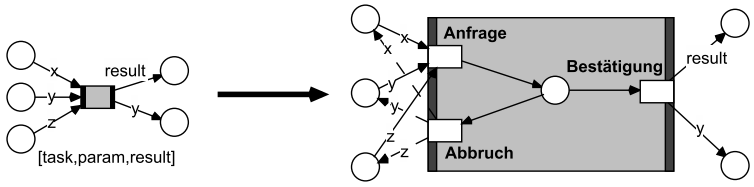

Abbildung 5.2.: Task-Transition. Vereinfacht und entnommen aus: (Wagner, 2018) nach (Jacob, 2002)

te Konzepte an. Zunächst ist hierbei der Remote Procedure Call (als Konzept) zu erwähnen. Ein Einsatz erfordert einen zentralen Verzeichnisdienst bzw. eine Registry. Im Normalfall werden statische Verbindungen zwischen Applikationen etabliert. Dieser Umstand ist jedoch sowohl mit der Agentenmobilität als auch mit der Plattformmobilität nur schwer zu vereinen. Die Agentenmobilität könnte durch entsprechende Berücksichtigung innerhalb der Plattform abgefangen werden, die Mobilität der Plattform selbst jedoch nur schwer. Daher ist der Remote Procedure Call tendenziell eher ungeeignet als Konzept zur Kommunikation.

Als weitere Möglichkeit wäre die Verwendung eines zustandslosen allgemeinen Kommunikationsprotokolls, wie es bei den Schnittstellen der Plattform gefordert wurde, denkbar. Dabei gilt es jedoch zu beachten, dass im Allgemeinen die Synchronisation zwischen referenznetzbasierten Agenten deutlich komplexer ausfällt. Dennoch sollte die Möglichkeit dieser Kommunikationsform für einfache Fälle nicht gänzlich vernachlässigt werden. Referenznetzbasierte Synchronisation erfolgt über synchrone Kanäle, welche zur Auswertung Unifikation erfordern. Unifikation ist aus Komplexitätssicht nicht leicht zu berechnen, wie beispielsweise in (Knight, 1989) oder aktueller (Akutsu u. a., 2017) beschrieben wird. Wird die Unifikation nun verteilt, entsteht dadurch ein beträchtlicher Nachrichtenoverhead, welcher im Allgemeinen nicht tragbar ist.

Es liegt daher nahe, die Agentensynchronisation aus einem anderen Blickwinkel zu betrachten. (Reese, 2009) und (Wagner, 2018) beschrieben in ihren jeweiligen Arbeiten die Integration zwischen Workflow-Management-System und Agenten-Management-System. Ohne auf die konkreten Details einzugehen, da diese überwiegend auf der Ebene der Agenten bzw. Workflows formuliert sind, kann jedoch daraus abgeleitet werden, dass auch Lösungen für Workflow-(Petri)netze eingesetzt werden können, um die Synchronisation zu modellieren. Im Kontext von Workflow-Netzen existiert ein zentrales Konzept: Das der Task-Transition, welche in (Aalst u. a., 1999) vorgestellt wurde.

Die Semantik einer Task-Transition ist in Abbildung 5.2 dargestellt. Die wesentliche Eigenschaft besteht darin, dass Task-Transitionen fehlschlagen können. Im Falle eines Fehlschlages werden alle konsumierten Marken zurück in den Vorbe-

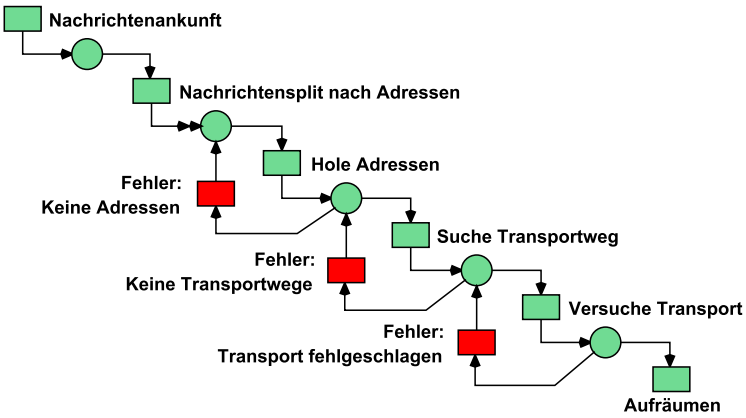

Abbildung 5.3.: Agent Communication Channel Implementation in Capa. Abstrahiert und entnommen aus: (Duvigneau, 2002, Seite 106)

reich gelegt. Die Task-Transition erzeugt damit eine Form von Atomizität, da alle beinhalteten Operationen vollständig oder gar nicht abgeschlossen werden. Interessant ist dabei auch die Eigenschaft, dass dadurch eine Situation innerhalb der Task-Transition bestehen muss, zu der die Reversibilität der Transition aufgegeben wird. Diese Situation kann als »Point of no return« (dt. »Punkt, ab dem es kein Zurück gibt«) bezeichnet werden. Sie befindet sich bei der abgebildeten beispielhaften Task-Transition bei der Transition »Bestätigung«. In Kombination mit Referenznetzen wurden Task-Transitionen in (Jacob, 2002) betrachtet.

Rückblickend kann das in (Duvigneau, 2002) beschriebene Nachrichtenmanagement insgesamt ebenfalls als Task-Transition betrachtet werden. Der grundlegende Ablauf ist daher noch einmal in Abbildung 5.3 dargestellt. Dabei wird der Point of no return nach dem Split der Nachrichten durch die flexible Kante erreicht. Diese Operation ist nicht direkt reversibel, da hierzu alle Marken von einem Platz entfernt werden müssten. Dies wiederum benötigt einen Test auf Leerheit des Platzes, welcher seinerseits eine Inhibitorkante bedingen würde. Daher muss nach dem Split die Abarbeitung vollständig abgeschlossen werden. Zum Zeitpunkt der Arbeit lagen jedoch die Arbeiten (Reese, 2009) und (Wagner, 2018) noch nicht vor, sodass der Zusammenhang noch unbekannt war.

Daraus folgt, dass eine verteilte Task-Transition das fehlende Bindeglied der plattformübergreifenden Agentenkommunikation darstellen kann. Diese sollte nach Möglichkeit kompatibel mit Webservice-Lösungen und Konzepten sein, da diese Perspektive während der Realisierung eingenommen werden soll, wie bereits in Abschnitt 4.4.3 beschrieben. Insgesamt kann somit als Anforderung festgehalten werden: Es ist notwendig, eine verteilte Task-Transition zu definieren und die Möglichkeit zur Implementation geeignet aufzuzeigen.

Wie zuvor motiviert, kann für sehr einfache Nachrichtenübermittlungen der Overhead und die Zusicherung durch die Task-Transition entfallen. Somit kann ebenfalls als Anforderung formuliert werden: Für einfache plattformübergreifende Agentenkommunikationen sollen direkte Übertragungswege möglich sein. »Einfach« definiert sich dabei als unidirektionaler Informationsaustausch mit nur einer Nachricht.

5.2. Kommunikationsformen

In klassischen Agentenanwendungen findet Interaktion und Informationsaustausch zwischen Agenten in der Form von Nachrichten statt. Konventionen wie die der FIPA definieren üblicherweise Schnittstellen und Aufbau der Nachrichten. Im Falle der FIPA und damit auch Capa definieren MTS (Message Transport System) und ACCs (Agent Communication Channels) die Übertragung. Dabei ist die Ausgestaltung jedoch auch relativ abstrakt. Capa implementiert MTS und ACC auf Basis von Referenznetzen. Es liegt daher nahe, diesen Ansatz auf die hier betrachtete Skalierbarkeitsfrage zu erweitern.

Aus diesem Grund soll die Plattform-lokale Kommunikation hier nicht weiter vertiefend adressiert werden. Für die wesentlichen Mechaniken kann beispielsweise auf die Ausführungen zu Capa in (Duvigneau, 2002) zurückgegriffen werden. Diese wurden auch bereits in Abschnitt 5.1 aufgearbeitet.

Im Allgemeinen wurde durch die Verschiebung der Mulan-Architektur eine erhöhte Abstraktionsebene eingenommen. Während in klassischen Modellen einzelne Nachrichten von Agenten betrachtet werden, liegt es nahe, in der übergeordneten Sicht ganze Kommunikationen zu betrachten. Dies folgt im Wesentlichen (formal) aus der Darstellung durch synchrone Kanäle in der Protokoll-Ebene der Mulan-Agenten, welche zu Agenten in der neuen Architektur werden. Eine erweiterte Begründung, warum es sich dabei um eine sinnvolle Abstraktion handelt, gibt Abschnitt 5.2.2.

5.2.1. Kommunikation über Plattformgrenzen hinweg

Bei der Kommunikation über Plattformgrenzen hinweg treten die im Allgemeinen für verteilte Systeme geltenden Schwierigkeiten auf. Eine mögliche Schwierigkeit kann die (temporäre) Unerreichbarkeit von Teilnehmern sein, beispielsweise durch Netzwerkausfälle oder Abstürze. Darüber hinaus ist die Kommunikation über Plattformgrenzen hinweg ungleich zeitaufwändiger, da der Versand der Information im Zweifel über ein physikalisches Netzwerk erfolgen muss. Eine spezielle Besonderheit im Kontext von Petri- und Referenznetz-basierten Modellen

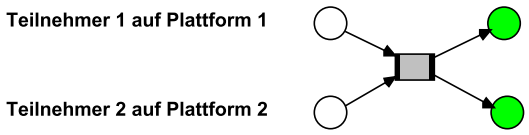

Abbildung 5.4.: Verteilte Task-Transition: Ausgangspunkt der Überlegungen

ergibt sich durch die wünschenswerte Eigenschaft der True-Concurrency-Semantik. Diese verhindert effektiv den Einsatz synchroner verteilter Protokolle, da sich das System sonst im Zweifel in die Richtung der Step-Semantik bewegt, bei der aktivierte Transitionen im Gesamtsystem gemeinsam in einem Schritt feuern.

In Abschnitt 5.1 wurde bereits motiviert, dass ein valider Ausgangspunkt durch die Konstruktion einer verteilten Task-Transition besteht. Dazu ist es hilfreich, diesen Ansatz in möglichst simpler Form in Augenschein zu nehmen. Der Ausgangspunkt ist in Abbildung 5.4 dargestellt.

5.2.2. Nachrichten und Kommunikation

Für die Konstruktion der verteilten Task-Transition ist es hilfreich, Annahmen bezüglich der Modalitäten eines Informationsaustausches machen zu können. Modalitäten umfassen dabei alle Aspekte des Austausches, wie Abhängigkeiten der ausgetauschten Informationen zueinander, die Anzahl der Teilnehmer, aber auch den Grad an Garantie, welchen das Transportmedium bereitstellen muss. Wie eingehend erläutert, unterscheiden sich Kommunikationen und der einfache Versand von Nachrichten in ihrer Abstraktionsebene. Intuitiv kann eine Kommunikation aus einer einzelnen Nachricht bestehen, aber auch deutlich komplexer sein.

Bei der Vorstellung hilft hierbei eine Abbildung auf ein alltägliches Beispiel. Der Versand eines einfachen Briefes betrifft zwei Kommunikationspartner, der Informationsaustausch ist unilateral und die Zusicherung der Zustellung nicht gegeben. Erfolgt der Versand als Einschreiben, kann zumindest die Zustellung garantiert werden. Verwenden die beiden Kommunikationspartner ein Telefon, so kann der Informationsaustausch bilateral erfolgen. Dabei kann beobachtet werden, dass die Zusicherung der Zustellung implizit erfolgt, da andernfalls keine Antwort erfolgen kann, welche Bezug auf die Information der Gegenpartei nimmt. Bezieht einer der Gesprächspartner eine weitere Person im Raum mit in das Gespräch ein, erfolgt die Kommunikation zwischen drei Gesprächspartnern und zwei Kanälen (Telefon und Raum). In einem anderen Beispiel können Fernsehzuschauer der Ansprache eines Redners zuhören. Die Kommunikation erfolgt dabei mit unilateralem Informationsaustausch und ebenfalls ohne Zusicherung der Zustellung, da nicht jeder

potenzielle Adressat in diesem Moment fernsehen muss. Gleichzeitig nehmen aber sehr viele Teilnehmer an der Kommunikation teil.

Es lässt sich also zusammenfassen, dass multilaterale Kommunikation mit aufeinander aufbauendem Informationsaustausch stets Zustellungszusicherung bedarf. Bauen die Informationen nicht aufeinander auf, kann die Kommunikation auch als zwei unilaterale Kommunikationen aufgefasst werden, die sich zufällig überlagern. Multilaterale Kommunikation mit aufeinander aufbauendem Informationsaustausch lässt sich hervorragend mittels synchronen Kanälen abbilden. Dabei kann die Ausführung eines synchronen Kanals gleich vollständige Kommunikationen abbilden. Eine Nachbildung zwischen verschiedenen Plattformen sollte also diese Form der Zusicherung der Zustellung in jedem Fall unterstützen.

Eine Zusicherung der Zustellung ist jedoch unlängst aufwändiger als der einfache Versand von Nachrichten. Zusätzlich lässt sich beobachten, dass einige Arten der Kommunikation diese Zusicherung überhaupt nicht benötigen. Für diese Fälle ist die Semantik der verteilten Task-Transition deutlich übersetzt. Da multilaterale Kommunikation implizit die Zustellungszusicherung erfordert, kann geschlussfolgert werden, dass Kommunikation ohne Zusicherung unilateral ist.

Da Effizienz gerade in einem verteilen System von erhöhter Wichtigkeit ist, sollte grundlegend zwischen den zwei herausgearbeiteten Abstraktionsstufen bzw. Kommunikationsformen unterschieden werden:

- Unilaterale Kommunikation ohne Garantie der Zustellung (Realisierung als klassische Nachricht)
- Multilaterale Kommunikation mit Garantie der Zustellung (Realisierung als Synchronisation)

Beides erfordert Bestandteile innerhalb des Plattformmanagements. Für Unilaterale Kommunikation ohne Garantie der Zustellung kann dabei auf die Konstruktion der Mulan-Plattform zurückgegriffen werden. Als Referenznetz dargestellt handelt es sich hierbei um eine einfache Transition zwischen zwei Mulan-Agenten bzw. zwischen zwei Plattformen im neuen Modell.

5.2.3. Multilaterale Kommunikation mit Zusicherung der Zustellung

Im vergangenen Abschnitt wurde die Notwendigkeit des Einsatzes einer verteilten Task-Transition auf multilaterale Kommunikation mit Garantie der Zustellung eingeschränkt. Die Garantie der Zustellung kann abstrakt als Garantie eines konsistenten Zustandes inklusive der zur Verfügung gestellten Informationen im entfernten System gesehen werden. Dabei können viele Zwischenschritte bestehen

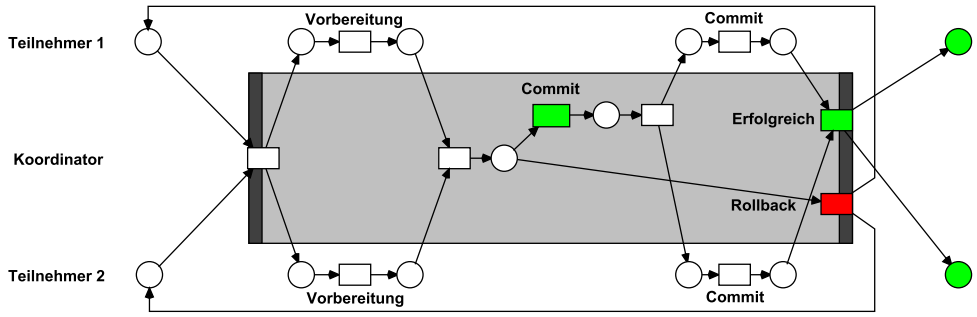

Abbildung 5.5.: 2-Phasen-Commit: Kontrollfluss dargestellt als Task-Transition

und die finalen Änderungen sollen nach den Zwischenschritten bei allen Kommunikationspartnern feststehen. Damit entspricht die Anforderung aber im Wesentlichen denen einer verteilten Transaktion, bei der alle Teilnehmer am Ende ein gemeinsames Commit erreichen sollen.

Für die Realisierung von verteilten Transaktionen bestehen viele Überlegungen und Ansätze. Einer der klassischsten ist hierbei der Zwei-Phasen-Commit. Während viele Abwandlungen und Verbesserungen oder gänzlich andere Ansätze existieren, soll der Zwei-Phasen-Commit zunächst genügen. Für einen Entwurf ist der Zwei-Phasen-Commit in Abbildung 5.5 als Petrinetz innerhalb einer Task-Transition dargestellt. Dabei ist nur der Kontrollfluss modelliert, nicht aber der Datenaustausch zwischen den Kommunikationspartnern bzw. Transaktionsteilnehmern. Die Idee zum Einsatz des Zwei-Phasen-Commit wurde inspiriert durch die Implementation des Distribute Plugins des Simulators Renew, welches in (M. Simon, 2014) beschrieben wird. Dort wurde der Simulationsalgorithmus des Simulators in ähnlicher Weise erweitert.

Bezogen auf den Agenten- bzw. Referenznetzkontext kann daher im Wesentlichen die Gesamtkommunikation in drei Phasen unterteilt werden:

- Lesende Phase: Teilnehmer tauschen Informationen aus.
- Entscheidungsphase: Es stehen genug und passende Informationen zur Verfügung und die Kommunikation kann so stattfinden.
- Aufarbeitungsphase: Die Entscheidung ist gefallen und alle Teilnehmer persistieren den Austausch lokal.

Diese Aufteilung wird im späteren Verlauf der Betrachtungen noch besondere Bedeutung erhalten. Mit diesem Ausgangspunkt können weitere Details herausgearbeitet werden. Somit sollte ein Koordinator im Netz existieren und die Kommunikation die drei beschriebenen Phasen modellieren. Die Existenz eines Koordinators wird zu einigen Zielkonflikten im Kontext der Modellierung der Plattform

in Kapitel 6 insbesondere bezüglich der Abhärtung führen. Diese führen jedoch nicht zur unmittelbaren Inkompatibilität und können dennoch erfolgreich integriert werden, wie die Realisierung zeigt, welche in Kapitel 12 beschrieben wird. Aus diesem Grund kann die Annahme der Existenz eines Koordinators für weitere Überlegungen angenommen werden.

5.3. Entwurf der Kommunikationsinteraktionen

Nachdem im vergangenen Kapitel die grobe Struktur der Kommunikation aufgeschlüsselt und beschrieben wurde, konkretisiert dieses Kapitel die Natur der Kommunikation weiter. Das erklärte Ziel dabei umfasst die Beschreibung geeigneter Interaktionssequenzen für die Definition des jeweiligen Kommunikationsprotokolls.

5.3.1. Identifikation und einfache verteilte Kommunikation

Zunächst ist es notwendig, die grundlegende Identifkation von Agenten im Gesamtsystem zu ermöglichen. Dazu sollen neben Plattform-lokalen Alias auch global eindeutige Identifikatoren zum Einsatz kommen. Beim Erzeugen eines neuen Agenten durch die Plattform wird dem globalen Plattformmanagement die Identifikation zur Verfügung gestellt. Folglich muss das Plattformmanagement eine Form von Verzeichnisdienst bereitstellen. Dieser Verzeichnisdienst ist konzeptuell nicht unterschiedlich zum klassischen Verzeichnisdienst DF (Directory Facilitator) nach FIPA, wie er beispielsweise auch im Rahmen von Capa beschrieben ist. Verzeichnisdienste sind vielfältig verfügbar und Identifikationslösungen wie UUIDs existieren. Daher wird an dieser Stelle auf die konkrete Ausgestaltung der Verzeichnislösung verzichtet und lediglich die entsprechende Anforderung an das Plattformmanagement festgehalten. Der Ablauf ist grafisch im oberen Teil von Abbildung 5.6 dargestellt.

Der untere Teil der Abbildung zeigt den Ablauf einer einfachen unidirektionalen Konversation mit Lokalisation des Gesprächspartners und ohne Zusicherung der Zustellung. Dabei wird vorausgesetzt, dass »Agent 1« bereits über die Existenz von »Agent 2« informiert ist. Dies kann beispielsweise durch vorangegangene Kommunikation (auf einer anderen Plattform) erfolgt sein oder aber dadurch, dass das Pseudonym »Agent 2« allgemein bekannt ist und damit auch »Agent 1« statisch bekannt ist. »Agent 1« bittet seine Plattform, eine Nachricht an »Agent 2« zu senden. Die Plattform erfragt den Aufenthaltsort von »Agent 2« beim Plattformmanagement und schickt der entsprechenden Plattform die Nachricht von »Agent 1«. Das Plattformmanagement fungiert dabei als Kommunikationsmedi-

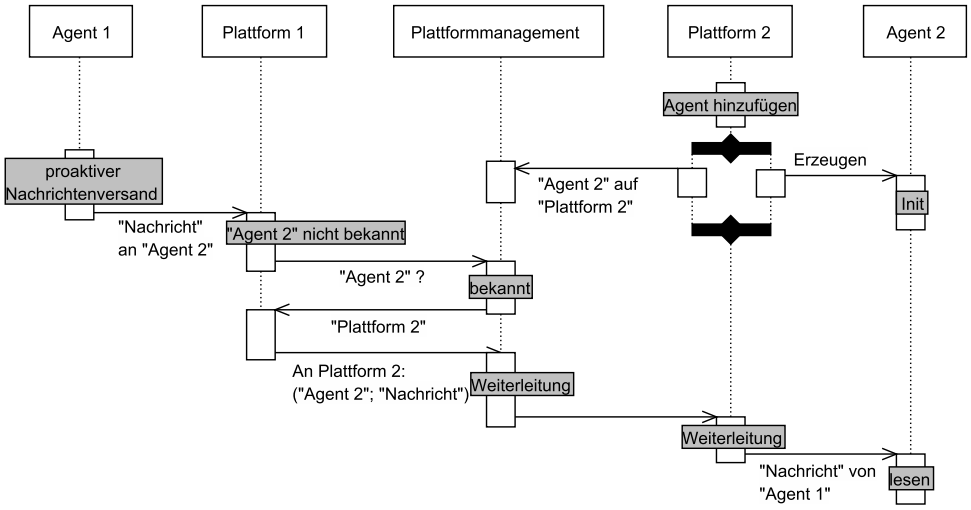

Abbildung 5.6.: »Einfacher« verteilter Nachrichtenversand mit Identifikation

um, erfüllt aber kein intelligentes Routing. Diese Umsetzung beachtet bereits die intendierte Umsetzung mittels Webservices.

Im Kontext von Webservices kommen, wie in Abschnitt 2.6.1 beschrieben, häufig sogenannte »dumb pipes« zum Einsatz, welche kein aktives Routing zum Zielservice unterstützen (sollten). Nach der Weiterleitung durch die Plattform von »Agent 2« kann dieser die Nachricht empfangen. Insgesamt muss für diesen Kommunikationsweg vom Plattformmanagement lediglich der einfache Kommunikationskanal bereitgestellt werden.

5.3.2. Verteilte Kommunikation mit Zusicherung

In diesem Abschnitt soll der skizzierte Kommunikationsablauf aus Abbildung 5.5 auf eine konkrete Interaktion zwischen Agenten abgebildet werden. Wie zuvor erläutert, handelt es sich hierbei um eine Kommunikation mit aufeinander aufbauender Informationsbereitstellung. Dies entspricht der Modellierung durch einen synchronen Kanal im Modell des Multiagentensystems.

Abbildung 5.7 zeigt exemplarisch den Ablauf der Interaktion mit zwei Gesprächsteilnehmern und einer Hin- und einer Rückrichtung des Informationsaustausches. Das System ist jedoch beliebig für weitere Austausche innerhalb der Konversation erweiterbar. Ferner wurde hier auf die Darstellung der Weiterleitung durch den Plattformmanager verzichtet, da dieser wenig Inhalt beiträgt, die Darstellung aber verkomplizieren würde. Jede Kommunikation zwischen Plattformen und / oder Ablaufmanager wird vom Plattformmanager weitergeleitet. Darüber hinaus

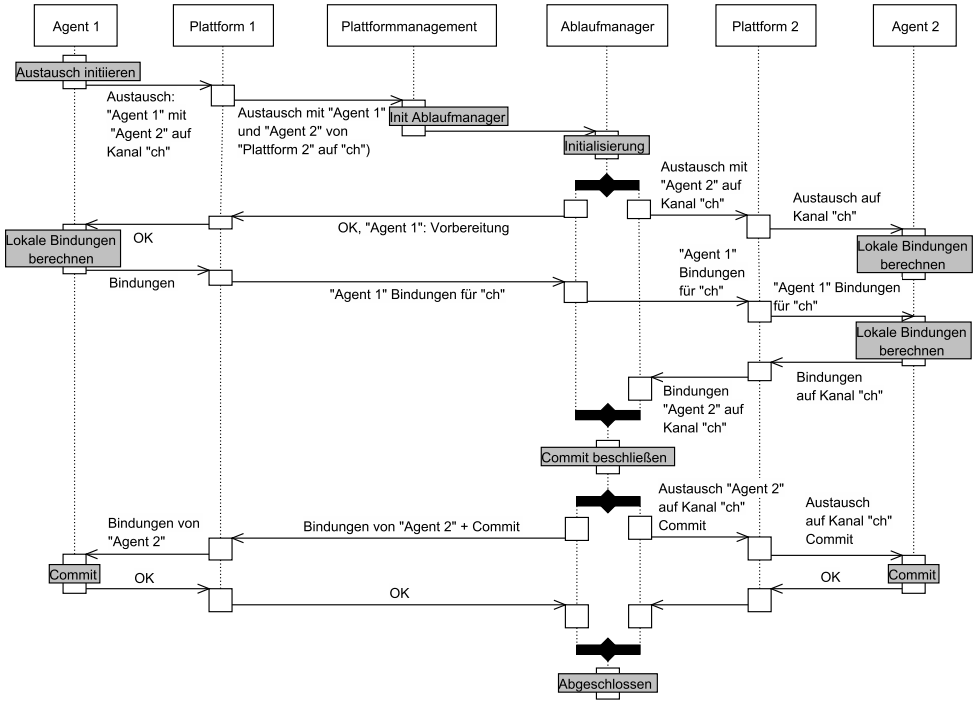

Abbildung 5.7.: Komplexer Austausch ohne Identifikation

wird die Registrierung von »Agent 2« nicht dargestellt, da diese identisch zum dargestellten Ablauf in Abbildung 5.6 ist.

Nach der Initiierung des Austausches durch »Agent 1« wird vom Plattformmanagement eine neue Instanz eines Ablaufmanagers gestartet. Dieser sorgt dafür, dass alle Kommunikationsteilnehmer mit der lokalen Vorbereitung beginnen. Auf Basis des Petrinetz-Modells der jeweiligen Agenten bedeutet dies die Berechnung von jeweils lokalen Bindungen an der entsprechenden Transition. In diesem Beispiel werden die Bindung von »Agent 1« sodann über das Plattformmanagement und den Ablaufmanager zu »Agent 2« übertragen. Durch die neuen Informationen von »Agent 1« kann »Agent 2« nun weitere Bindungen berechnen und das Ergebnis zurücksenden. Im Beispiel ist damit eine volle Bindung berechnet und die Kommunikation abgeschlossen. Der Ablaufmanager kann somit mit den weiteren Informationen von »Agent 2« beide Agenten benachrichtigen, dass die Information lokal persistiert werden müssen. Auf der Ebene des Petrinetz-Modells bedeutet dies, dass die gefundene globale Bindung zum Feuern der Transitionen verwendet werden kann und bei den jeweiligen Agenten entsprechende neue Marken produziert werden können. Nach Abschluss der Persistierung ist der Vorgang abgeschlossen und der Ablaufmanager kann beendet werden.

Für die weiteren Überlegungen bezüglich der komplexen Kommunikation zwischen Agenten ist dieses Referenzmodell äußerst hilfreich, da es die komplexe vollwertige Berechnung eines synchronen Kanals über Plattformgrenzen hinweg auf eine handhabbare Größe reduziert. Dies entsteht im Wesentlichen durch die Reduktion auf einzelne Kommunikationsrunden. Diese Einschränkung bezüglich verteilter synchroner Kanäle ist nicht neu, sondern wurde so auch bereits in der Implementation des Distribute Plugins in Renew in (M. Simon, 2014) beschrieben. Sie bekommt in diesem Kontext jedoch jetzt ein solides Fundament in Form der Beschreibung der Kommunikation zwischen Agenten.

Die Herleitungen dieses Kapitels stellen einen der Beiträge des Autors im Rahmen der Arbeit dar.

5.3.3. Ausfallresistenz

Wie bereits zuvor motiviert, ist der Zwei-Phasen-Commit bereits eine seit Jahrzehnten existierende Technologie. Das allein macht sie selbstverständlich nicht ungeeignet für den hier intendierten Einsatz, dennoch weist der Ansatz gerade im Kontext des Einsatzes in einem skalierenden System Nachteile auf. Auch schon damals diskutierte Probleme sind beispielsweise die (frühzeitige) Aufgabe der Teilnehmerautonomie. Sobald ein Teilnehmer meldet, dass seine Vorbereitung abgeschlossen ist, übergibt er die Kontrolle über den lokalen Commit an den Koordinator. Daher sind Ausfälle in diesem Zusammenhang als besonders kritisch zu

betrachten. Mit steigender Zahl der Teilnehmer innerhalb der verteilten Transaktion steigt somit auch das Risiko für Ausfälle.

Ferner ist der Zwei-Phasen-Commit ein Konsistenz-orientiertes Protokoll. In der Anforderungsanalyse und in den Grundlagen wurde bereits motiviert, dass ein Verfügbarkeits-orientiertes Protokoll sowohl in einem skalierenden System als auch in einer Webservice-basierten Realisierung Vorteile aufweist. Folglich sollte die abschließende Konzeptualisierung nicht auf Basis des Zwei-Phasen-Commits erfolgen. In Abschnitt 5.2.3 wurde bereits herausgearbeitet, dass der grobe Ablauf der verteilten Simulation in die drei Bestandteile aufgeteilt werden kann. Zunächst erfolgt die lesenden Phase bzw. die Bindungssuche, danach die Entscheidungsphase bzw. Beurteilung, ob eine Bindung gefunden wurde und schlussendlich die Aufarbeitungsphase bzw. dem jeweiligen lokalen Feuern der Transitionen. Aus diesem Grund sollte ein geeignetes aktuelles Verfahren auf Basis von Eventual Consistency zum Einsatz kommen.

Auf die Anforderung zu diesen drei Bestandteilen passt das Saga-Pattern hervorragend. Es wurde in dieser Arbeit bereits im Grundlagenkapitel in Abschnitt 2.5.2 eingeführt. Die drei Bestandteile beim Saga-Pattern umfassen mit der dortigen Terminologie kompensierbare Transaktionen, eine Pivottransaktion und wiederholbare Transaktionen. Semantisch handelt es sich dabei um äquivalente Konstrukte. Die einzige Problematik besteht darin, dass Sagas des Saga-Patterns zumeist mit Zustandsmaschinen beschrieben werden und daher eine sequenzielle Natur aufweisen. Die Kommunikation zwischen Agenten ist hier aber explizit und bewusst als nebenläufiges System konstruiert worden. (Lanese und Zavattaro, 2009) beschreiben in einem Prozesskalkül die Erweiterung des Saga-Patterns auf ein paralleles Modell. Aus diesem Grund sollte es generell möglich sein, auch eine nebenläufige Variante des Saga-Patterns zu definieren. Genau diesem Ansatz werden die Ausführungen in Kapitel 12 folgen, um eine Erweiterung des Saga-Patterns auf nebenläufige und petrinetzbasierte Modelle zu ermöglichen.

Für das Konzept soll es daher allerdings genügen, wenn Plattform und Plattformmanagement die gegebenen Rahmenbedingungen für die Implementation einer sagabasierten Kommunikation erfüllen. Konkret handelt es sich dabei um das Vorhandensein einer Form von Nachrichtenbroker sowie der Bereitstellung eines Orchestrators. Der Orchestrator hält alle Informationen, die zum korrekten Ablauf der Saga notwendig sind. Die puristische Alternative des Saga-Patterns auf Basis von vollständig dezentraler Choreografie kann hier nicht eingesetzt werden, da die Kommunikationsabläufe erst dynamisch aus der Agenteninteraktion entstehen und nicht statisch bekannt und damit auch nicht in den Agentendefinitionen verwurzelbar sind.

5.4. Anforderungen an Plattform und Plattformmanagement

Zusammenfassend werden in diesem Abschnitt noch einmal alle Anforderungen an Plattform und Plattformmanagement, welche sich aus der Agentenkommunikation ergeben, festgehalten. Zunächst muss die Plattform in der Lage sein, entfernte Agenten zu adressieren und ihre Netzwerkkoordinaten (die gegenwärtige Plattform) beim Plattformmanagement anzufragen. Darüber hinaus muss die Plattform grundlegende lokale Kommunikation ermöglichen, beispielsweise durch ein System, wie das Messaging System, welches in Capa implementiert ist. Um die Identifikation von Agenten zu ermöglichen, wird zusätzlich ein Verzeichnisdienst auf Plattformmanagementebene benötigt. Dabei handelt es sich im Wesentlichen um einen Directory Facilitator nach FIPA-Standard.

Für einfache verteilte Kommunikation muss eine einfache Transportmöglichkeit vorhanden sein. Dabei genügt es, wenn eine einfache Übertragung gewährleistet wird. Ein eigenes, komplexes Lokalisieren und Ansteuern von Plattformen während des Versands ist nicht erforderlich.

Für anspruchsvollere Kommunikation zwischen Agenten ist darüber hinaus zusätzlich eine Plattform für Instanzen des Ablaufmanagers wünschenswert. Des Weiteren muss diese Art Ablaufmanager ebenfalls vom Plattformmanagement bereitgestellt werden. Zusätzlich zum Ablaufmanager ist eine Nachrichtenbroker-Implementation erforderlich. Um das Ausfallrisiko gering zu halten, bietet es sich an, eine replizierende Variante zu bevorzugen. Nachrichten sollen dort vorgehalten werden und beliebig abgerufen werden können, wenn die abrufende Plattform in der Lage ist, Nachrichten zu empfangen.

Bei Betrachtung des Nachrichtenbrokers fällt auf, dass ein Verzeichnisdienst ebenfalls als Broadcast-Nachricht realisiert werden kann. Bei Registrierung wird diese Nachricht im Nachrichtenbroker persistiert. Wird nun ein bestimmter Agent gesucht und befindet sich dieser nicht im lokalen Cache, so können sukzessive alle Nachrichten der Registrierung in chronologisch umgekehrter Reihenfolge durchsucht werden, um nach der Adresse zu forschen. Eine effizientere Implementation würde den Cache durch die Nachrichten stets aktuell halten, sofern die Speicherkapazität der Plattform dies zulässt. Aus diesen Überlegungen entsteht nun zusätzlich die Anforderung, bei Bedarf und Wunsch doppeltes Zustellen von Nachrichten zu ermöglichen. Als Gegenleistung kann in diesem Falle der Verzeichnisdienst entfallen.

5.5. Zusammenfassung - Agentenkommunikation

Dieses Kapitel hat die Anforderungen an die architekturellen Entscheidungen bezüglich Plattform und Plattformmanagement aus Sicht der Agentenkommunikation adressiert. Dabei wurde zwischen einfacher und komplexer Kommunikation unterschieden, wobei für einfache Kommunikation eine simple Weiterleitung durch das Plattformmanagement genügt, während komplexe Kommunikation eine Zusicherung der Zustellung benötigt. Die Zusicherung wurde auf Basis einer verteilten Task-Transition und des Zwei-Phasen-Commits entworfen. Jedoch wurde sie aufgrund der Nachteile des Zwei-Phasen-Commit-Protokolls nicht abschließend mit diesem spezifiziert. Stattdessen wurde motiviert, eine zu definierende Abwandlung des Saga-Patterns einzusetzen, welche Nebenläufigkeit erlaubt und Eventual Consistency unterstützt.

6. Skalierbare Agentenplattformen

Zur Konstruktion spezieller Agentenplattformen, welche sich gut für den Einsatz in einem skalierenden System eignen, wurden entsprechende Anforderungen herausgearbeitet. Dabei soll in diesem Kapitel nun die abstrakte Architektur einer solchen Plattform hergeleitet werden. Die Überlegungen stützen sich im Wesentlichen auf die Cloud-Native-Architektur und die bereits in Capa eingesetzte Grundüberlegung, Plattformen wie eigene Agenten zu behandeln. Die Konzeption der Plattform auf dieser Basis stellt einen der Beiträge durch den Autor dieser Arbeit dar.

Dazu werden ähnlich zum vergangenen Kapitel die genauen Funktionsumfänge, soweit sie auf dieser Ebene beschreibbar sind, aufgenommen und die Interaktionen zwischen den Komponenten definiert.

6.1. Eine skalierbare Architektur

Die Anwendungsarchitektur der Plattform bildet einen entscheidenden Stützpfeiler für die Skalierbarkeit der verteilten Referenznetzsimulation. Die Plattform bildet die größte Ausprägung einer einzelnen Anwendung, daher liegt es nahe, Entwurfsmuster auf der Ebene von Applikationen heranzuziehen. Es wurden verschiedene Architekturstile im Rahmen des Grundlagenkapitels vorgestellt.

Das erklärte Ziel dieses Abschnittes umfasst die kritische Auseinandersetzung mit verschiedenen Architekturstilen, welche bereits für skalierende Systeme eingesetzt werden und die nachfolgende Adaption eines der Stile für das Konzept der skalierenden Agentensimulation. Dabei sollen weitere Konstruktionsdetails für die Integration von Mulan-Plattform und Mulan-Agent herausgearbeitet werden. Die Prämisse ist dabei, dass die Konstruktion der neuen Plattform die Konstruktion einer Anwendung für skalierbare Systeme ist. Daher sollten sich bestimmte Umstände und Lösungen aus bekannten Architekturen übertragen lassen.

Klassische Architekturansätze wie das Schichtenmodell legen wenig Fokus auf eine Ausführbarkeit in dynamischen Umgebungen. Hervorragende Synergie weisen dagegen beispielsweise die Ansätze der Zustandslosigkeit, welche jedoch nur einen Aspekt einer Anwendung betrifft, sowie die hexagonale Architektur, die Twelve-Factor-App, sowie der Ansatz der Cloud-Nativity auf. All diese Archi-

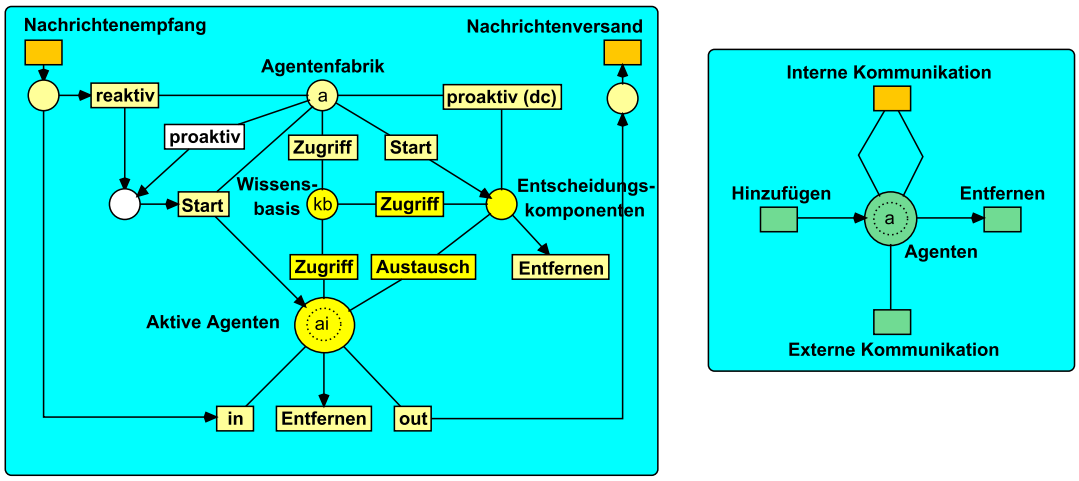

Abbildung 6.1.: Gesucht: Integration von Plattform (links) und Mulan-Plattform (rechts)

tekturansätze sind nicht unmittelbar austausch- oder vergleichbar, da sie sehr unterschiedliche Aspekte und Granularitäten der Anwendung betreffen.

Um die Anforderungen an die Integration herauszuarbeiten, sind beide Anteile des Grobentwurfs noch einmal in Abbildung 6.1 dargestellt. Dabei ist zu beachten, dass der linke Teil der Abbildung der Idee des Mulan-Plattformagenten (Reese, 2009; Cabac, 2010) entspricht und der rechte Teil der Plattform aus (Rölke, 2004).

Bei der Frage nach Anwendungsarchitekturen steht die Überlegung im Vordergrund, welche Aspekte und in welcher Detailtiefe die Plattform beschrieben werden soll. Die Mulan-Architektur beschreibt in erster Linie die Hierarchie zwischen Infrastruktur, Plattform und Agenten. Basierend auf der Einheitentheorie werden Operationen zu übergeordneten Einheiten, untergeordneten und gleichberechtigten Einheiten beschrieben. Daher sollte die Plattform nach einem Architekturstil entworfen werden, welcher diesen Eigenschaften Rechnung trägt. Dabei sei an dieser Stelle noch einmal erwähnt, dass die folgenden Architekturstile nicht in direkter Konkurrenz miteinander stehen, sondern ganz unterschiedliche Umfänge adressieren. Viele Architekturen entstammen (u.A.) dem Bereich der Webserviceentwicklung. Ziel dieser Betrachtung ist es, eine Architektur mit geeignetem Umfang für die Realisierung einer Plattformarchitektur zu finden.

Die hexagonale Architektur ist im Wesentlichen technisch motiviert. Im Zentrum der Anwendung steht (Geschäfts)logik und äußere, konkretere Schichten können auf innere, abstraktere Schichten zugreifen, jedoch nicht umgekehrt. Während diese Anwendungsform hilfreich für eine saubere Trennung der Zuständigkeiten

innerhalb der Anwendung ist, werden Interaktionen wenig beschrieben. Durch ihre Trennung von Spezifikation der (externen) Schnittstellen im Inneren und Implementation in äußeren Ebenen ist sie für den Einsatz im hier betrachteten Kontext nicht uninteressant. Da sie wie einleitend gesagt in erster Linie technisch motiviert ist, fällt es schwer auf der hier betrachteten abstrakten und konzeptuellen Ebene neue Gedankenanstöße aus ihr zu extrahieren. Die Trennung von Spezifikation und Implementation beschreibt die Essenz der modellbasierten Entwicklung, daher kann auch die hexagonale Architektur unterstützend bei konkreten Umsetzungen zum Einsatz kommen, wie sie ab Kapitel 9 und insbesondere in Kapitel 14 betrachtet werden.

Die Twelve-Factor-App ist als übergreifendes Rahmenwerk für Anwendung, Entwicklung und Deploymentlandschaft – oder in Analogie zur hier betrachteten Erweiterung von Mulan: Plattform und Plattformmanagement – konzipiert. Sie ist ebenfalls für technische Lösungen optimiert und macht teils technisch sehr spezifische Vorgaben, wie beispielsweise das Binden von Services an Ports. Während sie wie die hexagonale Architektur viele hilfreiche Gedankenanstöße bietet, insbesondere im Kontext der Entwicklung eines Webservice-Systems, passt der Umfang und die Abstraktionsebene der Vorschläge nicht umfassend auf den hier intendierten Einsatzzweck. Ein denkbarer Umgang wäre es jedoch, nur die entsprechend passenden Aspekte der Twelve-Factor-App einzusetzen und die verbleibenden Aspekte für die Realisierung und technische Details aufzusparen.

Cloud-Nativity kann als Weiterentwicklung der Twelve-Factor-App gesehen werden und adressiert in einem geringeren Rahmen, in erster Linie, aber nicht ausschließlich die Anwendung. Eine besondere Rolle spielen dabei Einsetzbarkeit ohne manuelle Eingriffe und Kommunikationsmöglichkeiten der Anwendung, aber auch die Stabilität des Gesamtsystems durch lokale Umsetzungen. Insgesamt passt der Umfang der Cloud-Nativity hervorragend für die Spezifikation der Plattform. Gleichzeitig macht der Ansatz der Cloud-Nativity ausreichend wenig Vorgaben, um nicht in Konflikt mit der komplexen Natur der Mulan-Architektur zu geraten.

Aus der Auswahl kann somit argumentiert werden, dass Cloud-Nativity die vielversprechendste Bereicherung für die Integration von Mulan-Plattform und Mulan-Agent darstellt und daher Gegenstand detaillierterer Untersuchungen sein sollte.

6.1.1. Cloud-Nativity und Mulan

Cloud-Nativity schlägt die Umsetzung der Konzepte der »Beobachtbarkeit«, »Operabilität«, »Agilität« und »Abhärtung« vor (Garrison und Nova, 2017), um damit das primäre Ziel erreichen zu können, dass die Anwendung bedien- und

nutzbar ist, ohne dass Administratoren dabei Zugriff auf das zugrunde liegende System zu benötigen. Der Hintergrund ist, dabei eine möglichst hohe Unabhängigkeit der zugrunde liegenden Infrastruktur zu erreichen, um den Einsatz in heterogenen und Cloud-Umgebungen zu ermöglichen. Ein weiteres Ziel umfasst die automatisierte Verwaltung solcher Anwendungen, da alle Aspekte direkt in der Anwendung über eine einheitliche Schnittstelle abgerufen oder konfiguriert werden können.

Auf diese Weise können einfach neue Plattformen erzeugt werden, ohne einen großen individuellen Aufwand für die Integration auf einer bestimmten Hardware zu betreiben. Zusätzlich kann durch individuelle und gekapselte Funktionalitäten eine entsprechende Heterogenität erreicht werden. Dies bezieht sich sowohl auf die Kompatibilität mit der darunterliegenden (heterogenen) Infrastruktur, in welchem Fall die Funktionalität der Rolle von Treibern in Betriebssystemen ähnelt, als auch auf das Angebot spezifischer Routinen und Algorithmen.

Bei genauerer Betrachtung fällt auf, dass diese Anforderungen recht gut auf die hier zuvor herausgearbeiteten Anforderungen an die Plattformarchitektur passen. Der Einsatz einer Cloud-nativen Anwendung ist vor allem im Kontext von Webservices vorteilhaft, da die Mulan-Architektur in dieser Hinsicht interpretiert werden kann, wie in (Moldt, Ortmann und Offermann, 2004) beschrieben wurde.

Aber auch die detaillierten Bestandteile der Architektur passen gut zu den Anforderungen, wie im Folgenden im Einzelnen erläutert wird. Das Anstreben einer Cloud-Native-Architektur der Plattform ermöglicht die Nutzung einzelner bestehender Softwarelösungen aus dem Webservice-Bereich. Darüber hinaus kann die hier zu spezifizierende Architektur auf eine etablierte Architektur abgebildet werden und (neben der Mulan-Architektur im großen) aus ihr hergeleitet werden.

Im Folgenden sollen die bereits eingeführten Aspekte von Cloud-Nativity im Kontext der angestrebten Erweiterung von Mulan interpretiert werden:

Beobachtbarkeit fordert, dass die Anwendung umfassende Auskünfte über ihren internen Zustand machen kann. Als Plattform umfasst dies insbesondere die individuellen Funktionalitäten der Plattform und etwaige Auskünfte über Agenten auf der Plattform.

Operabilität fordert, dass die Anwendung zur Laufzeit direkt administriert werden kann. Der Plattform kann so zur Laufzeit zusätzliche Funktionalität zur Verfügung gestellt werden. Ebenso können neue Agentendefinitionen eingespielt werden. Auch das Starten bzw. Aktivieren neuer Agenten kann so realisiert werden (reaktive Erzeugung von Agenten).

Agilität erfordert die flexible Einsetzbarkeit und Entwicklung der Anwendung. Diese Anforderung ermöglicht insbesondere den Einsatz der Plattform in verschiedenen (heterogenen) Umgebungen.

Abhärtung fordert, dass die Anwendung robust und gegen kaskadierende Fehlschläge geschützt ist. Während dies in keine der gestellten Anforderungen direkt passt, ist es stets eine wünschenswerte Eigenschaft einzelner Anwendungen in verteilten Systemen.

6.1.2. Beobachtbarkeit

Als erster betrachteter Punkt soll die *Beobachtbarkeit* adressiert werden.

Beobachtbarkeit meint in diesem Sinne die Fähigkeit der Anwendung, Informationen über ihren internen Zustand und das System, auf dem sie ausgeführt wird, mitteilen zu können. Dabei sollte das verwendete Kommunikationsprotokoll[1] möglichst universell sein, damit die Anforderung, um es einzusetzen, von möglichst vielen Kommunikationspartnern erfüllt werden kann. So kann ein flexibler Konsum der bereitgestellten Informationen über Technologiegrenzen hinweg gewährleistet werden. Darüber hinaus wäre es für einfache Statusinformationen hilfreich, wenn das Kommunikationsprotokoll nicht zustandsbehaftet ist, sodass es nicht nur nach bestimmter Vorbereitung und Bekanntheit der beiden Kommunikationspartner funktioniert. Dies führt dazu, dass Kommunikationspartner wesentlich dynamischer ausgetauscht werden können.

Während zur Beobachtbarkeit auch eine Information über die Simulation selbst zählt, wird dies nicht direkt von der intendierten Architektur gefordert. Im Idealfall sollte die Simulation jedoch über externe Systeme durch Nutzer nachvollzogen werden können. Dafür wäre die Bereitstellung eines übergreifenden Simulationsfeeds hilfreich: Das System soll Beobachtungen erlauben, die nicht auf individuelle physikalische Systembestandteile beschränkt sind. Dies wäre streng genommen eine Anforderung an das Plattformmanagement, sie muss jedoch auch lokal innerhalb der Plattform beachtet werden.

Die Möglichkeit des persistenten Speicherns des Simulationszustandes könnte eingesetzt werden, um Auskunft über den Zustand einer Simulation zu erteilen. Der Ansatz der Persistierung des genauen Simulationsablaufes ist innerhalb der True-Concurrency-Semantik äußerst komplex, da keine Garantien zur Konsistenz der Zustände (bzw. der Markierung in Petrinetz-Terminologie) zu einem bestimmten Zeitpunkt bestehen. Eine Möglichkeit wäre dabei der Einsatz von Checkpoints.

[1] In Abgrenzung zum Protokoll aus der MULAN-Architektur wird hier das Wort »Kommunikationsprotokoll« verwendet. Beispiele für konkrete Kommunikationsprotokolle sind HTTP, Java RMI, etc.

Umfassende Überlegungen hierzu werden später in Abschnitt 11.6 noch einmal ausgeführt.

Jenseits der Bereitstellung von Informationen aus der Simulation selbst heraus kann es hilfreich sein, die physikalische Auslastung des zugrunde liegenden Systems, wie beispielsweise Prozessorauslastung oder Speicherbelegung zugreifbar zu machen. Da die grundlegende Annahme bei Cloud-Nativity davon ausgeht, dass kein direkter Zugriff auf das System selbst besteht, liegt es nahe, diese Funktionalität ebenfalls von innerhalb der Plattform bereitzustellen: Systembestandteile sollen Informationen über den Zustand des physikalischen Systems (Speicherauslastung, Prozessorauslastung, etc.) durch die Anwendung bereitstellen.

Als letzter Punkt der Beobachtbarkeit kann auch die Bereitstellung von sog. »Health-Metriken« genannt werden. Dabei handelt es sich um anwendungsspezifische Laufzeitinformationen, durch die ein Rückschluss auf die korrekte Funktionsweise der Plattform möglich ist. Da diese Beurteilung von innerhalb der Anwendung sehr viel einfacher zu bewerkstelligen ist als durch einen externen Beobachter, sollte diese Art Informationen auch in detaillierter Fassung von der Anwendung bereitgestellt werden.

6.1.3. Operabilität

Die Operabilität beschriebt alle Aspekte, welche die Interaktion mit der Plattform und der Simulation vonseiten eines Kommunikationspartners betreffen. Ein Kommunikationspartner kann dabei ein klassischer menschlicher Nutzer eines Systems sein, jedoch ist auch vorstellbar, dass ein anderes System (hier: Eine andere Plattform) die Rolle des Kommunikationspartners einnimmt. Somit kann beispielsweise ein Teil der Simulation prüfen wollen, ob eine bestimmte Plattform in der Lage ist, an einer aktuell laufenden Simulation teilzunehmen. Dagegen sprechen könnte beispielsweise das Fehlen bestimmter Funktionalitäten der Plattform.

Die Operabilitätsaspekte sollen jedoch die Synchronisation auf Simulationsebene über Rechnergrenzen hinweg in der verteilten Simulation selbst explizit nicht adressieren, da diese auf der Agentenebene separat in Kapitel 5 behandelt wurde. Die Operabilitätsaspekte beziehen sich im Kern auf die dort genannte einfache Form der Kommunikation.

Analog zur Beobachtbarkeit sollte die Steuerung der Anwendung über eine ähnlich allgemeine Schnittstelle möglich sein. Auf diese Weise kann garantiert werden, dass sowohl anwendungsseitig als auch durch Nutzerinterfaces Aktionen ausgeführt werden können. Der Einfachheit halber sollte garantiert sein, dass die Schnittstelle nicht nur ähnlich ist, sondern auch das gleiche Kommunikationsprotokoll einsetzt wie die der Beobachtbarkeit. Somit kann auf einfache Weise

die gebündelte Funktionalität von Beobachtbarkeit und Steuerung an einer Stelle integriert werden.

Da kein Zugriff auf das zugrunde liegende System selbst angenommen wird, sollte die Plattform auch grundlegende Möglichkeiten zur Steuerung der (lokalen) Simulation bereitstellen. Mögliche Befehle können dabei das Starten neuer (lokaler) Simulationen, das Anhalten laufender Simulationen sowie das schrittweise Weiterschalten von Simulation sein. Dabei handelt es sich aus Sicht der neuen Architektur um die Funktionalität von Plattformen, reaktiv Agenten zu aktivieren.

In einer dynamischen Umgebung kann nicht davon ausgegangen werden, dass sämtliche Agentendefinitionen, welche für alle Simulationen notwendig sind, bereits beim Ausrollen der Anwendung verfügbar sind. Daher sollte die Anwendung eine Schnittstelle bereitstellen, mit der weitere Agentendefinitionen zur Laufzeit in die Plattform geladen werden können. Auf diese Weise können Plattformen und damit Simulationen einfach und effizient erweitert werden.

Analog dazu kann es notwendig sein, dass für bestimmte Agentendefinitionen weitere plattformspezifische Funktionalitäten erforderlich sind. Dies kann insbesondere dann problematisch bzw. unglücklich sein, wenn softwareseitige Unterstützung zwar generell verfügbar ist, jedoch auf der speziellen Plattform fehlt. Die Plattform soll daher ebenfalls dynamisch durch weitere Funktionalität erweitert werden können.

6.1.4. Agilität

Die Umsetzung von *Agilität* innerhalb der Architektur der Plattform zielt insbesondere auf die Mobilität der Plattform und die Lauffähigkeit in heterogenen Umgebungen ab. Analog zur Mobilität von Mulan-Agenten in der Mulan-Architektur soll die Mobilität von Plattformen ebenfalls durch Erzeugung und Zerstörung auf Ebene des Plattformmanagements umgesetzt werden.

Mobilität bedeutet daher in diesem Fall das Hochfahren und Herunterfahren ohne manuellen Eingriff durch Nutzer. Aus diesem Grund ist es erforderlich, eine hohe Portabilität vonseiten der Anwendung zu gewährleisten. Wie zuvor bereits im Abschnitt 2.8.4 angedeutet, ist es außerordentlich schwierig, einen Referenznetzsimulator effizient zu betreiben, ohne dass dieser seinen Zustand (lokal) speichert. Diese Überlegung wird noch einmal detailliert am Beispiel des Simulators Renew in Abschnitt 10.2.5 aufgegriffen werden.

Der Einsatz einer zustandslosen Anwendung würde jedoch das permanente Speichern in persistenten Datenbanken erfordern, sodass jeder Feuervorgang durch den Wartevorgang auf die Datenbank stark verlängert werden würde. Eine Möglichkeit wäre es, wie im Abschnitt zur Beobachtbarkeit erläutert, ein Checkpoint-

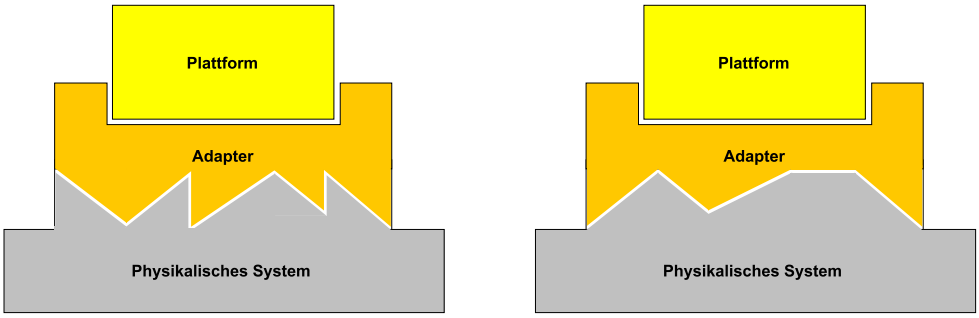

Abbildung 6.2.: Unterstützung von Heterogenität erfordert Abstraktion zum physikalischen System. Dies kann durch geeignete Adapter erfolgen.

basiertes Verfahren einzusetzen und auszulösen, sobald eine Situation mit Nichtdeterminismus aufgetreten ist. Allerdings würde auch ein solches Checkpoint-basiertes Verfahren im Allgemeinen zu erheblichen Verzögerungen führen, je nach simuliertem Netz unter Umständen sogar mehr als eine Persistierung in jede Transitionsfeuerung einzubetten. Zusammengefasst kann daher leider nicht davon ausgegangen werden, dass die Plattform die Anforderung der Zustandslosigkeit im allgemeinen Fall erfüllen können wird.

Insofern muss der Begriff der Portabilität insoweit aufgeweicht werden, dass an dieser Stelle mit dem Begriff die Äquivalenz der Plattform bezüglich ihrer Funktionalitäten gemeint ist, jedoch nicht der aktuell dort ansässigen Agenten. Nichtsdestotrotz wäre es denkbar, zunächst die Agenten individuell auf eine Übergangsplattform umziehen zu lassen und danach die gesamte Plattform zu bewegen. Dabei kann ein Verfahren, wie es in (Duvigneau, 2002) im Rahmen der Capa Plattform eingesetzt wurde, verwendet werden. Dort wurden ausstehende Nachrichteneingänge, ausstehender Nachrichtenversand sowie aktive Protokolle der Agenten überwacht, um die vorhandene Aktivität des Agenten zu definieren. Nicht aktive Agenten (welche bei allen drei genannten Punkten keine Einträge vorweisen konnten) waren in der Lage ihre Koordinaten bzw. Adresse auf der Plattform zu ändern. Eine Erweiterung des Prinzips auf mehrere Plattformen sollte aus Sicht der Agenten problemlos möglich sein und reduziert sich auf allgemeine Fragen der Adressierung zwischen Plattformen. Die Adressierung zwischen Plattformen wird noch einmal in Abschnitt 6.1.5 zur Abhärtung der Anwendung aufgegriffen.

Zuvor wurde ebenfalls die Lauffähigkeit in heterogenen Umgebungen angesprochen. Bei der Mobilität von Plattformen spielt dies eine wichtige Rolle, da die Einschränkung auf komplett gleichartige Systeme problematisch sein kann. Umgang mit Heterogenität kann beispielsweise durch den Einsatz von Adaptern realisiert werden. Abbildung 6.2 zeigt dies bildlich.

Da sich der Einsatz von Adaptern jedoch nicht auf die einzelne Plattform beschränkt, muss auch das Plattformmanagement entsprechend den Umgang mit den Adapter-Layern beherrschen. Daher kann sich aus dem Kontext der Realisierung der Plattform eine weitere Anforderung an das Plattformmanagement formulieren lassen: Das Plattformmanagement muss die Umgebung der Plattform in geeigneter Weise vorbereiten, damit eine einheitliche Plattformimplementation mit heterogenen Umgebungen umgehen kann. Aus Plattformsicht ergibt sich die zugehörige Anforderung: Die Plattform soll in einem portablen Deploymentformat ausgeliefert werden, welches schnelle Umzüge auf neue Infrastruktur erlaubt.

Neben der Heterogenität der Systeme, auf denen die Plattform ausgeführt werden soll, kann die Heterogenität auf einen weiteren Aspekt bezogen werden: Mehrere Plattformen müssen nicht identisch aufgebaut sein, während eine Kommunikation zwischen den Plattformen immer noch möglich sein soll. Das Gleiche gilt für die Agenten, welche sich auf der Plattform befinden.

Mit Capa existiert eine Implementation für die Mulan-Plattform, welche diese Unterstützung für Heterogenität durch ein gemeinsames FIPA-basiertes Kommunikationsprotokoll ermöglicht. Mit den bisher herausgearbeiteten Anforderungen an die Plattform entstehen jedoch einige Schwierigkeiten, Capa zusätzlich direkt einzubinden. Außerdem beschränkt sich die Unterstützung auf FIPA-konforme Agenten, was als Einschränkung an die Architektur bisher noch an keiner Stelle so formuliert wurde. Spätestens bei der Integration von Protokoll und Agent im nächsten Unterabschnitt wird auf Synchronisation als Mittel des Informationsaustausches zurückgegriffen werden müssen. Folglich sollte eine Plattform mit ihrer Kommunikationsmechanik zu anderen Plattformen auf ein unabhängiges und allgemein verfügbares Kommunikationsprotokoll setzen.

Wird dies ebenfalls für die Synchronisation auf Agentenebene eingesetzt, wird es möglich – sofern der Rahmen von synchronen Kanälen als Kommunikationsprotokoll eingehalten wird – Plattformen mit gänzlich anderen Formalismen auszustatten und sie trotzdem in eine einzelne (verteilte) Simulation zu integrieren. Eine interessante Ergänzung sind hierbei die Curry Colored Petri Nets (CCPN) aus (M. Simon, Moldt u. a., 2019), welche funktionale Konstrukte als Anschriftensprache verwenden und damit seiteneffektfrei sind.

Curry Colored Petri Nets sind außerdem aus einem gänzlich anderen Grund an dieser Stelle von Interesse und zwar dem der Eigenschaft der Zustandslosigkeit. Ideal für die Agilität einer Plattform wäre, wie einleitend erläutert, wenn diese zustandslos wäre. Aus erläuterten Gründen ist dies jedoch für Referenznetze nicht einfach umsetzbar. Anders verhält es sich indessen mit Curry Colored Petri Nets, bei denen Transitionen und ihre Inschriften Funktionen sind. Zustandslose Agentenplattformen sind dort Teil aktueller Untersuchungen und werden daher an dieser Stelle nicht weiter behandelt. Dementsprechend verbleiben diese für weitere Untersuchungen jenseits dieser Arbeit.

Neben den genannten Eigenschaften bezüglich der Agilität des Systems, welche sich aus der Anforderung an die Architektur ergeben, existieren weitere Agilitätseigenschaften, die ebenfalls hilfreich sein können. Agilität adressiert im Architekturstil der Cloud-Nativity zusätzlich auch die Entwicklungs- und Anpassungsgeschwindigkeit der Software. Der Einsatz von Continuous Integration wird nahegelegt sowie der Einsatz agiler Entwicklungsmethoden. Folglich sollten diese Eigenschaften auch als Anforderungen an etwaige Realisierungen beachtet werden: Bei der Entwicklung sollte eine Continuous Integration Pipeline zum Einsatz kommen. Die Entwicklung einer Plattform sollte nach agilen Prinzipien erfolgen, um schnell auf geänderte Anforderungen reagieren zu können. Beides betrifft die Anwendung indirekt, indem schnelles Feedback an die Entwickler getragen werden kann und gleichzeitig der personelle Aufbau des Entwicklerteams in die Struktur der Anwendung übergeht. Bei der letzten Eigenschaft handelt es sich wie in Abschnitt 2.4.3 bereits im Paose-Kontext beschrieben um das Gesetz von Conway (Conway, 1968).

6.1.5. Abhärtung

Die Abhärtung im klassischen Sinne der Cloud-Nativity umfasst im Wesentlichen die Vermeidung von kaskadierenden Fehlschlägen und die Vermeidung von Abhängigkeiten von der Integrität von Daten, welche durch andere Services (bzw. Plattformen) im Gesamtsystem geliefert werden.

Für die Interaktion zwischen Plattformen kommen Nachrichten zum Einsatz und somit asynchrone Kommunikation. Während diese Eigenschaft grundlegend dazu führt, dass keine direkte Abhängigkeit zu anderen Services besteht, könnte es dennoch vorkommen, dass Algorithmen indefinit auf Antworten warten und andere Ausführungen blockieren.

Eine weitere Anforderung bezüglich der Abhärtung wurde bereits im Abschnitt zu Agilität motiviert: die Adressierung sowohl innerhalb von als auch zwischen Plattformen. Dabei ist es wichtig, dass zwischen den drei im Wesentlichen betrachteten Hierarchiestufen unterschieden wird: Es sollten also eindeutige Identifikatoren für das Plattformmanagement, für jede Plattform und für jeden Agenten existieren. Das Problem der Adressierung in verteilten Systemen ist im Wesentlichen als gelöst zu betrachten, die Umsetzung im Kontext von Mulan und dem hier vorgestellten Aufbau ist aber nicht trivial. Eine mögliche Umsetzung ist die lokale Generierung von Adressen mit möglichst geringer Kollisionschance wie beispielsweise UUIDs.

Wird die architekturelle Sicht auf die Software verlassen und eine technische Sicht eingenommen, setzen sich die Anforderungen naturgemäß fort. Während der Fokus der Arbeit auf der architekturellen Sicht liegt, sollten technische Realisie-

rungen nicht erneute Möglichkeiten zu kaskadierenden Fehlschlägen einführen. Dabei ist es jedoch schwierig ohne den technischen Kontext einer Implementation konkrete Probleme zu adressieren. Aus diesem Grund soll die Diskussion zur Abhärtung bezüglich technischer Komponenten in späteren Kapiteln der Arbeit erfolgen.

6.2. Funktionsumfang der Plattform

Im Folgenden werden noch einmal kurz die zentralen Anforderungen aus der konzeptuellen Sicht an eine Plattform rekapituliert. Die Inhalte orientieren sich im Wesentlichen an der in Abschnitt 4.4.2 vorgestellten Übersicht.

Die zentrale Aufgabe von Plattformen ist es, Agenten zu beheimaten, sie nach bestimmten Gesichtspunkten zu erzeugen und auch wieder von der Plattform zu entfernen. Um dieses Ziel erfüllen zu können, muss eine Plattform eine Reihe an Rahmenbedingungen schaffen. Zu den einfachen Anforderungen zählt die Ermöglichung von Kommunikation zwischen Agenten der lokalen Plattform und anderen Plattformen und die Lokalisierung von Agenten. Ferner sollte die Plattform über Definitionen bzw. Blaupausen verfügen, mit denen neue Agenten erschaffen werden können. Eine weitere Aufgabe besteht darin, den Umzug eines Agenten zu einer Plattform vorzubereiten bzw. einen umziehenden Agenten aufzunehmen.

Darüber hinaus sollte sie in der Lage sein, eine gewisse Menge an Auskünften zu ihrem Zustand machen zu können. Dies beinhaltet bezogen auf die Plattform globale Informationen, aber auch beispielsweise Informationen zu ansässigen Agenten.

Hierarchische Systeme für Agenten und insbesondere solche, bei denen die übergeordneten Einheiten ihrerseits wieder als Agenten interpretiert werden können, wurden seit Ende der 1990er Jahre unter dem Begriff *holonische* Multiagentensysteme (J. Christensen, 1994; Fischer, 1999) erforscht. Die Grundidee umfasst dabei, dass Agenten ihrerseits aus Subagenten konstruiert sind, welche jedoch ihre innere Autonomie nicht aufgeben. Der übergeordnete Agent kann nun Dienste anbieten, die durch die Subagenten kooperativ ausgeführt werden. Das Konzept wurde im Kern bereits durch Mulan adressiert und sollte auch mit der Erweiterung auf ein Plattformmanagement nicht verloren gehen. Aus diesem Grund muss ein Plattformmanagement in der Lage sein, gewisse Informationen von seinen Plattformen abzurufen bzw. geliefert zu bekommen. Daher sollte auf Ebene der Plattform ein geeigneter Feed der Geschehnisse der Agenten bereitgestellt werden. Das Plattformmanagement kann dann eine Aussage über die Gesamtheit der Plattformen tätigen. Technisch kann dies durch einen Simulationsfeed gelöst

werden. Diese Idee wird in Kapitel 11 zur Realisierung der Plattform erneut aufgegriffen werden.

Zuletzt bleiben noch die Anforderungen aus der Agentenkommunikation an die Plattform. Die Realisierung komplexer Kommunikation adressiert in erster Linie die Kommunikation zwischen Plattformen. Daher soll für die Umsetzung der Plattform eine einfache Nachrichtenweiterleitung an den entsprechenden Agenten genügen. Dennoch muss die Anforderung für komplexe Kommunikation bzw. Konversation an das Plattformmanagement weitergetragen werden.

6.2.1. Start und Stop von Agenten

Eine der zentralen Funktionalitäten der Agentenplattform umfasst das Starten und Stoppen von Agenten. In einfachen Abstraktionen genügt eine Modellierung, wie sie die Mulan-Plattform beschreibt. Außer Acht gelassen wird dabei jedoch die genaue Konstruktion von Agenten, bzw. auf welcher Basis sie erzeugt werden. Wird ein Agenten-Standardmodell angenommen, ist dies weniger problematisch. Unterscheiden sich die Agenten jedoch beispielsweise durch den Aufbau der Protokollfabrik oder ist auch ein anderer als ein klassischer Mulan-Agent denkbar, so ist das Modell nicht mehr ausreichend.

Zu diesem Zwecke soll eine *Agentenfabrik* ähnlich der Protokollfabrik eines Mulan-Agenten zum Einsatz kommen, welche die Konstruktion von Agenten übernimmt. Dabei ist die Verwendung der Agentenfabrik abhängig von den Umständen, in denen sie verwendet wird. Somit kann die Erzeugung proaktiv geschehen, angestoßen von der Plattform selbst, oder aber reaktiv, wenn eine entsprechende Nachricht bei der Plattform eingeht. Die Erzeugung ist in beiden Fällen gleich und unterscheidet sich nur durch die Initiation. Nachdem die Erzeugung eines neuen Agenten beschlossen wurde und innerhalb der Agentenfabrik das Vorhandensein der Definition des gewünschten Agentenmodells bestätigt wurde, kann die Agentenfabrik damit fortfahren, den Agenten zu instantiieren.

Die Entfernung bzw. Zerstörung eines Agenten kann nur durch die Plattform angestoßen werden. Dabei sollte, ähnliche wie in (Duvigneau, 2002) beschrieben, darauf geachtet werden, dass etwaige Agenten zum Zeitpunkt des Herunterfahrens nach Möglichkeit inaktiv sind. Eine andere Möglichkeit der Entfernung stellt der Umzug eines Agenten dar. Dieser Fall wird noch einmal in Abschnitt 6.2.3 beschrieben.

6.2.2. Erweitern von Agentendefinitionen

Im letzten Abschnitt wurde die Agentenfabrik vorgestellt. Eine wesentliche Aufgabe der Agentenfabrik besteht darin, eine Reihe von Agentendefinitionen vorzuhalten. Auf der Basis dieser Definitionen können Agenten durch die Plattform instantiiert werden. Nun ist es denkbar, dass im Gesamtsystem in einer späteren Situation Agentendefinitionen nachgeliefert werden müssen. Dies kann beispielsweise in Reaktion auf geänderte Umgebungsbedingungen erfolgen, oder aber auch durch Agentenumzüge. Aus diesem Grund sollte es der Plattform möglich sein, spezialisierte Nachrichten zu empfangen, welche die Agentendefinitionen der Agentenfabrik erweitern.

In Implementationen ist an dieser Stelle jedoch ein genaues Augenmerk auf den Aspekt der Sicherheit zu legen. Somit sollte die Plattform nur Definitionen aus vertrauenswürdigen Quellen akzeptieren. Da aber Sicherheitsbetrachtungen nicht im Umfang der Arbeit liegen, sei auf diesen Umstand lediglich hingewiesen.

6.2.3. Umzug von Agenten

Wie zuvor bereits motiviert, sollte es im Verbund mehrerer Plattformen für Agenten möglich sein, auf andere kompatible Plattformen umzuziehen. Der Umzug von Agenten wurde auch bereits im Kontext von Mulan im Rahmen von Mapa (Cabac, Moldt, Wester-Ebbinghaus u. a., 2009) und dem Mobilitätsgedanken untersucht. Die generellen Rahmenbedingungen ergeben sich für den Umzug wie folgt: Ein Umzug kann nicht real erfolgen, sondern sollte als Dematerialisierung und Materialisierung behandelt werden. Im Kontext des hier betrachteten Agentensystems bedeutet dies ein Herunterfahren bzw. Zerstören des Agenten auf der alten Heimatplattform und ein darauf folgendes Instantiieren auf der neuen Plattform. Dabei muss darauf geachtet werden, dass auf der neuen Plattform die Agentendefinition des Agenten existiert und dass seine gegenwärtige Wissensbasis übertragen wird.

Die Funktionalität des Umzugs umfasst dabei im weitesten Sinne eine Aneinanderreihung der bereits definierten Funktionalität. Die Erzeugung auf der neuen Plattform kann durch das reaktive Starten des Agenten modelliert werden, nachdem mittels Erweiterung der Agentendefinitionen sichergestellt wurde, dass die neue Plattform in der Lage ist, den Agenten zu erzeugen. Die alte Wissensbasis kann durch die alte Plattform durch eine einfache Nachricht an den Agenten auf der neuen Plattform übermittelt werden.

Eine Operation fehlt an dieser Stelle jedoch im Modell der Plattform: die Zerstörung des Agenten mit Absicht, ihn auf eine andere Plattform umzuziehen. Dies sollte vom Agenten durch eine Nachricht an die Plattform initiiert werden können,

welche sodann seine Wissensbasis entgegennimmt, ihn zerstört und sodann den Transfer der Agentendefinition, den entfernten Start und die Übertragung der Wissensbasis veranlasst. Der Umzug sollte daher strukturell auf der Plattform als spezialisierter Entfernungsprozess (mit ausgehender Nachricht) von Agenten behandelt und modelliert werden.

6.2.4. Angebot spezialisierter Funktionalitäten der Plattform

In den Anforderungen wurde die Heterogenität von Plattformen adressiert. Neben der reinen Lauffähigkeit von Plattformen auf verschiedenen Umgebungen, sollten Möglichkeiten für Agenten bestehen, Vorteile aus den Gegebenheiten von Umgebungen ziehen zu können. Da eine Skalierbarkeit des Gesamtsystems bzw. die dynamische Erzeugung von Plattformen ein vorgelagertes spezialisiertes Design jeder Plattform für eine spezifische Umgebung unmöglich macht, sollten derartige Funktionalitäten in einer dynamischen Weise verfügbar gemacht werden können.

Funktionalitäten adressieren in diesem Fall Interoperationsmöglichkeiten mit der Umgebung einer konkreten Plattform, können aber auch allgemeine spezialisierte Routinen zur Benutzung bereitstellen. Funktionalitäten verhalten sich im Gegensatz zu Agenten nicht aktiv und nicht autonom. Sie sind im Wesentlichen als Funktions- bzw. Protokollsammlungen zu verstehen, auf die Agenten zurückgreifen können, und sollten statisch und als Ressourcen verstanden werden.

Um eine dynamische Verwaltung der Funktionalitäten zu ermöglichen, sollten sie analog zu Agentendefinitionen nachträglich bereitgestellt werden können. Dies soll im Wesentlichen durch eine entsprechende Nachricht an die Plattform erfolgen. Sie unterscheiden sich von Agentendefinitionen insofern, als sie nicht zur Instantiierung von Agenten verwendet, sondern von diesen direkt (und gemeinsam) genutzt werden können. Agentendefinitionen bilden die Basis der aktiven Elemente der Plattform (die Agenten), während Funktionalitäten die passive Struktur der Plattform und damit die Ausführungsumgebung der Agenten vorgeben.

Analog zur Bereitstellung von Agentendefinitionen sollte bei Implementationen auf eine entsprechende Sicherheit geachtet werden.

6.2.5. Zustandsinformationen

Die Anforderung bezüglich der Bereitstellung von Zustandsinformationen kann als eine spezielle Form der Nachrichtenbeantwortung interpretiert werden. Dabei wäre die Modellierung als einzelner spezialisierter Agent innerhalb der Plattform theoretisch denkbar. Dennoch sind die Zugriffe auf die Bestandteile der Plattform ausschließlich (aus)lesend im Gegensatz zur Natur der üblichen Zugriffe durch die

Agenten der Plattform. So werden Plattformfunktionalitäten normalerweise einzeln angefragt und die Ausführung angefordert, anstatt eine Liste aller Funktionalitäten anzufordern. Um diesen Unterschied abzubilden, sollte die Zustandsabfrage als eigenständiges Konstrukt ins Modell der Plattform aufgenommen werden.

Ferner ergibt sich der Zustand der gesamten Plattform neben globalen Eigenschaften der Plattform durch die Summe der Zustände ihrer Agenten. Eine zwangsläufige Abfrage der Agenten wäre denkbar, widerspräche aber dem agententechnischen Grundgedanken der Autonomie. Daher sollte das Teilen von Statusinformationen auf freiwilliger Basis erfolgen. Für diese Sammlung globaler und lokaler Zustände bietet sich die Modellierung eines Status an, der von der Plattform und den einzelnen Agenten aktuell gehalten werden kann.

Bei der Anfrage bezüglich dem Zustand der Plattform können dann Plattformfunktionalitäten und der gesammelte Zustand ausgelesen und eine entsprechende Antwort formuliert werden. Der Status der Plattform entspricht so in gewissem Maße der Wissensbasis und den laufenden Prozessen eines Agenten.

6.3. Entwurf der Plattformstruktur

Auf der Basis der erläuterten Funktionsumfänge der einzelnen Komponenten der Plattform können nun die Interaktionen und Abläufe innerhalb dieser beschrieben werden. Zu diesem Zweck werden in diesem Abschnitt abermals Agent Interaction Protocols (AIPs) eingesetzt werden. Sie dienen dabei dem Zweck abschließend ein entsprechendes Referenznetzmodell der Plattform zu entwickeln und zusätzlich eine Referenz für die zugehörigen Abläufe darzustellen.

6.3.1. Start und Zerstörung von Agenten

Der proaktive Start eines neuen Agenten ist in Abbildung 6.3 dargestellt. Die Plattform kann jederzeit die Entscheidung fällen, einen neuen Agenten zu starten. Dazu wählt sie aus den verfügbaren Agentendefinitionen der Agentenfabrik eine passende aus und übergibt sie der Agenteninstantiierung, welche ihrerseits wieder Teil der Agentenfabrik ist. Nach der Instantiierung übergibt die Agenteninstantiierung der Plattform einen lokalen Agentenidentifikator. Diesen konvertiert die Plattform in geeigneter Weise in einen globalen Identifikator und übergibt ihn dem Plattformmanagement zur Registrierung.

Die reaktive Erzeugung eines Agenten läuft grundlegend ähnlich zu der proaktiven Erzeugung ab und ist in Abbildung 6.4 dargestellt. Der wesentliche Unterschied besteht darin, dass vor der eigentlichen Erzeugung eine Prüfung auf das Vorhandensein der gewünschten Agentendefinition vorgenommen wird. Sollte diese

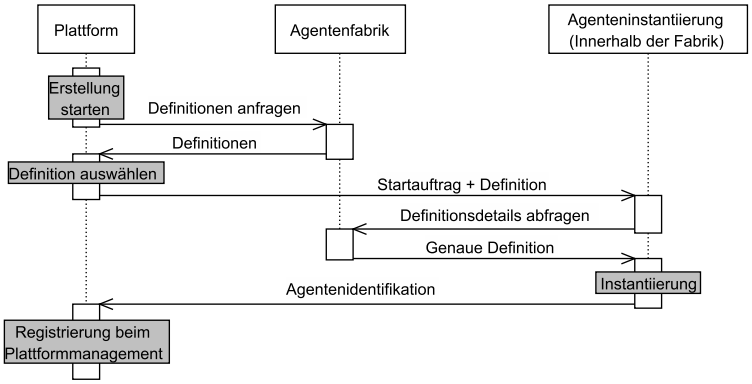

Abbildung 6.3.: Proaktiver Start eines neuen Agenten

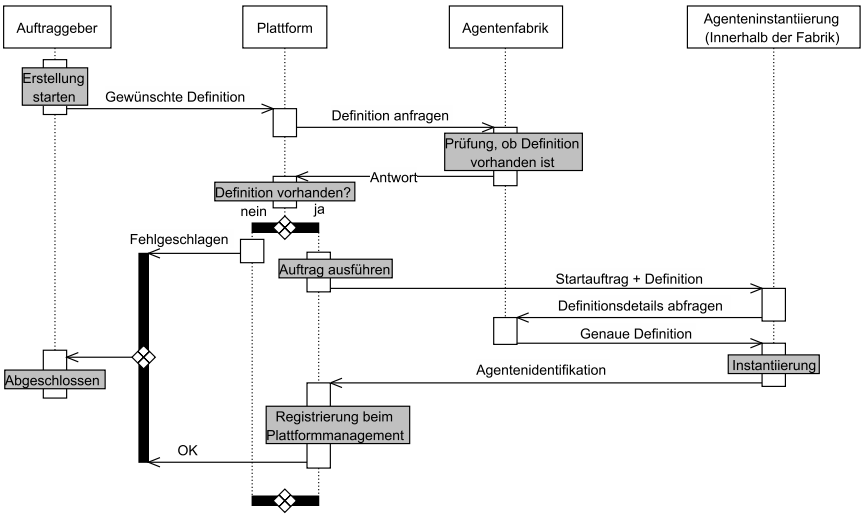

Abbildung 6.4.: Reaktiver Start eines neuen Agenten

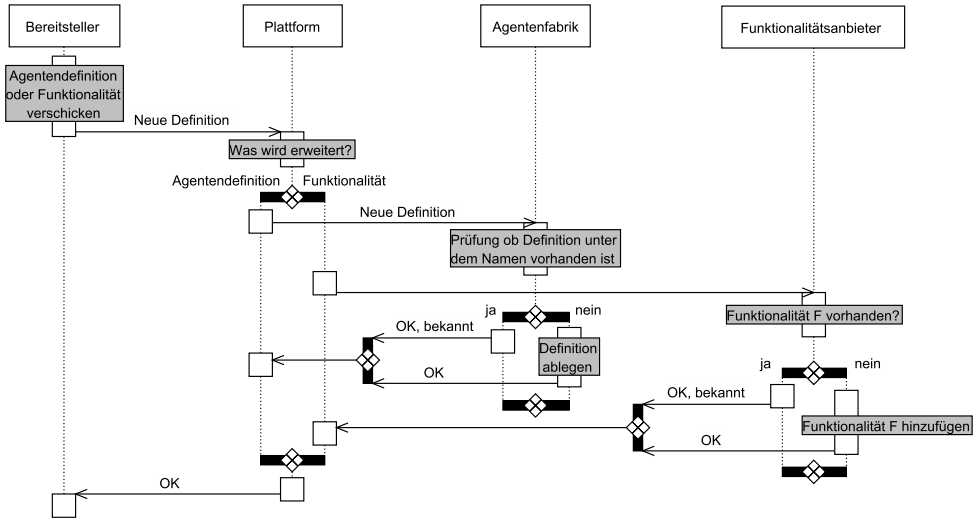

Abbildung 6.5.: Erweiterung der Agentendefinitionen oder Funktionalitäten einer Plattform

negativ ausfallen, so wird der Auftraggeber über den Fehlschlag informiert. Liegt die gewünschte Agentendefinition vor, so verfahren Agentenfabrik und Agenteninstantiierung wie im proaktiven Fall. Nach erfolgreicher Registrierung des neuen Agenten beim Plattformmanagement wird dem Auftragsteller der Erfolg gemeldet.

Die Zerstörung von Agenten umfasst im Wesentlichen nur die weitere Operation, den Agenten beim Plattformmanagement abzumelden. Der schematische Ablauf ist sehr einfach und daher nicht gesondert dargestellt.

6.3.2. Erweitern der Plattform

Auch der Prozess der Erweiterung der Agentendefinitionen ist im Kern wenig kompliziert. Über die Plattform schickt ein Bereitsteller eine neue Definition an die Agentenfabrik. Diese gleicht den Bezeichner der Definition mit den ihr bekannten Definitionen ab. Dem Bereitsteller wird in jedem Fall eine positive Nachricht zugestellt; abhängig davon, ob die Definition bereits vorhanden war, wird dieser Umstand auch entsprechend mitgeteilt. Der Ablauf ist grafisch in Abbildung 6.5 dargestellt.

Neben der Erweiterung von Agentendefinitionen zeigt die Abbildung ebenfalls die Erweiterung von Funktionalitäten. Der Prozess ist bezüglich der Erweiterung analog und betrifft mit dem Funktionalitätsspeicher nur einen anderen Anteil der Plattform. Funktionalitätsspeicher und Agentendefinitionen trotzdem als zwei

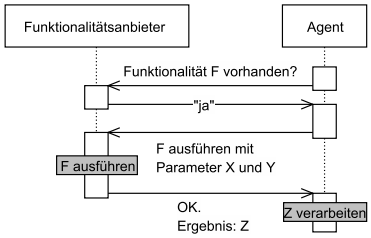

Abbildung 6.6.: Nutzung von Funktionalitäten einer Plattform durch Agenten

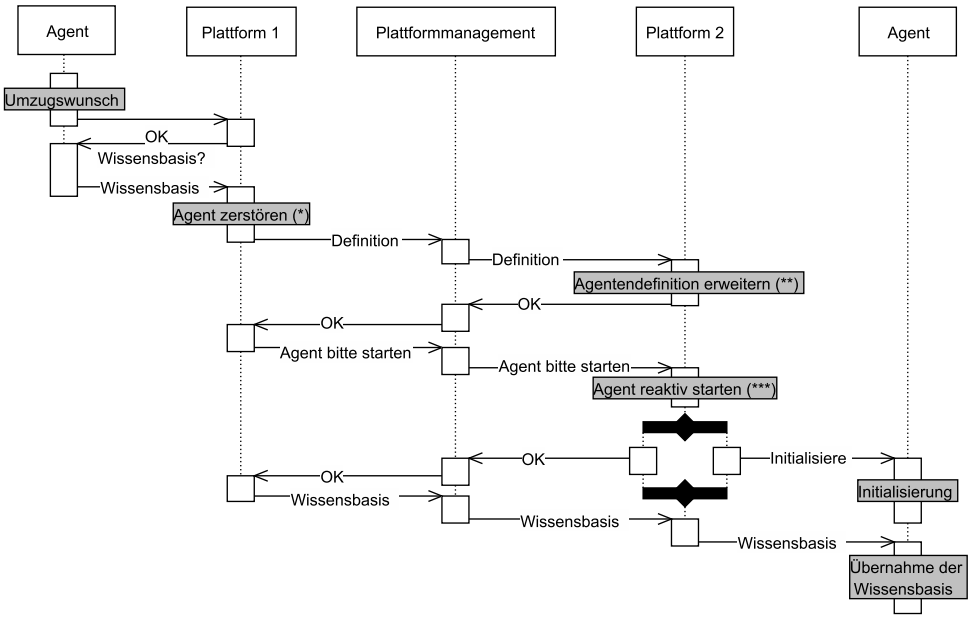

Abbildung 6.7.: Erfolgreicher Umzug eines Agenten zu einer anderen Plattform. (*) Siehe Ausführung in Abschnitt 6.3.1, (**) Siehe Abbildung 6.5, (***) Siehe Abbildung 6.4

separate Elemente zu modellieren, begründet sich darin, dass sie grundlegend anders verwendet werden. Zum Vergleich ist der Ablauf der Verwendung von Funktionalitäten in Abbildung 6.6 dargestellt.

6.3.3. Umzug von Agenten

Der Agentenumzug ist das komplexeste aller Protokolle im Bereich der Plattform. Glücklicherweise können die Hauptbestandteile wie bereits erwähnt durch die Protokolle der Agentenzerstörung, der Erweiterung von Agentendefinitionen und des

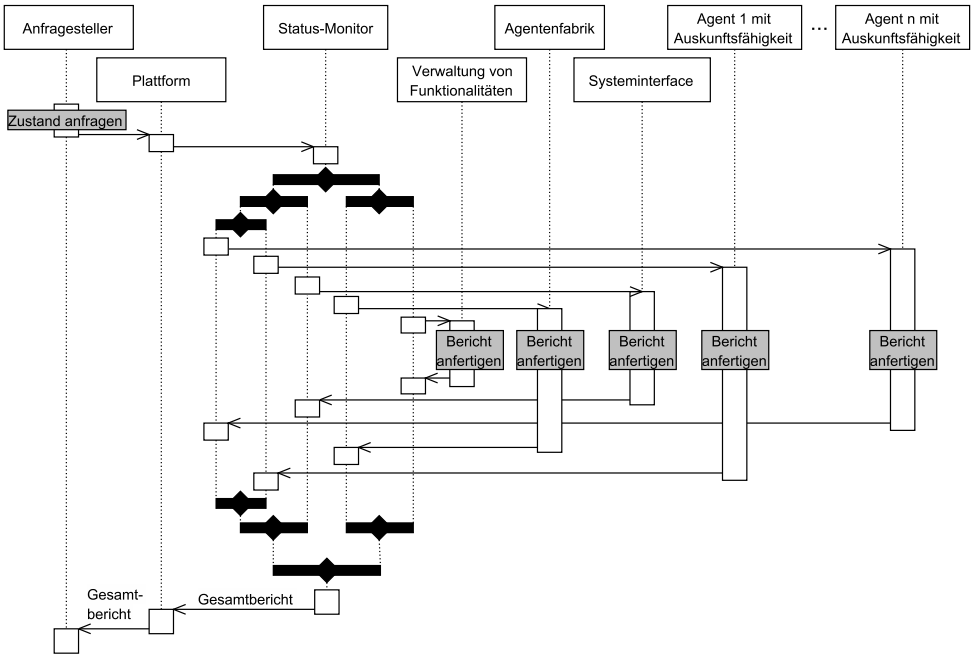

Abbildung 6.8.: Zustandsabfrage an einer Plattform

reaktiven Agentenstarts abgebildet werden. Die Übersicht des Agentenumzugs ist in Abbildung 6.7 dargestellt.

Zunächst bekundet der Agent der Plattform seinen Umzugswunsch. Diese fordert daraufhin einen Abzug der Wissensbasis des Agenten an, um diese nach dem Herunterfahren an die neue Inkarnation des Agenten übertragen zu können. Nach der Übertragung der Wissensbasis führt die Plattform das Protokoll zur Zerstörung des Agenten aus. Nachdem der Agent entfernt wurde, überträgt die Plattform die Agentendefinition an die neue Plattform und veranlasst danach den reaktiven Start des Agenten. Die zweite Plattform startet den Agenten und informiert die erste Plattform über den Start. Danach kann die Plattform die Wissensbasis an den neuen Agenten übertragen. Der Vorgang ist damit abgeschlossen.

6.3.4. Zustandsabfrage

Der letzte beschriebene Ablauf umfasst die Abfrage des aktuellen Zustands einer Plattform. Er ist in Abbildung 6.8 beschrieben. Die dabei gelieferten Informationen umfassen im Allgemeinen die von der Plattform angebotenen Funktionalitäten und Agentendefinitionen, die Befindlichkeit der Umgebung (das System) der

Plattform sowie Informationen von allen Agenten, welche zusätzliche Auskünfte geben wollen. Die Fähigkeit, Auskünfte zu geben, ist für Agenten optional.

Zunächst wird vom Antragsteller die Anfrage an die Plattform getragen. Danach wird an einen dedizierten Status-Monitor ein Auftrag übermittelt, welcher nebenläufig die entsprechenden Berichte bei Agentenfabrik, Funktionalitätsverwaltung, Systeminterface und bei allen Agenten mit Auskunftsfähigkeit anfragt. Sind alle Berichte eingegangen, wird ein Gesamtbericht angefertigt und der Plattform übermittelt, welche diesen sodann an den Anfragesteller übermittelt.

Auch die Bereitstellung eines Simulationsfeeds kann als spezialisierte Zustandsabfrage bezüglich der Zustandshistorie betrachtet werden. Die konzeptuelle Sicht ist hierbei weit weniger komplex als die Realisierung innerhalb von Referenznetzen. Daher soll dieser Aspekt an dieser Stelle abstrakt gehalten werden. Im späteren Abschnitt 11.6 wird eine mögliche Umsetzung in erhöhter Detailtiefe diskutiert.

6.4. Definition der Plattform

Abschließend kann auf Basis der Aufgaben der Komponenten und der Interaktionen der Komponenten miteinander eine Definition der Plattform erfolgen. Diese wird als Referenznetz dargestellt, um sie im Kontext der verbleibenden Elemente der Architektur und auch der Mulan-Architektur betrachten zu können.

6.4.1. Partiell schreibende Operationen

Bei der Betrachtung der Modellierungsmittel der Referenznetze und deren Implementation im Simulator Renew fällt auf, dass zwei Kantenarten für Operationen bereitstehen, welche eine Marke in einem Platz betreffen (konsumieren), und dass nach dem Feuern dort dieselbe Marke erneut zu finden ist. Dabei handelt es sich um Test- bzw. rückgerichtete Kanten. Testkanten tragen dabei die Semantik des lesenden Zugriffs. Die Marke wird nie vom Platz entfernt, andere Feuervorgänge können die Marke nebenläufig verwenden. Rückgerichtete Kanten hingegen stellen Schreiboperationen dar, bei denen die Marke beim Feuervorgang anderen Feuervorgängen nicht zur Verfügung steht und nach Abschluss der Operation einen veränderten Zustand aufweist.

Werden die Aktualisierung des Status durch Systemevents und der Zugriff darauf von Agenten oder aber die Aktualisierung von Plattformfunktionalitäten betrachtet, fällt dabei auf, dass die Operationen schreibend sind, jedoch nur einen sehr geringen Teil der Marke (bzw. des von ihr referenzierten Netzes) betreffen. Mit einer klassischen rückgerichteten Kante wäre aber die gesamte Marke gesperrt,

Abbildung 6.9.: Partielle Schreibkante.

was sich unnötig negativ auf die Performance einer entsprechenden Simulation auswirken kann.

Aus diesem Grund soll an dieser Stelle ein dritter Kantentyp für die Modellierung vorgeschlagen werden. Es handelt sich dabei um die partielle Schreibkante, sie wird dargestellt als gepunktete rückgerichtete Kante, um den schreibenden Anteil der Operation zu signalisieren, wie sie in Abbildung 6.9 dargestellt ist. Falls andere Feuervorgänge auf andere Bestandteile der Marke zugreifen wollen, so ist dies durch Testkanten und auch durch weitere partielle Schreibkanten möglich. Dabei orientiert sich die konkrete Umsetzung stark an der Struktur der Marke selbst und ist schwer im allgemeinen Fall zu definieren. Aus diesem Grund wird die Kante auch als Modellierungswerkzeug vorgestellt und nicht etwa als Implementation beispielsweise im Simulator Renew. Die Umsetzung in den späteren Realisierungen und Prototypen wird in den zugehörigen Kapiteln 10 bis 14 diskutiert.

Eine möglicher konkreter Einsatz im Bereich von Datenbanken wäre beispielsweise die Modellierung von Schreibsperren auf Datensatzebene anstatt der kompletten Schreibsperre auf die Tabelle (was der rückgerichteten Kante entsprechen würde).

6.4.2. Darstellung als Referenznetz

Abbildung 6.10 zeigt die Darstellung der Plattform als Referenznetz mit partiellen Schreibkanten. Orangene Transitionen symbolisieren Kommunikationen zum Plattformmanagement oder aber zu Agenten untereinander. Hellgelbe Plätze und Transitionen zeigen generelle Abläufe und Zustandsdetails der Plattform an. Gelbe Plätze hingegen zeigen Details und Operationen einzelner Agenten der Plattform an, sofern sie keinem übergeordneten Ablauf zugerechnet werden können (wie beispielsweise der Zustandsabfrage). Die anderen Farben tragen detailliertere Bedeutungen und werden im Folgenden kurz erläutert.

Die einzelnen Bestandteile modellieren die strukturellen Komponenten der Plattform, welche entsprechend der vorgestellten Ablaufprotokolle verbunden sind. Die Abbildung zeigt dabei die Integration aller Einzelkomponenten in ein komplexes Plattformmodell.

Im oberen Bereich können Zustandsanfragen durch einen Status-Monitor bearbeitet werden, welcher durch die Plattform erzeugt wird und in grau dargestellt ist.

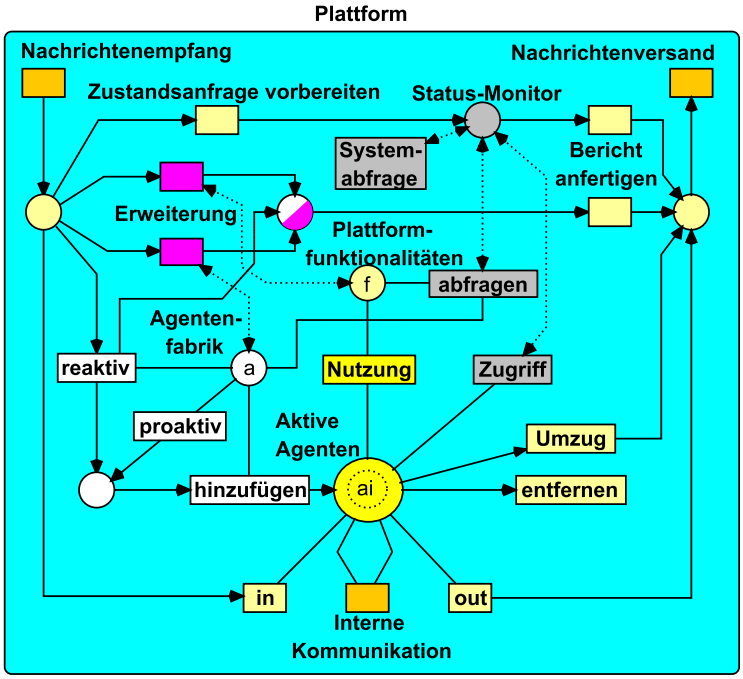

Abbildung 6.10.: Gesamtentwurf der Plattform.

Die einzelnen Transitionen »Systemabfrage«, »Abfragen« und »Zugriff« können ihre jeweiligen Anteile am Status schreiben. Sofern diese sich nicht überlagern, sind nebenläufige Schreibzugriffe möglich, daher werden die Kanten als partielle Schreibkanten gezeigt. Sobald der Bericht erzeugt wurde, kann er auf der rechten Seite durch die obere »Bericht anfertigen« Transition als Nachricht verpackt und versandt werden.

Auf der linken Seite können durch die beiden magentafarben dargestellten Transitionen »Erweiterung« entsprechende Erweiterungen an Agentenfabrik und Plattformfunktionalitäten vorgenommen werden. Die Erweiterungen ändern zwar den Zustand der Marken (bzw. der Entitäten/Netze dahinter), betreffen aber nicht dort bestehende Elemente. Eine Modellierung durch partielle Schreibkanten ist daher hier ebenfalls angebracht. Beide Transitionen erzeugen ein Ergebnis, welches über einen halb magentafarbenen, halb weißen Platz nach rechts für eine Rückmeldung (hellgelbe Transition »Bericht anfertigen«) weitergegeben wird.

Im linken unteren Bereich im Modell ist in Weiß die Agentenerzeugung dargestellt. Im reaktiven Fall wird ebenfalls der halb magentafarbene, halb weiße Platz eingesetzt, um eine entsprechende Rückmeldung zu liefern. Da in allen drei Fällen die Rückmeldungen semantisch gesehen nur kurze Bestätigungen sind, liegt es nahe, diese in nur einem Platz zu modellieren. Wurde reaktiv oder proaktiv die Agen-

tenerzeugung initialisiert, so kann mithilfe der Agentenfabrik die Instantiierung des Agenten erfolgen.

Auf der rechten unteren Seite des Modells ist die Entfernung bzw. der Umzug von Agenten dargestellt. Obwohl der Umzug eine komplexe Operation ist, kann er mit nur einer Transition dargestellt werden. Die Semantik der Transition umfasst die Abfrage der Wissensbasis des Agenten, die Entfernung und die Erzeugung der entsprechenden Nachrichten an die neue Plattform. Dabei ist die umgesetzte Erwartungshaltung, dass die Instantiierung auf der neuen Plattform generell möglich ist.

Zuletzt wird in der Mitte der Abbildung die Plattformfunktionalität gezeigt. Diese kann durch Agenten genutzt werden. Da es sich dabei um Funktionssammlungen handelt, welche im Allgemeinen keine Zustandsbehaftung aufweisen sollten, kann hier mit einer Testkante gearbeitet werden.

6.5. Zusammenfassung des Konzepts zur Plattform

In diesem Kapitel wurde ein Konzept zur Konstruktion der Plattform in der neu zu entwerfenden Architektur vorgestellt. Es orientiert sich dabei maßgeblich am Architekturmuster der Cloud-Nativity, deren erklärtes Ziel die Ausführbarkeit der Anwendung in unbekannten Umgebungen ist. Zum intendierten Funktionsumfang der Plattform zählt der Start und Stopp von Agenten, das Erweitern der bekannten Agentendefinition, ein Angebot plattformspezifischer und erweiterbarer Funktionalitäten, durch die eine Anpassung auf heterogenen Umgebungen erfolgen kann, sowie der Ausgabe von Zustandsinformationen durch die Plattform. Dabei wurden die Interaktionen zu diesen jeweiligen Punkten spezifiziert und das gesamte Zusammenspiel aller Komponenten in Form einer Petrinetzdarstellung in Abbildung 6.10 fixiert.

7. Plattformmanagement

Als dritter Teil der konzeptuellen Überlegungen wird in diesem Kapitel nun die Konzeption des Plattformmanagements vorgestellt. Dabei handelt es sich um einen der wesentlichen Beiträge des Autors im Rahmen dieser Arbeit.

7.1. Vorüberlegungen

Die im Rahmen der Grundlagen in Kapitel 2 vorgestellten Formalismen und Werkzeuge bieten eine solide Grundlage für ein Plattformmanagement und somit eine skalierende Simulation von interagierenden Agenten. Der Referenznetzformalismus bietet bereits sehr viele Aspekte der natürlichen Kommunikation zwischen Agenten in einer formalen Form. So können durch Referenznetze die Aspekte der Nebenläufigkeit, der Autonomie, der Synchronisationsfähigkeit und der hierarchischen Anordnung der Agenten realisiert werden.

Im Folgenden werden die zuvor herausgearbeiteten Fragestellungen bezüglich des Plattformmanagements erörtert. Dazu stellt Abbildung 7.1 noch einmal den Grobentwurf des Plattformmanagements dar.

7.1.1. Erzeugung und Vernichtung von Plattformen

Die zentrale Anforderung an das Plattformmanagement umfasst die Fähigkeit dynamisch Plattformen zu erzeugen und zu zerstören. Eine der Kernentscheidungen, die getroffen werden müssen, ist die Verortung der Kontrolle über die Anzahl der Plattformen und damit der physikalischen Ausdehnung der Simulation. Das Hinzuschalten und Entfernen von Plattformen entspricht im Wesentlichen der Skalierung von Prozessen auf mehrere physikalischen Berechnungsknoten. Aus diesem Grund sollte ein erster Ansatz für die Beschreibung eines Plattformmanagements auf dieser Basis beruhen.

Klassische Ansätze in diesem Bereich beruhen meist auf externen Metriken, wie der Prozessorlast oder Speichernutzung der laufenden Prozesse. Dabei steht die Generalität und Einfachheit der Methode im Fokus. Die betrachteten Kenngrößen sind meist leicht zu ermitteln und stammen aus der Laufzeitumgebung der

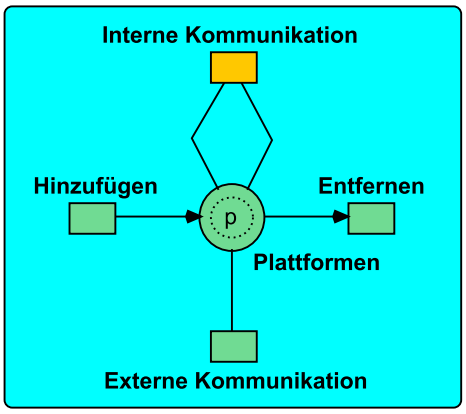

Abbildung 7.1.: Grober Erstentwurf des Plattformmanagements

Anwendung. Folglich benötigt die Software selbst keine Information über ihre physikalische Ausdehnung, ist dabei jedoch externen Infrastrukturmanagementtools ausgesetzt. Dieser Ansatz wird gemeinhin auch als Autoscaling (dt.: Autoskalierung) bezeichnet, da die Skalierung für die Software transparent geschieht. In aktueller Literatur finden sich vielfältige Ausführungen zu algorithmischer Ressourcenallokation und Autoskalierung. Um einige Beispiele zu nennen, fokussieren (H. Zhang u. a., 2016) und (Kaewkasi und Chuenmuneewong, 2017) Ressourcennutzung in Docker Umgebungen, (Xu, Toosi und Buyya, 2019) adressiert Autoskalierung mit Fokus auf verbesserte Energieeffizienz des Clusters. (Guerrero, Lera und Juiz, 2018) untersuchen Containerverteilung in Kubernetes-Umgebungen mit Fokus auf Reduktion des Netzwerkoverheads und (Rodriguez und Buyya, 2018) betrachten einen hybriden Ansatz, der über Autoskalierungs- und Ressourcenverteilungsfunktionalität verfügt.

Nachteilhaft am Autoskalierungs-Ansatz ist, dass Skalierung immer nur *nachträglich* erfolgen kann. In Fällen, in denen komplexere Berechnungen vorbereitet werden, kann die Infrastruktur erst reagieren, wenn die Berechnung bereits angelaufen ist. Somit liegt der Overhead, bis sich die verfügbaren Berechnungsinstanzen der Belastung angepasst haben, ungünstig und während der bereits laufenden Berechnung. Dieser Overhead ist um so schwerwiegender, je aufwändiger die Bereitstellung zusätzlicher Berechnungsinstanzen ist. Der gegenwärtige Wandel von virtuellen Maschinen und komplexen Deploymentformaten hin zur Containerisierung entschärft diesen Umstand, vorhanden ist er jedoch nach wie vor. Der systematische Ablauf ist in Abbildung 7.2 dargestellt.

Ein interessanter Ansatz wird in (Nguyen u. a., 2020) dargestellt, bei dem der externe Autoscaler durch Informationen aus einem Monitoring Werkzeug angereichert wird. Dadurch kann die Skalierung deutlich differenzierter erfolgen, als

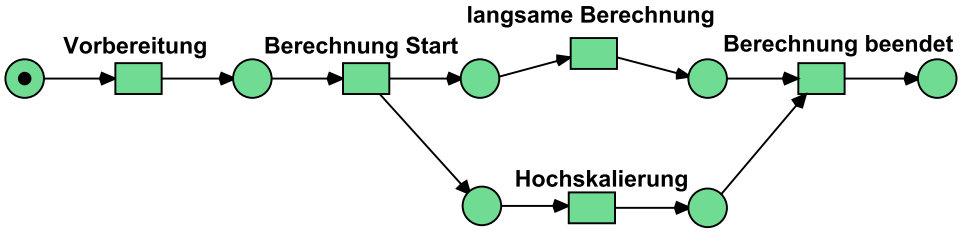

Abbildung 7.2.: Ablauf der Skalierung mittels externem Autoscaler

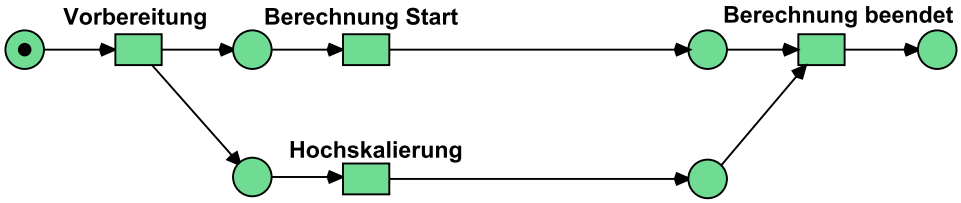

Abbildung 7.3.: Ablauf der Skalierung mittels Anwendung

beim klassischen Ansatz, der nur offensichtlich verbrauchte Ressourcen betrachtet und die Anwendung selbst als Blackbox handhabt. Nichtsdestotrotz bleibt das Problem der reaktiven Skalierung bestehen.

Ein weiterer Nachteil ergibt sich durch die Auslagerung der Skalierung an externe Infrastrukturtools. Die Software wird somit abhängig von der Einwirkung eines externen Tools und gibt Autonomie auf. Insbesondere dieser Aspekt ist schwer mit der hierarchischen Agentenmetapher vereinbar, bei der auch die Plattform (die Software) als Agent betrachtet werden kann und somit möglichst hohe Autonomie aufweisen sollte.

Diese Punkte legen nahe, dass eine Kontrolle aus der Software selbst über ihre physikalische Ausdehnung wünschenswert ist. Allerdings ist dies nicht einfach pauschal umzusetzen. Eine Kontrolle über die Skalierung aus der Simulation ermöglicht das proaktive Skalieren in Vorbereitung aufwändiger Berechnungen. Mit geschickter Implementation kann somit der Overhead des Hochfahrens neuer Instanzen verlagert oder sogar weitestgehend aus der Zeitspanne der Berechnung verbannt werden. Um der Agentenmetapher treu zu bleiben, sollte den Plattformen also die Möglichkeit eingeräumt werden, die Erzeugung neuer Plattformen anzustoßen. Der Ablauf in Abgrenzung zu Abbildung 7.2 findet sich hierzu in Abbildung 7.3.

Auch der Aspekt der Autonomie kann hervorragend abgedeckt werden, da die Software alle Kontrolle behält.

Das Hauptargument gegen eine Kontrolle aus der Anwendung heraus liegt in der erhöhten Komplexität der Implementation der Anwendung. Neben den fachlichen Anforderungen muss die gesamte technische Infrastruktursteuerung vom Entwickler bzw. Modellierer umgesetzt werden. Bibliotheken können die Entwicklung beschleunigen, das grundlegende Problem bleibt jedoch bestehen. Während dies eine feingranulare Steuerung der physikalischen Ausdehnung ermöglicht, verlangsamt es die Entwicklung maßgeblich. Darüber hinaus ist es sehr wohl möglich, dass zur Entwicklungszeit keine Information über die spätere Laufzeitumgebung vorhanden ist. Individuelle Infrastrukturszenarien lassen sich so nur ungenügend antizipieren und entsprechend in der technischen Programmierung berücksichtigen.

Ein weiterer Aspekt, welcher gegen eine direkte Steuerung aus der Simulation heraus spricht, ist die Verankerung von Laufzeitwissen in der Anwendung. In späteren Abschnitten werden genau die Anforderungen Verankerung von Laufzeitwissen in der Anwendung zu minimieren eine Rolle spielen. Um konfliktbehaftete Anforderungen an die Software zu vermeiden, sollte dies also auch hier bereits berücksichtigt werden.

Somit ergibt sich aus den beiden betrachteten Ansätzen, der Skalierung aus der Anwendung und der Skalierung aus der Infrastruktursteuerung, dass sich beide in ihren Vor- und Nachteilen in etwa aufwiegen. Abschließend soll noch ein hybrides Modell betrachtet werden, welches versucht die Vorteile beider Ansätze soweit wie möglich zu kombinieren. Da hybride Modelle beliebige Mengen von Anteilen der beiden Ansätze aufweisen können, ist dabei zunächst zu klären welche Problempunkte primär adressiert werden sollten. Eine möglichst angenehme Benutzbarkeit aus Sicht des Modellierers soll dabei gemeinsam mit der Effizienz im Vordergrund stehen.

Folglich liegt es nahe, zunächst die Notwendigkeit des technischen Wissens aus der vom Modellierer zu erledigenden Arbeit zu entfernen. Eine einfache Schnittstelle wäre wünschenswert, bei der bestimmte Zustände der physikalischen Ausdehnung angefordert werden können, die aber sonst keine Detailinformationen erfordert. Eine mögliche Realisierung dieser Anforderung besteht in der Einführung einer separaten Management-Komponente, welche die technische Interaktion kapselt. Darüber hinaus hält die Anwendung somit auch kein Wissen über die Laufzeitumgebung. Diese Lösung ist ebenfalls mit dem Autonomiegedanken vereinbar, da die Kontrolle über die Ausdehnung der Simulation nach wie vor in der Simulation selbst liegt.

Feature	*Autoscaler*	*Anwendung*	*Hybrid*
Umgebungskontrolle durch Anwendung	-	++	+
Anwendungsautonomie	-	++	++
Frühzeitiges Hochskalieren	-	++	++
Infrastruktur-Unabhängigkeit der Anwendung	++	- -	+
Unkomplexes Deployment	+	-	o
Technisches Wissen beim Modellierer verzichtbar	++	- -	+

Tabelle 7.1.: Verschiedene Ansätze zur Skalierungskontrolle

Nachteilhaft ist die Lösung einer weiteren Management-Komponente insofern, als diese separat bereitgestellt werden muss und somit die Komplexität der Anwendung steigt.

Abschließend kann festgestellt werden, dass eine hybride Lösung mit präparierter Management-Komponente viele Vorteile beider reinen Ansätze mit überschaubaren Nachteilen vereinen kann und somit die präferierte Lösung darstellen sollte.

Die insgesamt drei verschiedenen Ansätze sind abschließend in Tabelle 7.1 zusammengefasst, um einen schnellen Überblick zu gewährleisten. Schematische Darstellungen zur Kontrolle der Ansätze finden sich in Abbildung 7.4. Dabei symbolisieren Pfeile den Kommunikationsfluss und gestrichelte Pfeile eine äußere Inspektion ohne Kommunikation. Gestrichelte Kästen beschreiben neu hinzuzufügende Komponenten, während solche mit einer durchgehenden Linie bestehende Komponenten referenzieren.

Aus den Ausführungen ergibt sich der erste Kernbestandteil der Architektur: Die Kontrolle über die physikalische Ausdehnung der Simulation sollte aus der Simulation möglich sein, aber mit minimalem Implementationsaufwand bzw. technischem Wissen auf Seite des Modellierers.

Basierend auf dem Grundgedanken, dass auch das abstrakte Plattformmanagement als (komplexer) Agent betrachtet werden kann, sollte die Idee der Autoskalierung jedoch nicht gänzlich außer Acht gelassen werden. Autoskalierung ist in diesem Fall als Spezialfall der allgemeinen Nachrichtenbasierten Skalierung zu sehen, welche explizit vom Plattformmanagement ausgeht. Aus der Agentenperspektive des Plattformmanagements ist es naheliegend die hybride bzw. anwendungsgesteuerte Skalierung und die Autoskalierung als proaktive und reaktive Skalierung zu behandeln. Aus diesem Grund sollte das Modell beide Varianten unterstützen, der primäre Fokus sollte jedoch auf der reaktiven Skalierung liegen, wie zuvor ausgeführt wurde.

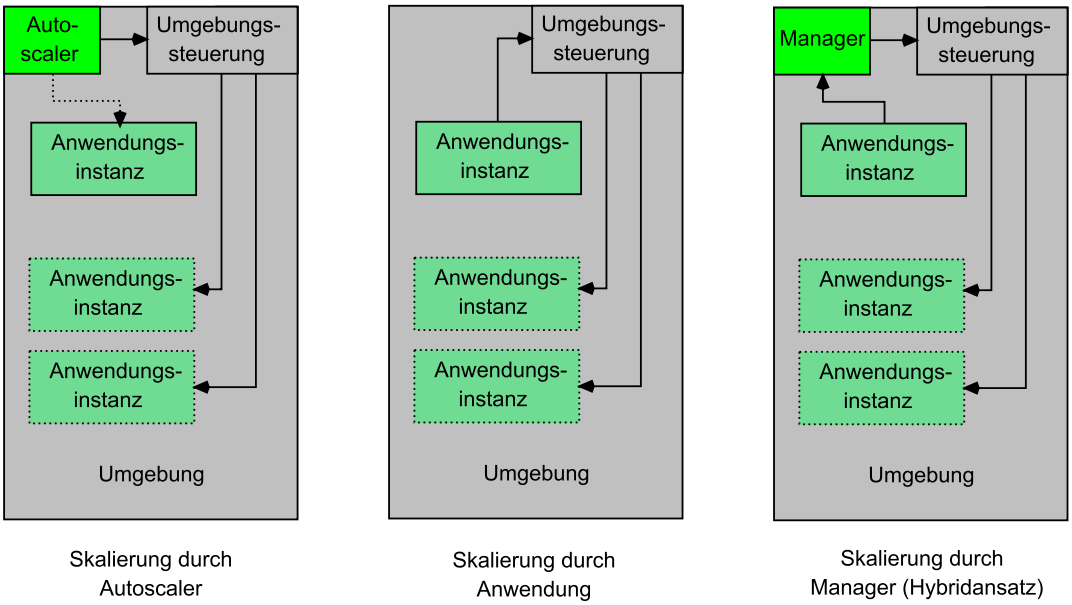

Abbildung 7.4.: Kommunikationsfluss (Pfeile) und Inspektion (gestrichelte Pfeile) der Ansätze zur Skalierungskontrolle. Gestrichelte Kästen stellen neu zu erschaffende und durchgehende Kästen bestehende Komponenten dar.

Damit folgt ein weiterer Bestandteil: Neben der reaktiven Skalierung durch Plattformen und Agenten soll das Plattformmanagement auch proaktive Skalierung beherrschen.

Basierend auf der später gewählten technischen Umsetzung ergibt sich die Möglichkeit der Autoskalierung jedoch auf natürlichem Wege, da diese wie eingehend erwähnt, vielfach untersucht und generell verfügbar ist. Folglich wird die Autoskalierung in Kapitel 10 zum Realisierungskonzept des Plattformmanagements eine untergeordnete Rolle spielen.

7.1.2. Kommunikation über das Plattformmanagement

Als weiteren Punkt gilt es die Kommunikation zwischen Plattformen und zwischen Agenten zu adressieren. Dabei unterscheiden sich diese beiden Arten der Kommunikation nur insofern, als Kommunikation zwischen Agenten auch innerhalb einer Plattform stattfinden kann. Beim Transfer von Nachrichten über Plattformgrenzen kann die Agentennachricht in eine Plattformnachricht verpackt werden, welche von der jeweiligen Zielplattform entsprechend verarbeitet werden kann. Daher soll es aus Sicht des Plattformmanagements genügen, generelle Nachrichten zwischen Plattformen zu modellieren.

Wird jedoch die Sicht der Agenten selbst eingenommen, sollte eine Kommunikation mit Synchronisations- oder Task-Transitions-Semantik zur Verfügung stehen, wie eingehend in Kapitel 5 erläutert wurde. Ein generelles Nachrichtensystem ist dafür unter Umständen nicht ausreichend. Zu fordern, dass ein System mit physikalisch voneinander getrennten Plattformen garantiert, dass Plattformen zu jeder Zeit verfügbar sind, ist nicht realistisch. Handelt es sich bei der Synchronisation also um eine Operation mit komplexer Schreib-Semantik mit mehreren Teilschritten, sollte vom Plattformmanagement sichergestellt werden, dass etwaige Nachrichten nicht verloren gehen und die Konsistenz des Gesamtsystems auf lange Sicht gewahrt bleibt. Für einfache Nachrichten, die lesenden Charakter haben oder nach Eingang komplett lokal abgearbeitet werden können, wirkt eine derartige Zusicherung aber übersetzt und kann negative Einflüsse auf die Performance des Systems haben. Aus diesem Grund sollte ein alternativer, einfacher Kommunikationskanal zur Verfügung stehen, welcher auf eine möglichst geringe Belastung des Gesamtsystems optimiert ist.

Nach den Anforderungen sind die Interaktionen der Plattformen häufig weniger komplexer Natur und von vorhersehbarerem Umfang, als es bei den beliebigen simulierten Agenten der Fall sein muss. Weitere Details zu einer möglichen Umsetzung der Kommunikationsstrukturen werden in den Erörterungen zum Realisierungskonzept der Agentenkommunikation in Kapitel 12 aufgeführt.

Dennoch kann an dieser Stelle bereits festgehalten werden, dass zwei Kommunikationsstrukturen umgesetzt werden sollen. Dabei wird die Kommunikation zwischen Plattformen in erster Linie über den unkomplizierten Weg erfolgen und die Synchronisation von Agenten über den Weg mit erhöhten Konsistenzgarantien.

Darüber hinaus muss für die Kommunikation eine Mechanik bereitgestellt werden, durch die Plattformen von ihrer jeweiligen Existenz erfahren können. Dies ist eine Grundvoraussetzung für eine Kommunikation, die nicht auf Broadcastbasis arbeitet. Analog zu der Webservice-Sicht auf die Architektur kommt hierfür eine Service Discovery infrage, die genau diese Funktionalität abdeckt. Die Details zur Methodik zur Identifikation werden ebenfalls im Realisierungskonzept der Agentenkommunikation in Kapitel 12 erörtert.

7.1.3. Externe Betrachter des Gesamtsystems

Das Plattformmanagement bildet die höchste Abstraktionsstufe der zu konstruierenden Architektur. Wird ein entsprechender Benutzer des Systems betrachtet, so liegt es nahe, dass dieser in seiner Interaktion mit dem System zunächst das Plattformmanagement ansprechen wird. Für diese Art der Nutzerinteraktion sollte das Plattformmanagement folglich entsprechende Schnittstellen bereitstellen.

Zur Bestimmung der Art dieser Schnittstellen soll im Folgenden die Nutzung durch einen Benutzer betrachtet werden.

Das System startet aus einem bestimmten Ausgangspunkt mit einer Aufgabe. Während der Abarbeitung könnte der Nutzer am technischen Status des Systems interessiert sein, sodass dieser auslesbar sein sollte. Unabhängig vom (technischen) Status kann auch das Voranschreiten der fachlichen Berechnung von Interesse sein. Die Granularität dieses Fortschritts könnte sich dabei von einzelnen Events der Berechnung bis hin zu fertigen aggregierten Ergebnissen bewegen. Das Auslesen eines finalen Ergebnisses des Systems durch den Benutzer kann somit auch als Spezialfall des Voranschreitens der fachlichen Berechnung gesehen werden.

Zusammengefasst sollte das System in der Lage sein, über seinen Ausgangszustand, seinen technischen Status und sein fachliches Voranschreiten zu berichten.

7.2. Funktionsumfang des Plattformmanagements

Analog zur Beschreibung der Plattform im letzten Kapitel dient dieser Abschnitt der Beschreibung des Funktionsumfangs des Plattformmanagements. Dabei werden Schritt für Schritt die in Abschnitt 5.4 und im Laufe von Kapitel 6 formulierten Anforderungen aus Agentenkommunikation und der Plattform selbst aufgegriffen und daraus entsprechende Komponenten im Plattformmanagement hergeleitet.

7.2.1. Globale Identifikation

Der erste Punkt umfasst eine Mechanik zur Ermöglichung globaler Identifikation. Wie bereits am Ende des Kapitels zu Agentenkommunikation angeregt, ist es nicht unbedingt erforderlich, zu diesem Zwecke einen dezidierten Verzeichnisdienst einzusetzen. Durch die Notwendigkeit eines Nachrichtenbrokers kann dieser ebenfalls verwendet werden, um eine persistente Zuordnung von Identifikatoren zu ermöglichen. Dennoch ist auch der Einsatz eines dedizierten Verzeichnisdienstes denkbar. Der Ansatz eines dedizierten Verzeichnisdienstes soll aber aus den eben beschriebenen Gründen nicht explizit abgebildet werden.

Zur Realisierung der Zuordnung muss bei der Erzeugung einer Einheit der Identifikator eben dieser an den Nachrichtenbroker herangetragen werden. Dieser speichert die Nachricht in einer persistenten Art und Weise, sodass sie später von allen Plattformen beliebig oft abgerufen werden kann, wenn diese dazu bereit sind. Dabei müssen sowohl die Identifikation von Plattformen als auch die von Agenten gespeichert und aufeinander abgebildet werden. Wird zusätzlich die übergeordnete Infrastruktur, welche mehrere Plattformmanagements miteinander verbinden

könnte, betrachtet, liegt es nahe, auch für das lokale Plattformmanagement eine Identifikation zu vergeben.

Während diese Ausgestaltung der Funktionalität für den konzeptuellen Teil der Arbeit genügen soll, sollte für Implementation erwägt werden, einen entsprechenden Cache in der Plattform vorzuhalten.

7.2.2. Kommunikation

Die Bereitstellung von plattformübergreifenden Kommunikationsfunktionen zählt zu den elementaren Anforderungen an das Plattformmanagement. Wie zuvor erläutert, soll dabei zwischen einfacher und komplexer Kommunikation unterschieden werden. Durch die sehr verschiedenen Eigenschaften dieser beiden Kommunikationsformen entstehen auch verschiedene Anforderungen an die Umsetzung auf der Ebene des Plattformmanagements.

Einfache Kommunikation zwischen Plattformen

Als erste Form der Kommunikation zwischen zwei Plattformen muss die einfache Kommunikation realisiert werden. Dabei kann aus der einfachen Agentenkommunikation rekapituliert werden, dass sich dabei um Kommunikation mit unidirektionalem Informationsaustausch und ohne Zusicherung der Zustellung handelt. Daher ist an dieser Stelle kein dynamischer Zwischenspeicher für Nachrichten vorzusehen. Ferner müssen beide Plattformen aktuell erreichbar sein, um Nachrichten verschicken zu können. Aus diesem Grund ist ebenfalls eine Vermittlungsplattform für diese Art Kommunikation erforderlich. Somit muss vonseiten des Plattformmanagements lediglich ein Übertragungsweg zwischen zwei Plattform bereitgestellt werden bzw. eine Möglichkeit für zwei Plattformen, sich zu synchronisieren.

Kommunikation zwischen Plattformen mit Garantien

Der herausforderndere Ansatz bei Betrachtungen der Kommunikation ist die Kommunikation mit Zustellungsgarantie. Hierbei sollen auch komplexe Informationsaustausche realisiert werden können. In Kapitel 5 wurde bereits motiviert, dass die Implementation eines nebenläufigen Saga-Patterns mit guter Chance ausreichend für die Realisierung der Funktionalität ist.

Aus diesem Grund muss vonseiten des Plattformmanagements im Wesentlichen eine Komponente beigesteuert werden: Einer oder mehrere Ablaufmanager, welche die Abarbeitung der Kommunikationsbestandteile überwachen und dafür sorgen,

dass nachfolgende Kommunikationspartner entsprechend informiert werden. Dies soll natürlich insbesondere dann der Fall sein, wenn einzelne Kommunikationspartner (oder der Ablaufmanager selbst) für den Moment nicht erreichbar sind.

Aus diesem Grund ist es wichtig, dass der Ablaufmanager im Kern mit dem Nachrichtenbroker operiert, da dieser genau die Eigenschaft von persistenten Nachrichten, welche nachträglich abgerufen werden können, bereitstellt. Da Plattformen die Nachrichten nur regelmäßig proaktiv abrufen (pollen) und nicht durch den Nachrichtenbroker informiert werden, muss dem Nachrichtenbroker auch keine derartige Funktionalität hinzugefügt werden. Das Polling gestaltet sich hier als effizienter, da so der Nachrichtenbroker keine Liste darüber führen muss, welche Plattform aktuell verfügbar und erreichbar sind, sondern einfach auf eingehende Anfragen warten kann.

7.2.3. Erzeugung und Zerstörung von Plattformen

Die Erzeugung und Zerstörung von Plattformen bildet das Herzstück der Funktionalitäten des Plattformmanagements. Es ist eben diese Funktionalität, welche die dynamische Skalierung und Anpassung der Größe der Simulation ermöglicht. Sie soll an dieser Stelle in drei Arten adressiert und definiert werden. Dabei umfassen zwei Arten die reaktive Skalierung auf bestimmte Anfragen hin und eine den Ansatz der proaktiven Skalierung, bei dem das Plattformmanagement selbst aktiv wird.

Die Definition einer Plattform ist dabei stets einheitlich. Dies ergibt sich aus dem Funktionsumfang und den Ausführungen der Plattform selbst, wie sie in Kapitel 6 beschrieben wurden. Die Heterogenität einer Plattform wird durch die auf ihr verfügbaren Funktionalitäten abgebildet. Diese sind dynamisch erweiterbar und können zur Laufzeit hinzugefügt werden, sodass die Bereitstellung einer Plattform mit Grundlagenfunktionalität als Referenzausgangspunkt hinreichend ist.

Zu diesem Zwecke bietet sich an, eine Plattformdefinition vorzuhalten, welche als Muster eingesetzt werden kann, sobald neue Plattformen erzeugt werden sollen. Dennoch ist es denkbar, dass mit der Zeit Änderung an der Plattformdefinition nötig sein könnten. Aus diesem Grund sollte die Plattformdefinition zusätzlich (extern) änder- und erweiterbar sein.

Reaktive Skalierung bezogen auf externe Nachrichten

Die einfachste Form der Skalierung des Systems ergibt sich durch die Reaktion auf eine externe Nachricht. Beim Absender kann es sich theoretisch um ein anderes Plattformmanagement, einzelne Plattformen oder Agenten handeln. Praktisch ist diese Operation aber am ehesten mit dem manuellen Eingriff durch einen realen

Nutzer gleichzusetzen. Dies steht nicht im Widerspruch, da auch reale Nutzer als Agenten ihrer jeweiligen Plattform interpretiert werden können. Eine Prüfung interner Faktoren bezüglich der existenten Plattformen entfällt hierbei.

Für produktive Implementationen ist es selbstverständlich von großer Wichtigkeit ein entsprechendes Sicherheitskonzept zu implementieren, um ungewünschte Effekte auf die Menge der Plattformen zu vermeiden.

Reaktive Skalierung bezogen auf Agenten einer der verwalteten Plattformen

Wie in der Anforderungsanalyse motiviert kann eine gezieltere Skalierung erreicht werden, indem Agenten Hinweise auf geplante Vorhaben und dafür benötigte Kapazität geben. Zu diesem Zweck können Agenten sich ebenfalls beim Plattformmanagement melden und analog zur im letzten Abschnitt beschriebenen reaktiven Skalierung Anfragen stellen. Der Weg zu solch einer Anfrage ist dabei nicht grundlegend anders als sonstige Kommunikation vom Agenten aus. Auch die Plattform, auf der der Agent heimisch ist, kann eine solche Nachricht einfach weiterleiten. Im Plattformmanagement muss mit einer solchen Nachricht jedoch anders umgegangen werden, und sie muss als eingehende Nachricht bezüglich der Plattformskalierung interpretiert werden.

Proaktive Skalierung (Autoskalierung)

Als letzten Ansatz kann noch die proaktive Skalierung betrachtet werden. Dabei handelt es sich im Wesentlichen um den Ansatz der Autoskalierung, da das Plattformmanagement selbstständig und ohne äußeren Einfluss entscheidet, die Anzahl der Plattform anzupassen. Agenten können dann auf die neu erzeugten Plattformen umziehen. Während bereits dargelegt wurde, dass dieser Ansatz gegenüber der zuvor beschriebenen Rückmeldungen durch Agenten gewisse Nachteile aufweist, sollte er der Vollständigkeit halber und auch um die proaktiven Elemente des Plattformmanagements einzubeziehen, verfügbar sein.

Das Plattformmanagement muss sich bei der proaktiven Skalierung jedoch auf einige Grunddaten beziehen, um eine sinnvolle Änderung am Zustand herbeiführen zu können. Dabei sind insbesondere extern zu beobachtende Faktoren der Plattform von Interesse. Folglich muss das Plattformmanagement versuchen, durch Messungen Daten der Plattform zu erheben und diese auch ggf. gegen Kosten der Erzeugung einer Plattform abwägen.

7.2.4. Umzug von Plattformen

Eine weitere denkbare Funktionalität ist der Umzug von Plattformen. Dabei orientiert sich die Umsetzung entlang der Einheitentheorie an der Fähigkeit von Agenten, ein Umzug zwischen Plattformen zu realisieren. Der Unterschied im hier betrachteten Fall liegt jedoch darin, dass jenseits der Ebene des Plattformmanagements lediglich eine abstrakte Infrastruktur modelliert ist. Demnach ist es durchaus denkbar, dass auf anderen Plattformmanagements ebenfalls die Möglichkeit besteht, eine bestimmte Plattform zu beheimaten. Die Chancen dazu stehen jedoch im Allgemeinen eher schlecht. Auch im Kontext des hier beschriebenen Plattformmanagements müsste ein weiteres Plattformmanagement mit der exakt gleichen Plattformdefinition arbeiten, um Plattformen eine bilaterale Umzugsmöglichkeit zu bieten. Wäre die Plattformdefinition des weiteren Plattformmanagements zumindest kompatibel bzw. eine Obermenge der lokalen Plattformdefinition wäre eine unilateral Umzugsmöglichkeit realisierbar. Eine andere Möglichkeit wäre es, eine Art Transformation zwischen den Plattformmanagements zu realisieren.

Während also generell der Umzug von Plattformen zwischen verschiedenen Plattformmanagements denkbar und möglich ist, überschreitet diese Betrachtung den Blickwinkel, den diese Arbeit einnimmt. Der Umzug zwischen verschiedenen Plattformmanagements soll daher nicht explizit adressiert werden. Dennoch kann es sich bei einem Plattformumzug um eine andere, aber sinnvolle Operation innerhalb des Plattformmanagements handeln. Dies liegt darin begründet, dass die Grundannahme davon ausgeht, dass bei einer Implementation mehrere physikalische Knoten innerhalb eines Plattformmanagements organisiert werden. Ein solcher Umzug kann beispielsweise die Beschleunigung des gemeinsamen Umzugs vieler Agenten auf eine neue physikalische Struktur bewirken. Ebenfalls ist es denkbar, dass nach dem Ausfall einer Plattform eine äquivalente Plattform an einer anderen Stelle hochgefahren wird. Effektiv handelt es sich dabei auch um einen Umzug.

Um die hier beschriebenen Möglichkeiten also bereitzustellen, solle das Plattformmanagement ein Plattformmanagement-internen Umzug von Plattformen ermöglichen.

7.2.5. Globale Zustandsberichte

Zuletzt sollte die Möglichkeit bestehen, für externe Anfragen globale Zustandsinformationen des Gesamtsystems aller Plattformen bereitzustellen. Der Detailgrad und die Inhalte dieser Zustandsberichte sollten von der Auskunftsfähigkeit und -willigkeit der jeweiligen Plattform abhängen. Aus diesem Grund liegt es nahe, die Zustandsabfragen lediglich an jede einzelne Plattform zu delegieren und auf die

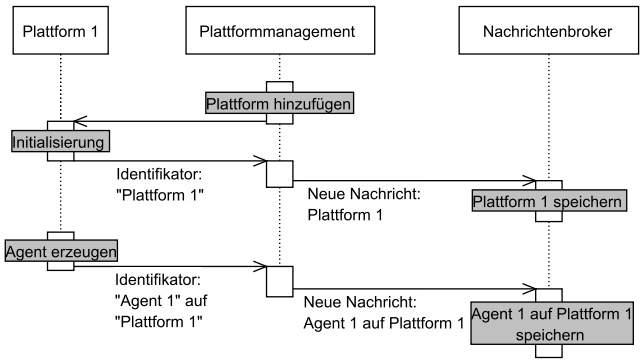

Abbildung 7.5.: Speichern von Identifikatoren von Plattformen und Agenten

Anfrage mit einer aggregierten Auswertung zu antworten. Die eingesetzte Plattformdefinition und die Anzahl der aktiven Plattformen könnte dabei auch von Interesse sein.

Neben den konkreten Zustandsberichten ist auch die Bereitstellung eines Simulationsfeeds für externe Beobachter von Interesse. Die Komplexität befindet sich dabei im Wesentlichen auf der Ebene der Plattformen, während das Plattformmanagement selbst dabei lediglich ein Aggregationsmedium bereitstellen muss.

7.3. Entwurf der Abläufe im Plattformmanagement

Nachdem die generellen Komponenten des Plattformmanagements vorgestellt wurden, werden im Folgenden die abstrakten Abläufe der einzelnen Funktionalitäten mit ihren beteiligten Einheiten dargestellt.

7.3.1. Globale Identifikation

Abbildung 7.5 zeigt den Entwurf der Interaktion zur Identifikation von Plattformen und Agenten. Die konkrete Erzeugung der Agenten ist in diesem Kapitel nicht modelliert, da sie dem Funktionsumfang der Plattform zuzurechnen ist und dort behandelt wurde. Die Erzeugung von Plattformen wird später separat in Abschnitt 7.3.3 vorgestellt und ist daher hier ebenfalls vereinfacht dargestellt.

Im Allgemeinen liegt die Verantwortung für die Erzeugung von Identifikatoren bei den Plattformen. Im Falle von Agenten können diese ebenfalls Zuordnungen zwischen etwaigen lokalen Identifikatoren und globalen Identifikatoren vorhalten. Bei der Plattform selbst sollte dies in der Regel entfallen können, sofern Plattformen

intern kein Identifikationsschema benötigen oder dieses nicht vom einheitlichen globalen Identifikationsschema abweicht.

Nachdem eine Plattform hinzugefügt wurde, meldet diese sodann in ihrem Initialisierungsprozess ihre Identifikation an das Plattformmanagement. Da, wie bereits erläutert, der Nachrichtenbroker die Funktionalität des Verzeichnisdienstes übernehmen soll, meldet das Plattformmanagement die Plattformidentifikation an diesen weiter. Der Nachrichtenbroker erzeugt eine persistente Nachricht, welche von allen Interessenten beliebig oft abgerufen werden kann. Dieser Ablauf ist hier nicht dargestellt, aber beispielsweise in Abbildung 5.6 zur einfachen Kommunikation zwischen Agenten zu beobachten.

Im Falle der Erzeugung von Agenten übermittelt ebenfalls die Plattform die Identifikation über das Plattformmanagement an den Nachrichtenbroker. Die Prozesse unterscheiden sich somit dahingehend leicht, dass die Anmeldung entweder durch die Einheit selbst oder die übergeordnete Einheit erfolgt. Die Begründung für diese Modellierung liegt ebenfalls in der Grundannahme, dass die Plattform die größte nicht-verteilte Einheit darstellt. Indem die Identifikation in die Verantwortung der Plattform übertragen wird, kann eine effiziente lokale Identifikation durch das (für den spezifischen lokalen Kontext) globale Wissen der Plattform erfolgen.

7.3.2. Kommunikation

Dem Plattformmanagement kommen wichtige Aufgaben in Bezug auf die Realisierung der Kommunikationsstrukturen zu. Dennoch erweitert die darüber hinausgehende Gesamtbetrachtung des Plattformmanagements die bereits geschilderten Abläufe der Kommunikation nicht maßgeblich. Aus diesem Grund sind die Beschreibungen der Abläufe in Abbildung 5.6 zur einfachen Kommunikation, sowie Abbildung 5.7 zur komplexen Kommunikation ausreichend.

Als Erweiterung kann noch betrachtet werden, dass der Nachrichtenbroker als konstante Einheit im Plattformmanagement mit seiner Erzeugung beginnt zu existieren. Eine explizite Konstruktion soll daher nicht gegeben werden und wäre ebenfalls abhängig von der gewählten Implementation. Dabei sollte jedoch beachtet werden, dass der Nachrichtenbroker selbst eine gewisse Menge an (physikalischer) Redundanz aufweist, damit sich aus ihm weder ein Flaschenhals, noch ein single point of failure entwickelt.

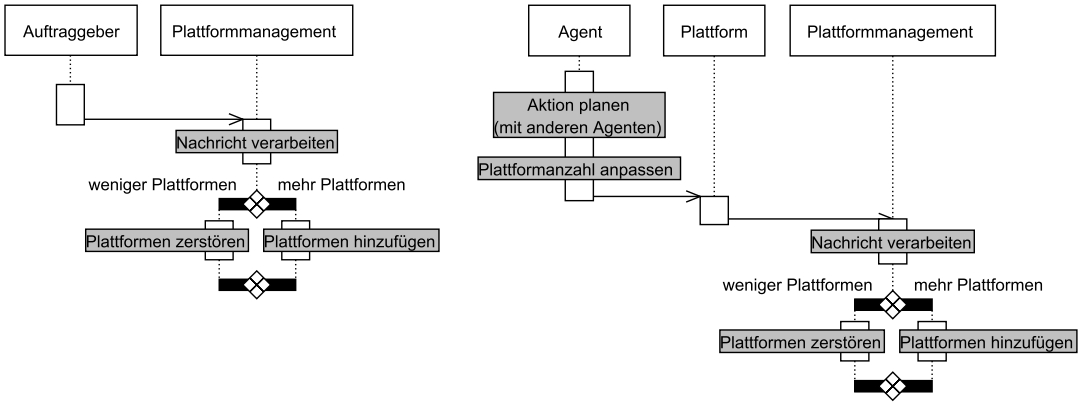

Abbildung 7.6.: Reaktives Anpassen der Menge an Plattformen. Durch externe Nachrichten (links) bzw. durch lokale Agenten (rechts)

7.3.3. Erzeugung und Zerstörung von Plattformen

Die Verwaltung der Anzahl an Plattformen wird je nach geschildertem Anwendungsfall definiert. Generell sollten Änderungen der Menge an Plattformen nicht klassisch auftragsbasiert erfolgen, sondern durch die Beschreibung eines gewünschten Referenzzustandes. Es ist sodann Aufgabe des Plattformmanagements, den aktuellen Zustand in den gewünschten Referenzzustand zu überführen. Dies hat den Vorteil, dass eingehende Anfragen die Eigenschaft der Idempotenz aufweisen und somit beliebig oft beim Plattformmanagement eingehen können, ohne dass das Ergebnis vom einmaligen Eingehen abweicht. Insbesondere im Kontext verteilter Systeme, bei denen Verbindungen ausfallen können oder fehlerhaft sein können, ist dies eine sehr wünschenswerte Eigenschaft. Dieses generelle Vorgehensmuster trägt den Namen »Reconciler-Pattern« und wird beispielsweise in (Garrison und Nova, 2017) beschrieben.

Reaktive Skalierung

Abbildung 7.6 zeigt die beiden Arten, die Anzahl der Plattform reaktiv anzupassen. Auf der linken Seite befindet sich die Verarbeitung auf Basis eines externen Auftraggebers, welcher direkte Behandlung durch eine eingehende Nachricht bewirkt. Auf Basis der Verarbeitung der Nachricht kann bestimmt werden, ob der aktuelle Zustand des Plattformmanagements weniger oder mehr Plattform benötigt, um den gewünschten Zustand zu erreichen. Je nach Ergebnis dieser Untersuchung kann das Plattformmanagement sodann Plattform zerstören oder hinzufügen. Diese Operationen führen zu Initialisierung bzw. zum Herunterfahren innerhalb der Plattformen, welche Teil der Plattform sind und hier nicht detailliert beschrieben werden.

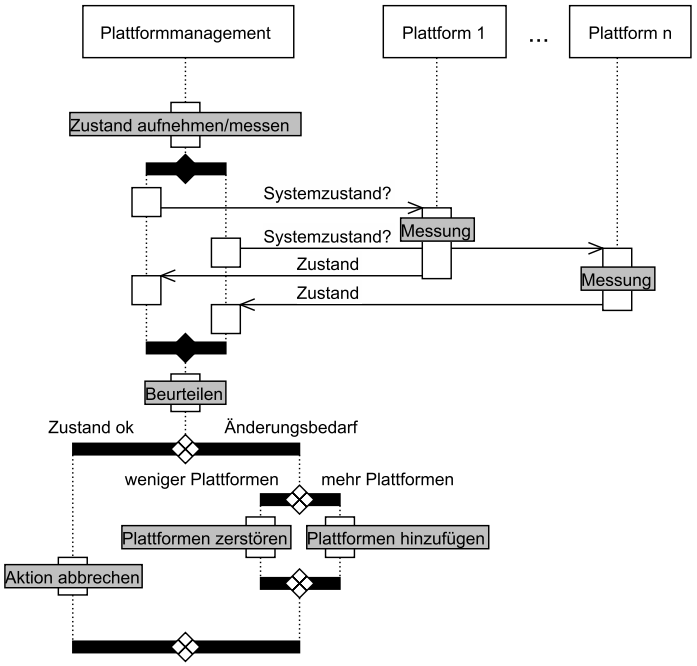

Abbildung 7.7.: Proaktives Anpassen der Menge an Plattformen durch Zustandsüberprüfung (Autoskalierung)

Der Ablauf einer Anpassung durch einen Agenten auf einer Plattform innerhalb des Plattformmanagements ist sehr ähnlich und auf der rechten Seite dargestellt. Aus der gemeinsamen Planung mit weiteren Agenten gibt ein Agent über seine lokale Plattform den Hinweis an das Plattformmanagement eine entsprechende Anpassung vorzunehmen. Die Anpassung selbst erfolgt wie eben beschrieben.

Proaktive Skalierung (Autoskalierung)

Während die Anpassung der Menge an Plattformen nicht wesentlich von der reaktiven Variante abweicht, ist der Prozess, welcher zu diesem Ereignis führt, bei der proaktiven Skalierung anders gelagert. Abbildung 7.7 zeigt den Ablauf der proaktiven Skalierung. Im Wesentlichen muss im proaktiven Fall das Plattformmanagement die Entscheidung für eine gewünschte Anzahl an Plattformen übernehmen. Zu diesem Zweck muss es eine gewisse Form von Einsicht in die jeweiligen Plattformen erlangen können. Dabei ist insbesondere die Auslastung von Interesse, da diese häufig ohne größeren Eingriff in die Plattform durch Messung ermittelt werden kann.

Nachdem proaktiv eine Beurteilung des aktuellen Zustandes gestartet wurde, führt das Plattformmanagement an jeder Plattform eine Messung durch bzw.

sendet eine entsprechende Nachricht an die Plattform. Sollte sich eine angemessene Auslastung ergeben, ist keine Handlung erforderlich. Sollten Plattform jedoch extrem entlastet oder extrem ausgelastet sein, so empfiehlt sich die Anpassung in die entsprechende Richtung. Mit dem Ergebnis der Berechnung wird wie in den Fällen der reaktiven Skalierung verfahren. Speziell für den entlasteten Fall sollte jedoch berücksichtigt werden, ob durch einen Plattformabbau eine Überlastung der verbleibenden Plattformen entsteht. Dabei handelt es sich um ein übliches Problem in verteilten Systemen und eine Lösung bedarf meist Informationen aus dem Gesamtsystem, wie sie hier im Rahmen der proaktiven Skalierung bereits abgefragt werden.

7.3.4. Umzug von Plattformen

Der Ablauf eines Plattformmanagement-internen Umzugs ist auf abstrakter Ebene leicht beschrieben. Die Plattform wird aus der Menge der aktiven Plattformen entfernt und ein entsprechender Auftrag zur Wiederherstellung der gewünschten Plattformanzahl aufgenommen. Dies entspricht im Wesentlichen der Zerstörung einer Plattform und der Abarbeitung einer Anfrage zur reaktiven Skalierung. Darüber hinaus existieren bereits Lösungen für diesen Aspekt, wie beispielsweise in der Arbeit (Rattihalli u. a., 2019). Aus diesem Grund wird an dieser Stelle auf die Darstellung eines separaten Ablaufdiagramms verzichtet.

7.3.5. Globale Zustandsberichte

Die Konstruktion von globalen Zustandsberichten ist ein geradliniger Ablauf. Ein Anfragesteller trägt eine Nachricht mit dem Bedarf nach Zustandsinformationen an das Plattformmanagement heran, welches sodann alle Plattformen nebenläufig mit der Erstellung von Zustandsberichten beauftragt. Sobald alle Berichte eingegangen sind, kann der gebündelte Bericht an den Anfragesteller übergeben werden. Der Ablauf ist grafisch in Abbildung 7.8 beschrieben.

Für die Bereitstellung eines Simulationsfeeds kann ebenfalls der Nachrichtenbroker eingesetzt werden. Durch die stark erhöhte Belastung des Brokers in diesem Falle sollte bei konkreten Implementationen ein erweitertes Sharding des Brokers erwägt werden.

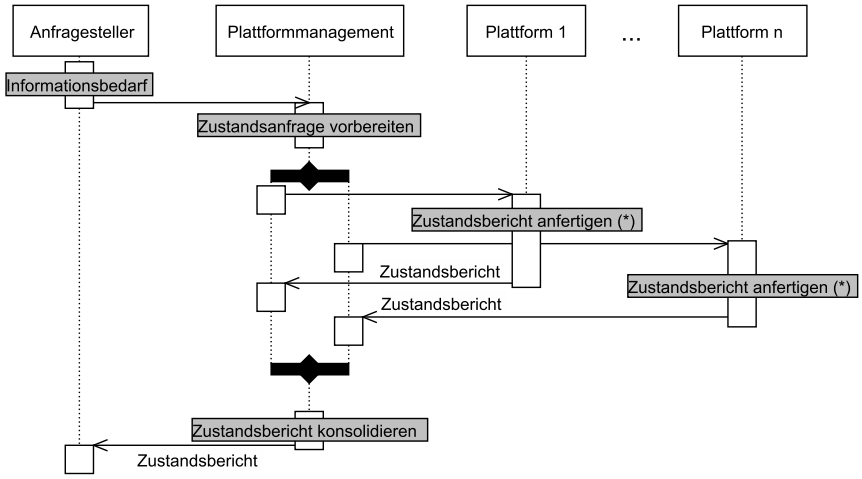

Abbildung 7.8.: Globaler Zustandsbericht aller Plattformen bereitgestellt durch das Plattformmanagement. (*) Siehe Abbildung 6.8.

7.4. Definition des Plattformmanagements als Referenznetz

Mit der Beschreibung aller beabsichtigten Interaktionen und den Funktionsumfängen kann nun die endgültige Konstruktion als Referenznetz vorgestellt werden. Sie ist in Abbildung 7.9 aufgeführt. Wie auch die Beschreibung der Plattform in Abbildung 6.10 verwendet die Darstellung das Konzept der partiellen Schreibkante, welche in Abschnitt 6.4.1 vorgestellt wurde.

Oben links und rechts finden sich agentenähnliche Transitionen zum Nachrichtenempfang und -versand. Diese sollten explizit dargestellt werden im Gegensatz zur vereinfachten Darstellung der MULAN-Plattform, um Nachrichten, welche an das Plattformmanagement selbst gerichtet sind, entsprechend behandeln zu können.

Im oberen Bereich ist in Grau die Abfrage des globalen Zustands modelliert. Durch Testkanten wird der Zustand aller Plattformen abgefragt. Die Transition »abfragen« synchronisiert sich somit mit dem Nachrichteneingang und -ausgang der jeweiligen Plattform.

In Dunkelgrün dargestellt finden sich links neben der Zustandsabfrage die Bestandteile zur proaktiven Skalierung. Durch die Transition »messen« können ähnlich zur Transition »abfragen« Zustandsinformationen abgefragt werden. Jedoch handelt es sich dabei, wie beschrieben, um oberflächliche Informationen, die gemessen werden können. Wird Bedarf erkannt, so wird eine entsprechende Beschreibung des Bedarfs in die weiße Stelle links im Netz abgelegt.

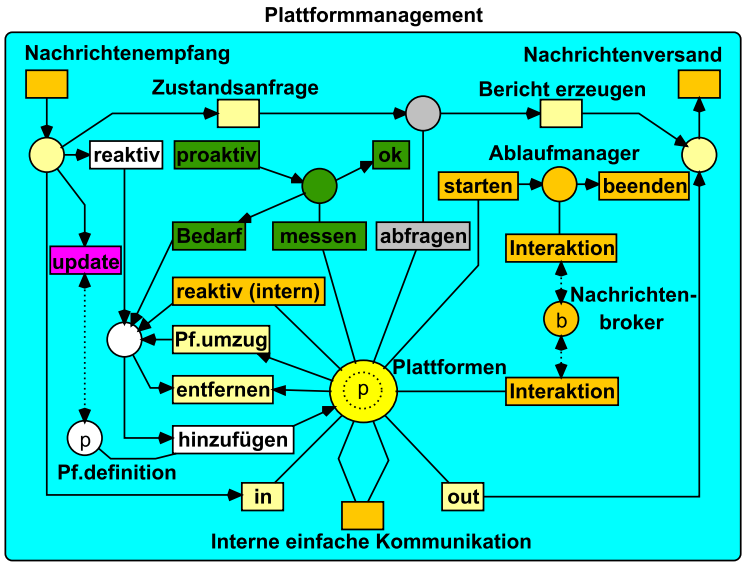

Abbildung 7.9.: Gesamtentwurf des Plattformmanagements.

Diese Stelle kann ebenfalls durch reaktive Skalierungsoperationen gefüllt werden. Dies geschieht, indem entweder eine Nachricht eingeht, welche von der weißen Transition »reaktiv« verarbeitet wird, oder aber von der orangen Transition »reaktiv (intern)«, welche eine entsprechende Anfrage eines Agenten weiterleitet. Eine letzte Möglichkeit für die Entstehung einer entsprechenden Marke ist durch die hellgelbe »Plattformumzug« Transition, welche abstrakt Plattformumzüge innerhalb des lokalen Plattformmanagements abbildet.

Die Marken können nun entweder von der hellgelben »entfernen« oder der weißen »hinzufügen« Transition verarbeitet werden, je nach gewünschter Änderung. Streng genommen legen die Transitionen des Vorbereichs des weißen Platzes (bis auf den Plattformumzug) eine der gewünschten Änderungsmenge entsprechende Menge an Marken im weißen Platz ab. Die Darstellung erfolgt jedoch der Übersichtlichkeit halber für den Spezialfall, dass stets eine Plattform hinzugefügt oder entfernt werden soll.

Für die Erzeugung von Plattformen wird die Plattformdefinition im entsprechenden Platz vorgehalten. Die Plattformdefinition kann durch eine entsprechende Nachricht und der magentafarbenen »update« Transition aktualisiert werden.

Im unteren Bereich unter den Plattformen findet sich eine orange Transition für die einfache Kommunikation, die ohne große Änderung aus der MULAN-Plattform übernommen werden konnte. Komplizierter ist hingegen die Realisierung der komplexen Kommunikation. Diese ist mit all ihren Komponenten in orange dargestellt.

Plattformen können für eine bestimmte Kommunikation im Namen eines Agenten einen entsprechenden Ablaufmanager starten, welcher nach den beschriebenen Kommunikationsprotokollen die Kommunikation durchführt. Dabei schreibt der Ablaufmanager in den entsprechenden Abschnitt des Nachrichtenbrokers. Plattformen können, sofern sie verfügbar sind, neue Nachrichten beim Nachrichtenbroker anfragen und beantworten. Der Umweg über Ablaufmanager und Nachrichtenbroker ermöglicht die Kommunikation auch mit temporär ausgefallenen Plattformen und verhindert, dass Plattformen starke Referenzen zueinander benötigen, da der Ablaufmanager die Adressierung übernehmen kann. Ist die Kommunikation abgeschlossen, kann der Ablaufmanager entfernt werden.

7.5. Zusammenfassung des Konzepts zum Plattformmanagement

Das Plattformmanagement bildet die oberste der detailliert betrachteten Ebenen der Architektur. Es wurde herausgearbeitet, dass ihre Hauptaufgaben die Erzeugung und Zerstörung von Plattformen umfasst. Diese Anforderung soll sowohl proaktiv als auch reaktiv auf innere und äußere Anfragen hin umgesetzt werden. Neben der Kernanforderung muss das Plattformmanagement aggregierte Zustands- und Fortschrittsinformationen, sowie Kommunikationsmedien für einfache und komplexe Kommunikation zwischen Plattformen und Agenten bereitstellen. Eine Lösung für plattformmanagementweite Identifikation ist ebenfalls erforderlich. Die Anforderungen wurden beschrieben und in Form von Ablaufdiagrammen und einem Referenznetzmodell festgehalten.

8. Die Mushu-Architektur

Abschließend wird die Konzeption der neuen Architektur, welche aus den Überlegungen und der MULAN-Architektur hervorgegangen ist, vorgestellt. Sie wurde in den vergangenen drei Kapiteln erarbeitet und wird hier übergreifend und zusammenfassend beschrieben. Die finale Architektur wurde im Rahmen der Arbeit auf den Namen MUSHU[1] getauft. MUSHU steht dabei als Akronym für »**Mu**ltiagentensystem mit **S**kalierbarkeit in **h**eterogenen **U**mgebungen« bzw. in der englischen Variante für »**Mu**lti-agent system with **S**calability in **h**eterogeneous **u**nderlying systems«.

Mit Abbildung 8.1 wird die zum Beginn der Arbeit gezeigte hierarchische Darstellung der MULAN- und ORGAN- Architektur erneut aufgegriffen. Ein Konzept für ein Plattformmanagement und eine damit kompatible Plattform wurde in den vergangenen Kapiteln ausführlich beschrieben. ORGAN behandelt auf der Ebene unter den Organisationen Multiagentensysteme. Der Begriff der Organisation bzw. Infrastruktur wird dort allgemeiner verstanden und das Plattformmanagement indirekt mit erfasst. Diese Inklusion sowie das Plattformmanagement selbst wird jedoch nicht detailliert in (WESTER-EBBINGHAUS, 2010) beschrieben.

[1] Mushu kann aus dem Japanischen sowohl als 無主 – Mushu – Besitzerlosigkeit, Herrenlosigkeit (passend zum Grundgedanken der Agententechnik) sowie auch als 無終 – Mushū – Endlosigkeit (passend zur Skalierung der Plattformen) übersetzt werden.

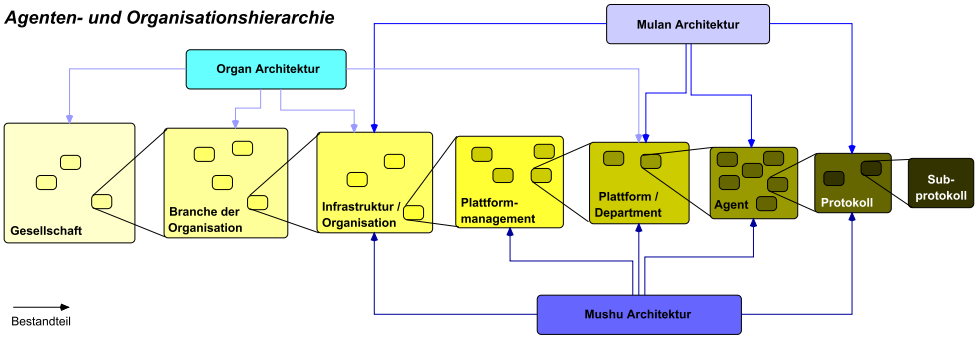

Abbildung 8.1.: Gesamthierarchie mit MULAN, ORGAN und MUSHU

8.1. Erweiterung der Mulan-Architektur zur Mushu-Architektur

Durch den hier dargestellten Aufbau wird ersichtlich, dass Mushu im Wesentlichen die Aspekte von Mulan abdeckt und darüber hinaus ein Plattformmanagement spezifiziert. Aus diesem Grund kann Mushu als eine Erweiterung der Mulan-Architektur um eine fünfte Ebene beschrieben und wahrgenommen werden. Abbildung 8.2 zeigt die zusammengefasste Gesamthierarchie. Die Abbildungen zur Plattform und zum Plattformmanagement entstammen dabei den Abbildungen 6.10 und 7.9, während die Darstellungen von Agent und Protokoll der Mulan-Architektur entnommen sind.

Bei der Betrachtung der vorgestellten Architektur fallen auf den ersten Blick einige strukturelle Ähnlichkeiten zwischen Plattform und Plattformmanagement auf. Dennoch sind die beiden Konzepte nicht austauschbar und auch nicht durch das jeweils andere abbildbar. Zu diesem Zweck diskutieren die folgenden Abschnitte die zentralen Unterscheidungsmerkmale zwischen dem neu eingeführten Plattformmanagement zu den jeweils nächsten Hierarchieelementen. Anschließend erfolgt eine Diskussion von Mushu im Kontext von Organ sowie eine Illustration der Einsatzmöglichkeiten der neuen Architektur.

8.2. Abgrenzung: Plattform und Plattformmanagement

Plattform und Plattformmanagement bieten ihren untergeordneten Einheiten unterschiedliche Formen des Umzugs an. Plattformen ermöglichen Agenten den Umzug zwischen Plattformen, während Plattformmanagements Plattformen im Allgemeinen nur lokal auf andere physikalische Orte umziehen lassen können. Ferner bieten Plattformen ihren Agenten statische Funktionalitäten, welche wiederum die Charakteristika der Plattform ausmachen, während sich Plattformmanagements primär durch die Gesamtheit ihrer (heterogenen) Plattformen definieren. Darüber hinaus bietet das Plattformmanagement den untergeordneten Einheiten (den Agenten) der Plattformen indirekt eine Möglichkeit an, Änderungen an der Plattform herbeizuführen. Dies ist Protokollen nur bedingt in Bezug auf Plattformen und Agenten möglich. Plattformmanagements besitzen darüber hinaus das Konzept eines Nachrichtenbrokers. Während dieses Konstrukt nicht undenkbar für Plattformen ist, wie bspw. die Implementation der Nachrichtenzustellung in Capa zeigt, weist der Nachrichtenbroker doch ein explizit passives Verhalten auf. Plattformen leiten eingehende Nachrichten im Wesentlichen aktiv an die Agenten weiter.

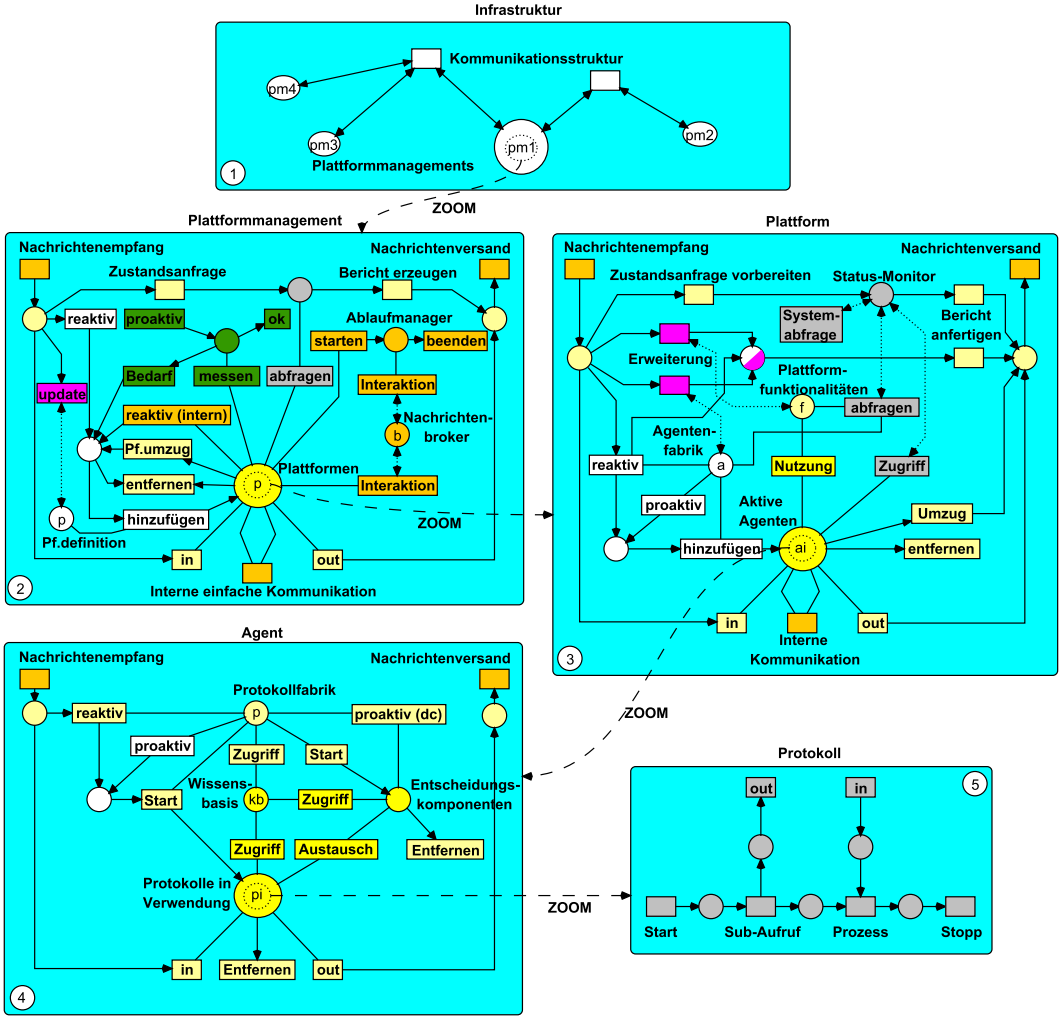

Abbildung 8.2.: Vorgeschlagene MUSHU-Architektur

Auch aus der Perspektive der intendierten Abbildung lassen sich Unterschiede ausmachen. Plattformen sind gestaltet, um Agenten einen Ort zu bieten, auf dem sie sich (lokal) austauschen können. Plattformen haben daher einen monolithischen oder zumindest modulithischen Charakter. Plattformmanagements hingegen bilden von Grund auf einen Zusammenschluss von mono- bzw. modulithischen Strukturen ab. Sie stellen spezielle Kommunikationswege bereit und sind föderierte Gebilde. Sie haben darüber hinaus nur einen schwach ausgeprägten individuellen Zustand, welcher sich überwiegend aus der Summe der ansässigen Plattformen ergibt, während Plattformen ein starkes individuelles Profil aufweisen können unabhängig von den beheimateten Agenten.

8.3. Abgrenzung: Infrastruktur und Plattformmanagement

Unter Betrachtung des klassischen Mulan-Modells erscheint es, als wenn das Plattformmanagement mit der Mulan-Infrastruktur konkurriert. Zunächst ist die Beschreibung der Infrastruktur im Mulan-Modell sehr allgemein gehalten. Zwischen Plattformen existieren dort lediglich einfache Kommunikationskanäle, welche Agentennachrichten weiterleiten können. Eine Erzeugung oder Zerstörung von Plattformen ist nicht vorgesehen, die zugrunde liegenden Strukturen (auf Infrastrukturebene) werden dort somit als statisch angesehen oder deren Dynamik zumindest nicht ausreichend beschrieben. Das Mushu-Plattformmanagement hingegen nimmt gezielt Einfluss auf den Verbund der kommunizierenden Plattformen. Es handelt sich dabei um die bzw. eine für die Konstruktion skalierbarer Systeme günstige Neugestaltung von Teilen der Mulan-Infrastruktur. Oberhalb des Plattformmanagements ist die Verbindung zu weiteren und anderen Plattformmanagements denkbar. Aus diesem Grund lehnt sich die Struktur der obersten Ebene von Mushu (die Mushu-Infrastruktur) stark an die oberste Ebene von Mulan an.

8.4. Einbettung von Mushu in Organ

Aus Sicht der Organ-Architektur kann auch argumentiert werden, dass ein entsprechendes Plattformmanagement ein fehlendes Konzept darstellt. In Organ entspricht die Mulan-Infrastruktur der Organisation und die Mulan-Plattform einem Department (Abteilung). Bei der Betrachtung von Unternehmen mittleren oder größeren Formats sind jedoch meist feinere Untergliederungen etabliert. So können generelle Bereiche existieren, wie die Personalabteilung, Controlling, Fulfillment, Operationen, Entwicklung, etc. Auch andere organisatorische Struk-

turen, welche keine Unternehmen sind, weisen ab bestimmten Größen ähnliche Aufbauten auf.

All diese Bereiche stellen jeweils einzelne Abteilungen (Plattformen) für Beschäftigte zur Verfügung, in denen intensive Kommunikation stattfindet. Darüber hinaus kann Kommunikation abteilungsübergreifend oder sogar bereichsübergreifend stattfinden. Bereiche können sehr unterschiedliche Strukturen in Hinblick auf ihre Arbeitsmaterialien, Aufgaben, handelnden Agenten, etc. aufweisen, die aber pro Abteilung (Plattform) weitestgehend einheitlich sind.

Darüber hinaus sind Bereiche häufig fest, da sie sich an den Anforderungen an die Organisation orientieren. Die interne Größe kann mit Aufgaben und Mitgliederzahl jedoch durchaus steigen. Werden Abteilungen zu groß, um mit Ressourcen wie Platz oder Zeit effizient arbeiten zu können, können sich Mitarbeitende (Agenten) an den Bereich (Plattformmanagement) wenden und die Erstellung einer neuen Abteilung (Plattform) erbitten.

8.5. Verwendbarkeit von Mushu

Das hier dargestellte Konzept der Mushu-Architektur kann eingesetzt werden, um damit als Referenzarchitektur eine verteilte und skalierende Agentensimulation zu konstruieren. Dies erweitert die bisherigen Möglichkeiten, da sowohl der Grundgedanke der echten Verteilung als auch der Skalierbarkeit bei der Konzeptualisierung explizit adressiert wurde. Physikalische Ausdehnung wird so bereits von der Architektur vorgesehen und muss nicht auf technischer Ebene nachträglich hinzugefügt werden. Dadurch entsteht ein stimmiges Gesamtkonzept, welches Softwareagenten in der technischen Realisierung Einflussnahme auf die physikalische Ausprägung und den Umfang der Gesamtsimulation ermöglicht.

Die primären Stärken von Simulationen auf der Basis der Mushu-Architektur sind wie eingehend beschrieben die genaue Abbildung der Nebenläufigkeit, Autonomie, Plattforminteraktion und Verteiltheit der realen Welt. Konkrete Anwendungsbeispiele für eine solche Simulation sind:

Kooperationsarbeit unter Beachtung von Ausfällen Mitarbeiter oder autonome Roboter kooperieren an Arbeitsplätzen zur Erfüllung einer gemeinsamen Aufgabe. Bei Ausfällen müssen schnell und dynamisch andere Entitäten einspringen, welche in den verschiedenen Bereichen (Plattformen) unterschiedliche Kompetenzen aufweisen. Durch eine Simulation können Verhalten unter verschiedenen Auswechselstrategien und inklusive der dynamischen Umstrukturierung von (gemeinsamen) Arbeitsplätzen beobachtet werden.

Informationsfluss in (sozialen) Netzwerken Durch die Erzeugung von Plattformen und der entsprechenden Kommunikation zwischen Teilnehmenden an Konversationen nehmen Informationen bestimmte Verläufe durch die Gesamtmenge der Teilnehmenden. Durch die Simulation mit Mushu können auch sowohl der Einfluss auf die Plattformen selbst sowie die Erzeugung neuer Plattformen modelliert werden. Dies kann als aktuelles Thema insbesondere für das Verstehen der Verbreitung von Falschinformationen von Interesse sein.

Marktanalsysen In wirtschaftlichen Anwendungen interagieren autonome Agenten auf (Handels)plattformen in der Form von Käufen und Verkäufen miteinander. Durch eine Simulation mittels Mushu kann der Aspekt beachtet werden, dass Plattformen durch Agenten geschaffen werden, wie etwa bei Privatverkäufen oder dem dynamischen Erzeugen neuer Handelsplätze mit verschiedenen Funktionalitäten und Assetklassen. Die Konstruktion einer aussagekräftigen Simulation für globale Märkte würde hierbei jedoch zusätzlich die Herausforderung der Modellierung der jeweiligen Agenten aufwerfen.

Auf der Basis der vorgestellten Architektur diskutiert der folgende Teil der Arbeit die Konstruktion einer Realisierung der Architektur.

8.6. Zusammenfassung - Mushu-Architektur

In diesem Kapitel wurde die Mushu-Architektur zusammenfassend dargestellt. Abbildung 8.2 zeigt das umfassende Modell der Architektur. Darüber hinaus wurde dargelegt, dass das Mushu-Plattformmanagement kein Spezialfall der Mulan-Plattform oder der Mushu-Plattform ist, sondern sich beide Konzepte strukturell unterscheiden. Der wesentliche konzeptuelle Unterschied besteht darin, dass Plattformen individuelle Ausprägungen und einen monolithischen Charakter aufweisen, wohingegen Plattformmanagements föderierte Einheiten sind. Im Organ Kontext stellen Plattformmanagement und Plattform die Konzepte Bereich und Abteilung dar, bei denen sich Bereiche aus der Summe ihrer Abteilungen ergeben. Die Mushu-Architektur kann als Referenzarchitektur für eine Simulation von Agentensystemen verwendet werden, in denen es Agenten möglich ist selbst die physikalische Ausdehnung des Systems zu bestimmen.

Teil III.

Realisierungskonzepte zur Mushu-Architektur

9. Technischer Ausgangspunkt

Die Beschreibung einer möglichen Realisierung der Bestandteile der Mushu-Architektur ist das erklärte Ziel dieses Teils der Arbeit. Dazu ist es nötig die Abstraktionsebene herabzusetzen und konkrete bestehende Technologien als Ausgangspunkt heranzuziehen. Um möglichst nah an der Referenzarchitektur bleiben zu können, empfiehlt es sich einen geeigneten Simulator für Referenznetze als Basis zu verwenden. Der in Abschnitt 2.8 vorgestellte Simulator Renew ist sowohl der leistungsstärkste existierende Simulator für Referenznetze als auch die Basis bisheriger Implementationen im Rahmen des generellen Themas der Simulation von Agentensystemen mit Referenznetzen. Auf Renew bauen beispielsweise die Mulan-Umsetzung aus (Rölke, 2004), Capa aus (Duvigneau, 2002), Paffins aus (Wagner, 2018) und einige weitere auf.

Aus diesem Grund eignet sich Renew hervorragend als Ausgangspunkt für eine Implementation von Mushu. Bei der Betrachtung der anderen Realisierungen stellt sich die Frage, ob ein direktes Aufsetzen auf beispielsweise Mulan nicht passender wäre, da dort viele agentenorientierte Grundlagen bereits vollständig implementiert vorliegen. Jedoch ist der Fokus dieser Arbeit deutlich auf die Interaktion von Plattformen und der Konstruktion eines geeigneten Plattformmanagements bezogen. Aus diesem Grund ist es nicht unmittelbar nötig, auf die Strukturen von Mulan zurückzugreifen, sondern die Funktionalitäten sollten sich *in allgemeinerer Form* direkt in den zugrunde liegenden Simulator Renew integrieren lassen. Auf diese Weise können die Verteilungs- und Skalierungsfunktionen in anschließenden Arbeiten für Konzepte eingesetzt werden, welche über den Fokus auf Agentenorientiertheit hinausgehen, während sie dennoch für diese zur Verfügung stehen.

Im Folgenden werden somit bestehende Anknüpfungspunkte aus dem Umfeld des Renew Simulators aufgearbeitet. Das Kaptiel stellt somit eine Bestandsaufnahme der existierenden Realisierungen dar, welche im Kontext der Mushu-Architektur eingesetzt werden können.

9.1. Kommunikation: Einsatz von bestehenden Verteilungslösungen

Die Realisierung eines Plattformmanagements und der darin laufenden Plattformen könnte in der konkreten Ausgestaltung selbstverständlich innerhalb einer einzelnen Anwendung implementiert werden. Dabei würde es sich jedoch lediglich um eine besondere Konstruktion eines Agentensystems handeln, da wie beschrieben die Plattform als Agent behandelt werden kann. Folglich würde die beabsichtigte Neuerung, welche im Rahmen dieser Arbeit erarbeitet wird, in solch einer Implementation ihr Potenzial nicht annähernd ausschöpfen.

Aus diesem Grund und auch, um eine sinnvolle Diskussion der Forschungsfragen zu ermöglichen, wird für die Realisierung bei der Kommunikation zwischen Plattformen ein echtes verteiltes System angenommen. Dabei umfasst jede Plattform genau einen Renew Simulator. Eine Implementation innerhalb einer einzigen Anwendung wäre unter Umständen zu anschaulichen Zwecken denkbar, hätte jedoch darüber hinaus keinen großen Nutzen und wird daher nicht innerhalb der Realisierungskonzepte und Prototypen im Rahmen der Arbeit verfolgt.

Daraus ergibt sich jedoch für die Implementation direkt die Notwendigkeit, entweder eine eigene Verteilungslösung für Referenznetze zu konstruieren oder aber auf bestehenden Verteilungslösungen aufzusetzen. Während es möglich wäre, eine dedizierte Verteilungslösung für den hier vorgeschlagenen Anwendungsfall zu konstruieren, ist dieses Unterfangen jedoch äußerst komplex. Da der Fokus der Arbeit jedoch nicht auf der Verteilung selbst, sondern auf der Konzeption einer Architektur für skalierende Simulationen von Agenten (auf Referenznetzbasis) liegt, sollte auf eine der bestehenden Lösung zurückgegriffen werden. Dennoch sollen Ideen und Ansätze in diese Richtung zum Abschluss der Arbeit in Kapitel 17 im Ausblick diskutiert werden.

Die Drop-Engine (S. Bendoukha, 2017) stellt im wesentlichen Mittel zur Modellierung von Inter-Cloud-Anwendungen bereit. Dabei werden Ausführungen in der Cloud jedoch als isolierte Operationen behandelt und die dortigen Simulation nicht als Teil der Gesamtsimulation verstanden. Dieser Zusammenhang ist allerdings genau die gewünschte Eigenschaft der zu konstruieren Architektur. Aus diesem Grund ist die Drop-Engine nur bedingt geeignet, um als Basis für die Verteilung infrage zu kommen.

Renew Remote (Kummer, Wienberg, Duvigneau, Köhler u. a., 2003) bietet eine Schnittstelle, um entfernte Renew-Simulatoren zu steuern. Während dies in späteren Abschnitten der Arbeit eine interessante Vorarbeit darstellt und daher auch noch einmal aufgegriffen werden wird, liefert Renew Remote keine Verteilung von Simulationen. Daher ist es als Verteilungsgrundlage ungeeignet.

Das Distribute Plugin (M. Simon und Moldt, 2016) beschränkt sich auf die verteilte Ausführung von Referenznetzen. Es kann genutzt werden, um eine Referenznetzsimulation über die Grenzen von physikalischen Maschinen hinweg ausführen zu können. Es ist überwiegend technisch motiviert, liefert dabei jedoch genau die gewünschte Grundlage einer integrierten Gesamtsimulation.

Darüber hinaus bildet des Distribute Plugin mit seiner Implementation genau eine grundlegende komplexe Kommunikation mit je einmaligem Datenaustausch in beide Richtungen ab. Dies entspricht im Wesentlichen dem Protokoll-Beispiel aus Abbildung 5.7.

Aus diesem Grund wird im Folgenden vorausgesetzt, dass bei maschinenübergreifender Kommunikation zwischen Agenten auf Plattformen das Distribute Plugin zum Einsatz kommt.

Bei der Kommunikation zwischen Plattformen ist unter Umständen keine komplexe Kommunikation notwendig. Die Anforderungen an geeignete Kommunikationsprotokolle werden in den folgenden Kapiteln genauer beschrieben.

Komplexe Kommunikation

Die komplexe Kommunikation, wie sie im konzeptuellen Teil der Arbeit beschrieben wird, fügt im Wesentlichen einen Ablaufmanager und einen Nachrichtenbroker mit den zugehörigen Protokollen hinzu. Während eine Art Ablaufmanager so im Renew Kontext noch nicht beschrieben wurde, existiert zumindest für den Einsatz von Nachrichtenbrokern die Arbeit (Schmidt, 2016) zum Einsatz von ActiveMQ in Capa. Bei ActiveMQ handelt es sich um eine einfache Implementation eines Nachrichtenbrokers.

Bei der damaligen Arbeit handelt es sich um eine Bachelorarbeit mit entsprechend eingegrenzten Umfang. Die Untersuchungen sind sehr spezifisch für die technische Implementation ActiveMQ und zeigten damals Schwierigkeiten bei der Integration mit Capa auf. Eine der Hauptproblematiken bestand aus heutiger Sicht darin, dass der konzeptuelle Teil der infrastrukturellen Aspekte der Agentenkommunikation in der Form und Detailtiefe wie in der Beschreibung durch Mushu noch nicht existierte.

Die gezielte Ablaufsteuerung, Entkopplung, Verortung auf der Ebene des Plattformmanagements und eventual consistency waren damals noch nicht Teil der Untersuchungen.

9.2. Plattform: Bisherige Umsetzung von Cloud-Nativity in Renew

Einige der vorgeschlagenen Lösungsansätze sind zu Teilen oder in anderer Form bereits in Renew und zugehörigen Plugins implementiert. Die Verwendbarkeit soll daher an dieser Stelle diskutiert werden, sowie die Anforderungen an Erweiterungen. Die Informationen sind dabei hilfreich, um in diesem Teil der Arbeit die konzeptuellen Vorüberlegungen für die Konstruktion von Prototypen und Proof-of-Concept Implementationen darzulegen.

Grundlegend sind Agenten im Renew-Kontext letztendlich Netzinstanzen. Aus diesem Grund bilden Netze bzw. Netzdefinitionen in Renew die in Mushu modellierten Agentendefinitionen ab. Plattformfunktionalitäten hingegen bieten statischen Code an und definieren die Ausprägung der Plattform. Die Funktionalitäten von Renew-Plugins stehen allen lokalen Netzinstanzen und damit allen lokalen Agenten zur Verfügung. Folglich kann das Konzept der Plattformfunktionalitäten durch Renew-Plugins abgedeckt werden. Weitere Details hierzu werden im folgenden Unterabschnitt zu Agilität erläutert.

An dieser Stelle kann es hilfreich sein, eine bestimmte Sicht auf die zu konstruierende Plattform einzunehmen. Eine Renew-Instanz soll nach dem Hinzuschalten bereitstehen, um Agenten zu simulieren und Operationen durchzuführen. Eine laufende lokale Simulation kann dabei ein Teil einer größeren Simulationsumgebung werden. Renew liegt damit auf Ebene des Plattformmanagements als eine Art nutzbarer *Service* vor. Analog zur Bereitstellung von Infrastruktur, Plattformen (allgemein, an dieser Stelle nicht bezogen auf die Mushu-Architektur) oder Software als Service und der zugehörigen Leitmetaphern »Infrastructure-as-a-Service (IaaS)«, »Platform-as-a-Service (PaaS)« und »Software-as-a-Service (SaaS)« soll die Leitmetapher zur Realisierung der Plattform lauten:

Renew-as-a-Service

9.2.1. Beobachtbarkeit

Im Simulator Renew ist der Ansatz der Wiedergabe von Statusinformationen durch das Plugin Renew *Remote* für eine einzelne Simulation umgesetzt. Als bestehende Lösung kann dabei zunächst die direkte Überwachung der Simulation mittels Ankopplung einer entfernten grafischen Oberfläche betrachtet werden. Im Falle von Renew *Remote* wird hierzu Java RMI eingesetzt und es ist ebenfalls ein weiterer Renew Simulator erforderlich. Das Plugin wurde jedoch nie unter dem Gesichtspunkt entwickelt, dass damit eine verteilte Simulation innerhalb einer Cloud Umgebung betrieben werden soll. Java RMI ist ein nicht-universelles,

nicht zustandsloses Kommunikationsprotokoll. Darüber hinaus ist die Beobachtung von nur einem Simulator möglich und auf den Ablauf der dortigen Simulation optimiert. Weiterführende Informationen zum Simulator und der Umgebung sind dabei nicht verfügbar.

Bei der Umsetzung eines Simulationsfeeds ist zunächst das *Logging* Plugin zu erwähnen. Es bietet ein System zum Aufzeichnen von Simulationsevents im Rahmen der Java-Simulation. Konsumierte und produzierte Marken der Schaltvorgänge werden aufgezeichnet und dabei String-Repräsentationen der Objekte verwendet. Interessant sind hierbei insbesondere die Eingriffe in die laufende Simulation, welche für einen allgemeineren Simulationsfeed ebenfalls notwendig sind. Das Logging-Plugin baut auf der String-Repräsentation und einfachem Speichern der Differenzen – vor und nach Transitionsschaltungen – auf. Die Einstellungen für das Logging sind durch den Einsatz des Logging-Frameworks Log4j vielfältig, jedoch ist eine Anbindung an ein Webinterface nicht vorhanden. Das Logging-Plugin hat keine konkreten Anforderungen bezüglich der eingesetzten Petrinetzsemantik. Jedoch sollte erwähnt sein, dass innerhalb des Logging-Plugins das Feuern einer Transition als *Step* bezeichnet wird, jedoch damit nicht ein *Step* im Sinne der Step-Semantik (vgl. Abschnitt 2.2.5) gemeint ist. In der Arbeit (Krön, 2021) wurde die Erweiterung des Logging-Plugins um eine Funktionalität diskutiert, welche Logs als Kausalnetze[1] abbildet. Durch den Einsatz eines Kausalnetzes kann der exakte Ablauf der Schaltung rekonstruiert werden. Die Funktionalität ist bisher leider jedoch nur für Interleaving-Semantik[2] verfügbar und nicht für True-Concurrency.

Für die Variante basierend auf Simulationspersistenz liefert (Jacob, 2001) einen Ansatz, jedoch ist die Umsetzung ebenfalls auf Interleaving-Semantik beschränkt. Dabei wird nach jedem Schritt der Gesamtzustand des Netzes in eine Datenbank persistiert. Objekte werden als serialisierte Java-Binärdatenobjekte gespeichert. Eine Kombination mit einem Web-Interface ist ebenfalls nicht vorhanden. Ein großer Nachteil der Methode besteht jedoch darin, dass für die Verwendung ebenfalls eine sequenzielle Simulation durch Interleaving-Semantik erforderlich ist. Somit kann keine weitere Nebenläufigkeit innerhalb der lokalen Simulation genutzt werden.

Für die anderen geforderten Beobachtbarkeits-Funktionalitäten an die Plattform existieren so keine bekannten Umsetzungen im Kontext der Referenznetzsimulation.

[1] Ein Kausalnetz erlaubt keine Verzweigung hinter Plätzen und besitzt daher keinen Nichtdeterminismus.

[2] Als Auffrischung bezeichnet die Interleaving-Semantik die sequentielle Ausführung von einzelnen Transitionsfeuerungen. Für weitere Details siehe die Einführung der Petrinetzsemantiken in Abschnitt 2.2.5

9.2.2. Operabilität

Bezogen auf die Operabilität ist in erster Linie erneut das Renew *Remote* Plugin zu nennen, welches das Steuern einer entfernten Simulation zulässt. Dabei setzt Renew *Remote* wie bereits beschrieben auf Java RMI, welches weder zustandslos noch universell ist. Dennoch war der hier intendierte Einsatzzweck niemals innerhalb der geplanten Funktionalität von Renew *Remote*. Renew *Remote* besitzt jedoch keine Funktionalitäten für den Upload von Netzdefinitionen oder Plugins, welche für das Nachliefern von Agentendefinitionen oder Plattformfunktionalität notwendig wäre.

Darüber hinaus könnte noch das Distribute Plugin als mögliche Basis der Umsetzung der hier geforderten Funktionalitäten herangezogen werden, da es die Steuerung entfernter Referenznetze ermöglicht. Es wäre denkbar, damit ein verteiltes Netz zu konstruieren, welches auf den entfernten Knoten die Operationen durch ein spezialisiertes Plugin ausführt. Neben der Tatsache, dass das Distribute Plugin niemals für diesen Einsatzzweck konzipiert wurde, würde es ebenfalls einen lokalen Renew Simulator inklusive laufendem Distribute Plugin und einer Distribute Registry erfordern. Die Entwicklung der Schnittstellen wäre darüber hinaus vergleichsweise kompliziert für einen an sich unkomplizierten Anwendungsfall und es könnte nicht auf weitere bestehende Lösungen außerhalb des Renew Ökosystems in direkter Art und Weise zurückgegriffen werden.

9.2.3. Agilität

Die Agilität ist in einigen Teilen bereits umgesetzt, so setzt Renew als Programmiersprache Java und damit als Laufzeitumgebung die Java Virtual Machine (JVM) ein. Die JVM ist dabei genau eine Art Adapter-Layer, wie er in der Anforderung gefordert wurde. Dennoch muss das System mit einer entsprechenden JVM ausgestattet sein, damit Renew lauffähig ist. Zusätzlich kann es unter Umständen zu Problemen zwischen den Java Versionen kommen, falls andere (Java-)Applikationen auf dem System Inkompatibilitäten aufweisen.

Heterogenität ist in jedem Fall im Rahmen einzelner (Mushu-)Plattformen gegeben, indem Renew mit bestimmter Funktionalität konfiguriert werden kann. Erste Vorarbeiten dazu liefert (Schleinzer, 2007) im Rahmen des hierarchischen Capa, bei dem Agenten ineinander geschachtelt auftreten können. Die Arbeit (Duvigneau, 2009) baut diese dann zu einem vollwertigen Plugin-System aus. Durch die Konfiguration der zu ladenden Plugins kann Renew wie eingehend erläutert sehr individuell konfiguriert werden.

Eine spannende Erweiterung in diesem Kontext ist die aktuelle Arbeit (Janneck, 2021), in der das Plugin-System auf ein Modulsystem erweitert wird. Die Ein-

führung von Modulen in Renew wurde bereits in Abschnitt 2.8.9 eingeführt. Dadurch wird es möglich Funktionalität zur Laufzeit als eigenständige Entitäten nachzuladen, eine sehr wichtige Vorarbeit, um die Anforderung Plattformfunktionalität dynamisch nachzuladen zu erfüllen.

Die allgemeine Entwicklung und Weiterentwicklung von Renew und Mulan orientiert sich am Paose Ansatz und setzt auf inkrementelle Entwicklung. Darüber hinaus wurde die Arbeitsorganisation auf das Scrum@Scale Modell[3] umgestellt. Somit folgt die Entwicklung agilen Entwicklungsmodalitäten.

Auf technischer Ebene unterstützt ebenfalls der Einsatz von Gitlab mit der integrierten Continuous Integration (CI) Umgebung. Die Umsetzung einer entsprechenden Pipeline im Rahmen von Gitlab wird in (Feldmann, 2019) diskutiert. Darüber hinaus adressiert die angesprochene Arbeit Inhalte bezüglich der Realisierung von Plattformportabilität, welche auch als Kooperationsarbeit (Röwekamp, Moldt und Feldmann, 2018) veröffentlicht wurde und später im Rahmen dieser Arbeit in Kapitel 10 weiter ausgeführt wird.

9.2.4. Abhärtung

Um die Aspekte bezüglich der Abhärtung der Software in den bereits bestehenden Komponenten von Renew zu betrachten, muss die Betrachtung auf die umgesetzten Mittel zur Verteilung gelenkt werden. Die Implementation von Mulan und Capa sieht nicht die Existenz eines verteilten Systems aus mehreren Plattformen vor. Daher existieren an dieser Stelle keine Umsetzungen von Abhärtung im Kontext von Mulan und Capa.

Das Distribute Plugin basiert auf Java RMI und benötigt daher eine zentrale Registry, in der alle entfernten Netzinstanzen referenziert werden. Problematisch ist jedoch beim Distribute Plugin, dass nicht zwischen Plattformverwaltung und Plattform unterschieden wird. Praktisch äußert sich dies darin, dass kein Unterschied zwischen Simulator und Simulation bezüglich Identifikationen und der Handhabung besteht. Somit verbinden sich neue Plattformen direkt beim Start mit der zentralen Registry anstatt erst, wenn sie einer bestimmten Simulation zugeordnet werden sollen. Sollte es zu Kommunikationsproblemen mit der Registry kommen oder diese neu aufgesetzt werden, so stürzt die gesamte globale Simulation ab: Es kommt zum kaskadierenden Fehlschlag.

Da ebenfalls keine Adressierung für Simulator und Simulation separat existiert, kann es auch vorkommen, dass Netzinstanzreferenzen aus vorherigen Durchläufen immer noch vorliegen. Sobald diese angesprochen werden sollen, kommt es zu Fehlern, da die Objekte selbstverständlich nicht mehr existieren.

[3] Siehe https://www.scrumatscale.com/ - Zuletzt abgerufen im April 2021

Die hier angesprochenen Probleme liegen zum Teil nicht in den Kernanforderungen, da sie aus dem Modell der Cloud-Nativity resultieren und nicht aus der vorgeschlagenen Mushu-Architektur. Da sie für die Konstruktion eines Webservice-basierten verteilten Simulators dennoch wichtig sind, wurden sie ebenfalls noch einmal aufgegriffen. Genauere Unterscheidungen der Anforderungen wurden in Abschnitt 6.1 dargelegt.

9.3. Plattformmanagement

Bezüglich der Eigenschaften des Plattformmanagements existieren sehr wenig Vorarbeiten. Die wesentliche Zahl der Publikationen beschreiben das System nur bis zur Plattform und lassen darüber liegende Strukturen abstrakt.

Auch bei den Cloud-basierten Ausführungen in (S. Bendoukha, 2017) wird die Infrastruktur zwar Cloud-basiert beschrieben, der konzeptuelle Ansatz weicht aber nicht wesentlich von den einzelnen manuellen Maschinen ab, die auch das Distribute Plugin adressiert. Das dynamische Hinzufügen von Plattformen wird nicht diskutiert.

Aus diesem Grund muss praktisch das gesamte Plattformmanagement von Grund auf konstruiert werden. Dazu sei noch ein weiteres Mal darauf hingewiesen, dass das »Plattformmanagement« keine Softwarekomponente an einer Stelle im verteilten System sein wird. Es ist viel mehr der gesamte logische Zusammenschluss verschiedener Systeme, welche insgesamt durch ihr Zusammenspiel die durch das Mushu-Plattformmanagement beschriebene Funktionalität realisieren. Auf technischer Ebene ist ein geeigneter Term für diesen Zusammenschluss »Cluster«.

9.3.1. Skalierbare Ausführung des Distribute Plugins

Die Simulationsmöglichkeiten mit dem Distribute Plugin sind insofern limitiert, als es das nicht mehr zeitgemäße Java RMI als zentrales Element einsetzt und eine zentrale Registry erfordert. Im Distribute Plugin wird die Kommunikation mit je einer Informationsübertragung dadurch erreicht, dass vom initiierenden Downlink eine Kommunikation zum aufgerufenen Uplink und zurück stattfindet anstatt der vollen Unifikation. Um diesem Umstand Rechnung zu tragen, müssen Variablen, welche eine lokale Bindung vor dem Absenden erhalten sollen, syntaktisch als solche kenntlich gemacht werden. Details zu dieser Syntax finden sich in Tabelle 2.1 im zugehörigen Abschnitt 2.8.8 im Grundlagenkapitel. Damit bildet das Distribute Plugin eine konventionelle Erweiterung des Referenznetzformalismus, wie er in Renew umgesetzt ist.

Während durch das Distribute Plugin die grundlegende Verteilung einer Simulation ermöglicht wird, geht die Funktionalität jedoch nicht über eben diese Eigenschaft hinaus. Es gibt keine Lösungen für das kontrollierte Hinzuschalten weiterer Instanzen in die Simulation. Darüber hinaus existiert keine Vorbereitung der umgebungsabhängigen Parameter für den Start einer weiteren Simulationsinstanz. Diese Eigenschaften sind nicht als Mangel des Distribute Plugins zu verstehen, sondern lagen eindeutig nicht im Umfang der damaligen Arbeiten, da für ihre Umsetzung erst durch MUSHU eine fundierte Grundlage bzw. Begründung besteht.

Daraus (und aus den Ausführungen in Kapitel 7.1.1) ergibt sich unter anderem die Notwendigkeit eines Management-Systems, welches das kontrollierte Hoch- und Herunterfahren weiterer Instanzen ermöglicht. Das Management-System sollte in der Lage sein, Einfluss auf die Infrastruktur zu nehmen und von dieser so weit wie möglich zu abstrahieren. Aus Sicht der MUSHU-Architektur entspricht das Management-System den Anteilen, welche das reaktive und proaktive Erzeugen und Vernichten von Plattformen modellieren.

Die Einführung einer neuen Systemkomponente will jedoch zu jeder Zeit wohlüberlegt sein. In jedem Fall sollte die Funktionsmächtigkeit eines derartigen Management-Systems auf die wesentlichen von der MUSHU-Architektur geforderten Eigenschaften beschränkt sein und von schlanker und flexibler Natur sein. Nur auf diese Weise kann gewährleistet werden, dass ein zusätzliches Management-System die Skalierung letztendlich nicht selbst behindert und das ausführende System nicht unnötig starker Zusatzbelastung aussetzt. Im Kontext von skalierbaren Architekturen sind zustandslose Systeme in der Regel tendenziell einfacher zu handhaben. Aus diesem Grund sollte zumindest geprüft werden, ob Zustandslosigkeit als Anwendungsarchitektur eingesetzt werden kann.

Abschließend ist der letzte relevant zu betrachtende Punkt der Gesamtaufwand der Implementation. Da Implementationsarbeit im Normalfall nur eine handwerkliche Tätigkeit beinhaltet, die wenig bis keine konzeptuelle Neuerung bietet, ist es im Interesse der Arbeit die Konzepte mit geringem Implementationsaufwand umsetzen zu können. Dabei ist dieser Aspekt nicht maßgeblich, sollte aber zumindest in kleinen Teilen in die Bewertung der einzelnen Umsetzungsmöglichkeiten einfließen, um Lösungen mit sehr ungünstigem (Implementations)kosten/Nutzen-Verhältnis auszuschließen.

9.3.2. Technische Anforderungen

Speziell im Kontext der Skalierung von (MUSHU-)Plattformen muss ein vollständig neues System konstruiert werden. Aus diesem Grund sollen genauere technische Rahmenbedingungen für diesen Teil der Arbeit eingeführt werden, um eine entsprechende Umsetzung konzipieren zu können.

Aus den Ausführungen des letzten Abschnitts ergeben sich zusammengefasst die folgenden (technischen) Anforderungen an ein entsprechendes Management-System zur Skalierung von Plattformen und an die zugehörigen Komponenten:

- Das Management-System und seine Komponenten sollen der Simulation selbst die Kontrolle über die Skalierung der Simulation ermöglichen.
- Das Management-System soll sich in das Distribute Plugin, wenigstens jedoch in einen bestehenden Verteilungsansatz für Referenznetzsimulationen, eingliedern.
- Das Management-System muss mit den Startparametern und Umständen des Renew Distribute Plugins bzw. der Verteilungslösung umgehen können.
- Eventuell technisch notwendige Anpassungen am Referenznetzsimulator sollten nach Möglichkeit gering gehalten werden.
- Das Management-System soll schlank, flexibel und aufs Wesentliche beschränkt sein und keinen extrem großen handwerklichen Implementationsaufwand besitzen.
- Die Umsetzung von Zustandslosigkeit ist wünschenswert, sofern dies möglich ist.

Diese Anforderungen werden die Eckpunkte der laufenden Evaluation verschiedener Ansätze zur Konstruktion der Realisierung von Management-System und den dazugehörigen Komponenten in Kapitel 10 bilden.

9.4. Aufbau der Konzepte zur Realisierung

In Kapitel 4 wurde in Abbildung 4.4 dargestellt, dass die Realisierung der einzelnen Komponenten in umgekehrter Reihenfolge ihrer Konzeptualisierung erfolgen soll.

Aus diesem Grund wird Kapitel 10 eine Umsetzung des Plattformmanagements diskutieren, Kapitel 11 die Umsetzung der Plattform und Kapitel 12 die Umsetzung (komplexer) Kommunikation. Abschließend beschreibt Kapitel 13 Verbesserungen auf der Ebene der Mustererkennung, welche hier anwendungsdomänenseitig auf der Ebene der Subprotokolle von Mushu vorliegt.

9.5. Zusammenfassung - Technischer Ausgangspunkt

In diesem Kapitel wurden die Ausgangsbedingungen für eine Realisierung von Mushu auf der Basis vom Referenznetzsimulator Renew diskutiert. Für die Realisierung von Agentenkommunikation stehen die Verteilungsmethoden Renew Remote, die Drop-Engine sowie das Distribute Plugin zur Verfügung, von denen das Distribute Plugin die beste Ausgangsposition bildet. Darüber hinaus existieren Vorarbeiten im Bereich Capa und ActiveMQ, welche zumindest Gedankenanstöße für die Realisierung komplexer Kommunikation geben können. Auf Ebene der Plattform existieren durch das Renew Remote Plugin und durch das Distribute Plugin eingegrenzte Funktionalitäten bezüglich des Cloud-Native-Ansatzes, welcher innerhalb der Plattform verfolgt werden soll. Zu diesem Zweck soll bei der Konstruktion der Plattform die Leitmetapher *»Renew-as-a-Service«* eingenommen werden. Heterogenität kann durch das hierarchische Capa und die Plugin-Architektur von Renew konstruiert werden. Für das Plattformmanagement existieren bisher keine bestehenden Lösungen, auf denen aufgesetzt werden könnte. Aus diesem Grund wurden im Rahmen des Kapitels eine Reihe an technischen Anforderungen an ein Plattformmanagement gestellt, welche eine neue Implementation umsetzen sollte.

10. Realisierungskonzept zum Plattformmanagement

Dieses Kapitel adressiert die im Rahmen der Arbeit als Beitrag geschaffene Ausführungsumgebung für skalierbare Referenznetzsimulationen. Wie bereits in Kapitel 9 motiviert, ist das Fehlen einer solchen Umgebung eines der größten Probleme beim dynamischen Erzeugen von Plattformen. Wie im Kapitel zum Stand der Forschung in Abschnitt 3.4.3 beschrieben, sind proaktive Skalierungsmöglichkeiten bereits für diesen Kontext ausreichend bekannt und verfügbar in der Form von Autoscalern. Aus diesem Grund wird in diesem Kapitel vor allem die reaktive Skalierung von Plattformen behandelt. Aufbauend auf den Ausführungen in Kapitel 7 wird als grundlegende Architektur der Skalierungsfunktion der Einsatz einer Management-Komponente angestrebt, welche die Interaktion mit der Infrastrukturumgebung kapselt.

Während generell eine mögliche Umsetzung des Mushu-Plattformmanagements beschrieben wird, wird innerhalb des Kapitels auch immer wieder die Perspektive des Modellierers eingenommen, welcher mit dem hier konstruierten Rahmenwerk eine Simulation definiert. Diese Perspektive ermöglicht es, das Rahmenwerk zugänglich und anwenderorientiert zu skizzieren. Die Umsetzung der Plattformmanagement-Komponente von Mushu kann möglicherweise nicht direkt als Referenznetz konstruiert werden, da es sich selbst auf technischer Ebene um ein verteiltes System zur Simulation von Referenznetzen handelt. An vielen Stellen wird wie bereits in Kapitel 4 angekündigt, die Sicht der Webservices auf das System eingenommen. webserviceorientierte Lösungen werden aber dennoch anderen Lösungen wie referenznetzbasierten gegenübergestellt, um die bestmögliche Lösung zu finden. Für die konkreten Ausgestaltungen der möglichen Aktionen sind die einzelnen Transitionen des Plattformmanagementanteils der Mushu-Architektur jedoch sehr grobgranular, sodass dazu diverse Komponenten im Rahmen des Kapitels weiter ausspezifiziert werden.

Da es sich bei der Erzeugung und der Vernichtung von Plattformen um die Kernfunktionalität des Plattformmanagements handelt, ist diesem Aspekt der größte Teil des Kapitels gewidmet. Das Kapitel untergliedert sich zunächst in grundlegende und allgemeine Realisierungsmöglichkeiten in Abschnitt 10.1. Allgemeine Architekturentscheidungen der Management-Komponente werden in Abschnitt 10.2 behandelt und die allgemeinen Architekturentscheidungen der Schnittstelle zur

Management-Komponente auf der Simulationsseite in Abschnitt 10.3. Darauf aufbauend erörtert Abschnitt 10.4 die grundlegenden Ansätze für Kommunikationsprotokolle für die Integration beider Komponenten. Abschnitt 10.5 diskutiert einige Besonderheiten bei der Konstruktion eines zustandslosen Systems und diskutiert die Anbindung externer Clustermanager, da diese unter Umständen notwendige Funktionalität bereitstellen können. Als weiterer Aspekt wird die Form des Deployments zum gezielten Einsatz in einem skalierbaren System der entwickelten Komponenten in Abschnitt 10.6 erörtert. Eine der Anforderungen der Agentenkommunikation umfasste die Bereitstellung einer Kommunikationsinfrastruktur. Diese Anforderung wird in Kapitel 12 behandelt. Schlussendlich werden die noch verbleibenden Aspekte der Architektur des Mushu-Plattformmanagements in Abschnitt 10.8 behandelt.

Die hier erörterten Realisierungskonzepte bilden einen der zentralen Beiträge im Rahmen der Arbeit und wurden auf Konferenzen und Tagungen vorgestellt und mit dem Fachpublikum diskutiert. Die zugehörigen Veröffentlichungen des Autors umfassen die Beiträge (Moldt, Röwekamp und M. Simon, 2017), (Röwekamp, Moldt und Feldmann, 2018) und (Röwekamp und Moldt, 2019). Entnommene Aussagen und Erkenntnisse sind als solche kenntlich gemacht.

Zu den konzeptuellen Ausführungen sowohl dieses Kapitels als auch den beiden folgenden Kapiteln schließt später die Beschreibung prototypischer Implementationen der getroffenen Entscheidungen in Kapitel 14 an.

10.1. Umsetzung der Komponenten des Plattformmanagements

Bevor die eigentliche Umsetzung des Plattformmanagements betrachtet werden kann, gilt es zu beschreiben, welche Komponenten der konzeptuellen Ausführungen in Kapitel 7 konstruiert werden müssen. Das Mushu-Plattformmanagements ist eine logische Entität und umfasst mehrere einzelne Bestandteile. Daher liegt es nahe, die einzelnen Aspekte des Mushu-Plattformmanagements separat zu betrachten. Einzig der Plattformumzug soll nicht gesondert betrachtet werden, da im konzeptuellen Kapitel in Abschnitt 7.3.4 bereits motiviert wurde, dass er sich als Spezialfall der anderen Abläufe darstellen lässt und dafür auch bereits Lösungen existieren, wie beispielsweise in (Rattihalli u. a., 2019).

10.1.1. Reaktive Skalierung von Plattformen

Die zentrale Funktionalität der reaktiven Skalierung von Plattformen umfasst die Annahme von Nachrichten und die entsprechende Umsetzung von Skalierungs-

operationen. Das Konzept in Kapitel 7.1.1 motiviert bereits den Einsatz eines dedizierten Management-Systems zur Kapselung der Interaktion mit der Infrastruktur. Ein Ziel dieses Kapitels sollte daher die Ausgestaltung eines solchen Management-Systems sein. Dabei wird ebenfalls der Einsatz einfacher Kommunikation eine Rolle spielen, um die Nachrichten zum Management-System abzubilden.

Da es sich um eine der grundlegenden Funktionalitäten der Architektur handelt, werden in diesem Zuge bereits Grundsteine der Kommunikationsformen gelegt und sowohl mögliche Realisierung als auch mögliche Kommunikationsformen ausführlich diskutiert. Innerhalb dieses Kapitels wird für die Differenzierung zwischen Management-System und der restlichen Realisierung der Architektur die Formulierung »Zwischen Simulation und Management-System« eingesetzt.

10.1.2. Proaktive Skalierung von Plattformen

Die proaktive Skalierung entspricht in ihren wesentlichen Bestandteilen der reaktiven Skalierung. Der einzige Unterschied liegt darin, dass das Plattformmanagement selbstständig die Skalierung anstößt. Da auf Ebene der Realisierung das Plattformmanagement aber eine logische Konstruktion ist, sollte dennoch eine Verortung dieser Funktionalität in einer Komponente des Plattformmanagements erfolgen. Wegen der Überlappung mit der reaktiven Skalierung liegt es daher nahe, auch diese Funktionalität im Management-System unterzubringen.

In späteren Ausführungen innerhalb dieses Kapitels wird sich für die Realisierung der proaktiven Skalierung eine hohe Synergie mit dem Einsatz bestehender Clustermanagementtools bilden.

10.1.3. Kommunikation, Nachrichtenbroker und Ablaufmanager

Die Funktionalität bezüglich Kommunikation umfasst im Wesentlichen die Bereitstellung von Infrastruktur. Da einfache Kommunikation keine Persistenz zu Zustellungsgarantie oder multilateralen Informationsaustausch vorsieht, genügt an dieser Stelle ein Netzwerk zwischen den einzelnen Plattformen als Basis der Kommunikation. Komplexe Kommunikation hingegen benötigt wie bereits angeführt unter Umständen einen Nachrichtenbroker. Nachrichtenbroker sind bereits in hoher Qualität als fertige und teils kommerzielle Projekte verfügbar. Konkrete Technologien diesbezüglich werden im Kapitel 14 zu realisierten Prototypen angesprochen.

Das Plattformmanagement fordert des Weiteren sogenannte Ablaufmanager, deren Funktionsumfangs jedoch zum jetzigen Zeitpunkt noch nicht vollumfänglich klar ist. Mit den Ausführungen in Kapitel 12 wird sich dies ändern und eine genaue Ausformulierung kann erfolgen. In diesem Kontext sollte daher insbesondere die Art der Bereitstellung bzw. das Deploymentformat der Ablaufmanager skizziert werden, um den Rahmen für die Realisierung aufzuzeigen. Das Deploymentformat sollte sich nach Möglichkeit in die Deploymentlandschaft der verbleibenden Komponenten eingliedern.

10.1.4. Update von Plattformdefinitionen

Das Update von Plattformdefinitionen ist eine weitere benötigte Funktionalität des Plattformmanagements. Dabei ist wichtig zu beachten, dass es sich hierbei um statische Änderung an der Basis der Plattformimplementation handelt. Da Plattformen generell für den Einsatz in heterogenen Umgebungen konzipiert sind, können Sie auf dynamische Unterschiede in ihren Umgebungen mithilfe spezialisierter Plattformfunktionalitäten reagieren. Folglich handelt es sich hierbei nur um Änderungen, die global alle Plattformen betreffen.

Entlang der Mushu-Architektur wird die Plattformdefinition als Ausgangspunkt für das Instanziieren neuer Plattformen eingesetzt. Daher ist eine Änderung der Plattformdefinition nicht als dynamische Operation zu verstehen. Vielmehr kann die geänderte Definition für neue Plattformen verwendet werden, hat aber keine Einflüsse auf bestehende Plattformen.

Die zentrale Anforderung besteht darin, auf entsprechende Updateinformationsnachrichten zu reagieren und das Update dementsprechend einzuspielen. Die konkrete Umsetzung hängt dabei stark vom Deploymentformat der Plattform ab, welches Teil des Realisierungskonzepts des Plattformmanagements ist. Das Deploymentformat wird in Abschnitt 10.6 aufgearbeitet. In diesem Zuge wird auch das Update von Plattformdefinitionen erörtert werden.

10.1.5. Zustandsabfragen

Die Fähigkeit, Zustandsabfragen zu bearbeiten, umfasst auf Ebene des Plattformmanagements in erster Linie Aggregationsfunktionen. Zu diesem Zweck sollte ein zentraler Service bereitgestellt werden, welcher diese Funktionalität ausfüllen kann. Die Kommunikation zwischen Aggregationsservice und einzelnen Plattformen und sonstigen Komponenten ist dabei nicht direkt eine einfache Kommunikation im Sinne der Definition aus Abschnitt 5.3.1, nutzt jedoch auch die Möglichkeiten der komplexen Kommunikation nur sehr eingeschränkt, da lediglich eine Anfrage und eine Antwort abgebildet wird. Daher kann argumentiert wer-

den, dass diese Form der Abfragen mit zwei einfachen Kommunikationen in Folge realisiert werden können. Dabei muss selbstverständlich auf die Möglichkeit des Nachrichtenverlustes eingegangen werden, falls Plattformen oder andere Komponenten aktuell nicht erreichbar sind.

Derartige Anforderungen sind jedoch nicht spezifisch für den hier genannten Anwendungsfall und je nach konkreter Implementation der Plattform kann unter Umständen auf bereits bestehende Aggregationslösungen zurückgegriffen werden. Diese Möglichkeiten werden im Rahmen der Prototypen in Kapitel 14 wieder aufgegriffen.

10.2. Mögliche Realisierungen des Management-Systems

Da das geplante Management-System die zentrale Komponente der zu erschaffenden Ausführungsumgebung darstellt, liegt es nahe, die Architekturüberlegungen für diese Komponente ins Zentrum der ersten Betrachtung zu legen. Hier getroffene Entscheidungen wirken sich maßgeblich auf die Interaktion vom Modellierer mit der der Simulation zugrunde liegenden Infrastruktur aus.

Das Management-System umfasst aus Sicht der Mushu-Architektur die (reaktive) Erzeugung und Vernichtung von Plattformen.

Da dem Modellierer eine möglichst allgemeine Schnittstelle zur Verfügung gestellt werden soll, sodass kein großes Infrastrukturwissen erforderlich ist, sollte selbstverständlich auch das Management-System eine derartige Schnittstelle bereitstellen. Da sich das Management-System je nach Infrastruktur, auf der die Referenznetzsimulation bereitgestellt wird, anders und angepasst verhalten muss, liegt die Lösung nahe, das Management-System als komplett autonomes, getrenntes System zu entwerfen. Dieser Ansatz hat zum einen den Vorteil, dass das Management-System als »grüne Wiese«-Projekt[1] gestartet werden kann und somit nicht direkt mit früheren Design- und Architekturentscheidungen integriert werden muss, weist aber auch den Nachteil auf, dass eventuell bereits bestehende Strukturen nicht einfach wiederverwendet werden können. Bei der Konstruktion können verschiedene Paradigmen bzw. Betrachtungswinkel verfolgt werden, von denen drei naheliegende an dieser Stelle genauer betrachtet und verglichen werden: ein Formalismus-basierter Ansatz, ein Webservice-basierter Ansatz und ein Kommunikationstechnologie-basierter Ansatz.

Beim ersten Ansatz soll das Management-System ebenfalls als Referenznetz (-simulation) konstruiert werden. Für diesen Ansatz bietet es sich an, ebenfalls

[1] Ein Projekt ohne zu integrierenden Legacy-Code

den Simulator Renew als Grundlage zu verwenden und ihn in der Form einer Managementkomponente als Plugin oder als Netzsystem zu erweitern. Ein Vorteil dieses Ansatzes läge darin, bei der Konstruktion der neuen Systemkomponente einheitlich auch den Referenznetz-Formalismus als Grundlage einzusetzen.

Der zweite betrachtete Ansatz umfasst den Einsatz eines Webservices. Diese Überlegung entsteht aus der gegenwärtigen Tendenz der architektonischen Entwicklung in Richtung von Microservices und webbasierten Diensten und der wünschenswerten Kompatibilität mit aktuellen Technologien. Die Konstruktion als Webservice würde die weitere Integration anderer Komponenten wie Clustermanager vereinfachen, da viele dieser Technologien bereits Webtechnologien als Grundlage einsetzen. Das Management-System könnte auf diese Art und Weise zu einem Bindeglied werden, um die Welt der Referenznetzsimulation und Webtechnologien zu integrieren. Diese Abbildung ist prinzipiell möglich. Dies wurde auch bereits in Abschnitt 4.4.3 beschrieben mit dem Verweis auf die Veröffentlichung (Moldt, Ortmann und Offermann, 2004), welche die Mulan-Architektur im Kontext von Webservices interpretiert.

Im direkten Anschluss und in Einheit mit diesem zweiten betrachteten Ansatz folgt die Frage, ob der ergänzende Einsatz fertiger Softwarelösungen und Industriestandards auch für diese wissenschaftliche Arbeit Vorteile bewirken kann. Dabei sind insbesondere großflächig genutzte Infrastrukturmanagementsysteme von Interesse. Wenn diese mit den Anforderungen an die Ausführungsumgebung für skalierende Referenznetzsimulationen vereinbar sind, wäre es möglich den Implementationsaufwand für die Neuerung auf den Adapter zwischen Referenznetzsimulation und Clustermanager zu reduzieren.

Eine weitere mögliche Implementation und damit der dritte Ansatz könnte auf Basis der verwendeten Kommunikationstechnologie erfolgen. Während die Wahl der verwendeten Kommunikationstechnologie zwischen Managementkomponente und Anbindung an die Simulation in Abschnitt 10.4 im Detail erläutert werden wird, hat die Wahl jedoch Wechselwirkungen auf die hier diskutierte Form der Architektur. Somit wäre es denkbar, eine Lösung zu konstruieren, die sich vollständig an Java-Technologien orientiert und somit direkter in die bisherigen technologischen Entwicklungen im Kontext der Referenznetzsimulation wie beispielsweise die bisherige Implementation des Distribute Plugins integriert werden kann. Die tatsächliche Herausforderung bei diesem Ansatz ist die Interaktion mit der zugrunde liegenden Infrastruktur und damit die Umsetzung der Kernaufgabe des Management-Systems.

Im Rahmen des Grundlagenkapitels wurde der Begriff der Zustandslosigkeit als eine zentrale, die Skalierbarkeit unterstützende Architekturrichtlinie eingeführt. Daher ist es unabdingbar, im Rahmen der Architekturüberlegungen zur Managementkomponente ebenfalls die Zustandslosigkeit zu diskutieren und zu entscheiden, ob die Zustandslosigkeit in diesem Kontext auch für das Management-System

eine erstrebenswerte Eigenschaft darstellt. Je nach gewählter Grundarchitektur bedarf es auch mehr oder weniger Aufwand, ein zustandsloses System zu konstruieren.

10.2.1. Realisierung im Referenznetzkontext

In Anbetracht des Einstiegs in die Thematik und der grundlegenden Entscheidung, Referenznetze als zugrunde liegenden Formalismus einzusetzen, liegt es nahe, auch bei der Konstruktion des Management-Systems einen Ansatz mit demselben Formalismus zu verfolgen und eine Realisierung als Referenznetzsystem anzustreben. Der Vorteil dabei läge in der gemeinsamen Bereitstellung von Modell und Implementation, wie es für Referenznetze typisch ist. Die dabei nahe liegenden verschiedenen technischen Ansätze umfassen wie eingangs bereits motiviert die Konstruktion eines separaten Plugins für den Simulator Renew oder aber die Konstruktion eines komplexen Referenznetzsystems zur Abbildung der Funktionalität des Management-Systems. Dieser gewählte Ansatz hätte den Vorteil, dass alle Anteile des Management-Systems auf demselben Formalismus aufsetzen. Dadurch wird die Analyse und insbesondere formale Verifikationen (die sich jedoch außerhalb des Umfangs dieser Arbeit befinden) des Systems deutlich erleichtert. Dagegen abzuwägen gilt es jedoch, wie einfach und geradlinig die Integration einer Realisierung auf Referenznetzbasis mit anderen Technologien ist.

Eine weitere Herausforderung entsteht dadurch, dass eine verlässliche Kommunikation zwischen den verteilten Komponenten der Referenznetzsimulation hergestellt werden muss. Auch hier könnte wiederum das Distribute Plugin zum Einsatz kommen, welches bereits vom Modellierer verwendet werden kann, um mit entfernten Knoten eine integrierte, verteilte Referenznetzsimulation zu erzeugen. In diesem Zuge muss für das Distribute Plugin auch eine Vermittlungseinheit bereitgestellt werden, bei der es sich klassisch um eine RMI Registry handelt. Daher ist es denkbar diese Vermittlungseinheit auch für den hier beschriebenen Anwendungsfall mitzuverwenden, wodurch der zusätzliche Aufwand nicht schwer ins Gewicht fallen würde.

In jedem Fall erfordert die Umsetzung mittels eines Referenznetzsystems die Bereitstellung eines geeigneten Simulators, wie Renew, als Management-System.

Konstruktion als Referenznetzsystem

Mit Referenznetzen lassen sich Abläufe durch die mächtigen Anschriftensprache sehr kompakt darstellen. In ihren Transitionsanschriften können komplexe Instruktionen aufgeführt werden. Dennoch liegt die Stärke der Petrinetze und allgemein von Modellierungssprachen darin, die Komplexität von realweltlichen fachli-

chen Abläufen – insbesondere im Falle der Petrinetze von nebenläufigen Abläufen – auf greifbare, grafische und meist zweidimensionale Art und Weise darzustellen. Wichtig im Kontext des Management-Systems ist dabei jedoch, dass nicht die Eigenschaft Systeme zu modellieren, sondern die technische Umsetzung mit Renew betrachtet werden soll.

Die mächtigen im Referenznetzformalismus umgesetzten Grundprinzipien wie hierarchische Strukturen, inhärente Nebenläufigkeit, synchrone Kanäle und multidirektionaler Informationsaustausch wirken übersetzt für die (exekutive) Kontrolle einfacher Infrastrukturkomponenten. Im Falle der Implementation im Simulator Renew können durch die in der zugrunde liegenden Programmiersprache Java verfügbaren Polymorphie verschiedene Infrastrukturen über gleiche Schnittstellen angesprochen werden. Eine Konstruktion mithilfe von Referenznetzen wäre also nicht nur wegen ihrer Turing-Mächtigkeit theoretisch durchaus realisierbar. Jedoch bedingt die Implementation der angesprochenen mächtigen Grundprinzipien eine deutlich erhöhte Komplexität und einen erhöhten Overhead, der nur dann zu rechtfertigen wäre, wenn die Anwendung daraus direkt Kapital schlagen kann.

Der größte Vorteil bei diesem Ansatz liegt in der direkten Erfüllung der als (technischen) Anforderung formulierten Bedingung, dass sich das Management-System nahtlos in bereits bestehende Verteilungslösungen für Referenznetzsimulation eingliedern würde, da es auf derselben Technologie basiert. Eine größere Herausforderung besteht darin, die für den Start des Distribute Plugins notwendigen Parameter zu verarbeiten. Da diese bereits vor dem Start eines entfernten Renew Simulators dem startenden System vorliegen müssen, ist es unabdingbar, dass auch ein als Referenznetzsimulation konstruiertes Management-System über andere Protokolle Kommunikation mit der Infrastruktur betreiben muss. Im Zweifel müssten direkte Programmaufrufe als Parameter an Transitionen angegeben werden, wie es beispielsweise in einem etwas anderen Kontext in der Veröffentlichung des Autors (Röwekamp, Moldt und Feldmann, 2018) zum Start von Docker-Containern umgesetzt wurde. Der referenzierte Prototyp wird im Abschnitt 14.1.2 im Detail vorgestellt werden. Die dabei entstehenden Referenznetze tragen unnötig viel technischen Detail-Ballast, welcher der Lesbarkeit des Modells abträglich ist. Dadurch schmälert sich der Vorteil, Referenznetze einzusetzen, erneut.

Konstruktion als Renew-Plugin

Aufgrund der zuvor beschriebenen Unübersichtlichkeit beim Einsatz eines reinen Referenznetzsystems liegt es nahe, die technischen Details als Plugin verpackt zu verbergen und hier eine weitere Abstraktionsschicht innerhalb des Management-Systems einzuziehen. Dabei würde die Generierung von Parametern in klassischen Java Code im Hintergrund ausgelagert werden, welcher durch einfache Befehle aus

einem Referenznetz abgerufen werden kann. Die Vorteile bei dieser Methode entsprechen im Wesentlichen den der im vorigen Abschnitt ausgeführten Umsetzung der Konstruktion als Referenznetzsystem mit dem Zusatz der übersichtlicheren Netze.

Als Hauptnachteil durch die weitere Einführung eines Plugins aufseiten des Management-Systems ist die zusätzliche technologische Komplexität der Lösung anzuführen. Diese steht im wesentlichen Widerspruch mit der generellen Anforderung, ein schlankes, flexibles System zu erzeugen. Die Implementation des Plugins mit Anbindung an Infrastruktursysteme wäre ebenfalls ein nicht unerheblicher handwerklicher Aufwand.

10.2.2. Realisierung als Standalone Webservice

Einen etwas anders gelagerten Ansatz verfolgt der diesem Abschnitt vorgestellte integrative Ansatz. Hauptziel dabei ist die möglichst einfache Integration von anderer Technologie, die in Unternehmens- und sonstigen Wissenschaftskontexten eingesetzt wird und als Industriestandard gilt. Ein Vorteil dabei ist zum einen die Möglichkeit auf vielfache Erfahrung und auch Implementationen dieser Ansätze als Stütze und Hilfsmittel zurückgreifen zu können. Zusätzlich kann so auch eine höhere Kompatibilität mit anderen Lösungen gewährleistet werden, sodass die Referenznetzsimulation und die hier vorgestellte Erweiterung auf eine skalierbare Ausführungsumgebung bessere Chancen für einen großflächigeren Einsatz erhält.

In Kapitel 4 wurde bereits erörtert, dass die Perspektive der Webtechnologien eine federführende Rolle spielen soll. Somit liegt es nahe, für das Management-System ebenfalls eine Konstruktion als Webservice bzw. als Webanwendung zu evaluieren. Über die Jahre und Jahrzehnte wurden Webtechnologien und Frameworks in unzählige Programmiersprachen und Umgebung übernommen. Zusätzlich liegt der Fokus der Technologien in diesem Zusammenhang ebenfalls auf der Integration verschiedener Technologien und Komponenten. Ferner arbeiten Webtechnologien inhärent mit Asynchronität in ihren Aufrufen und bieten dadurch eine solide Basis für eine verteilte Ausführung, wie sie erfolgen muss, wenn das Management-System – wie geplant – als eigenständige Komponente realisiert wird. All diese Eigenschaften bieten eine Bandbreite an Vorteilen für die Realisierung als Webanwendung.

Eine Integration mit bestehenden Tools zur schnellen Entwicklung geeigneter Prototypen wird gesondert noch einmal in Abschnitt 10.2.4 diskutiert.

Generell wird durch die Form als Webtechnologie kein großer Overhead an bereits implementierter Funktionalität vorgegeben. Insbesondere bestehen keine hochgradig komplexen Kommunikationsmechaniken wie bei der Umsetzung als Referenznetzsimulation. Vorgegeben werden in erster Linie Protokolle und Datenformate,

welche im Wesentlichen pro Aufruf zwei einfachen Kommunikationen (im Sinne der Definition im Rahmen der Arbeit) entsprechen. Von diesen Konventionen kann im Einzelfall auch abgewichen werden, sofern derartige Abweichungen gut begründet werden können, da sie auch immer mit einer Reduktion der Integrierbarkeit einhergehen. Dadurch wird die Anforderung an ein schlankes und flexibles System von dieser Technologie gut erfüllt.

Die beiden Herausforderungen bei der Konstruktion des Management-Systems als Webserver liegen darin, einerseits die Anbindung an die Referenznetzsimulation zu realisieren, sowie andererseits die für den Start des Distribute Plugins notwendigen Parameter an den entsprechenden Stellen in der Infrastruktur bereitzustellen. Jedoch kann der Ansatz Webtechnologien einzusetzen gerade bei der zweiten adressierten Herausforderung entscheidende Vorteile bieten, da aufseiten der Infrastruktur auf einschlägige Werkzeuge gesetzt werden kann, welche seit langer Zeit erprobt wurden. Die Integration mit der Infrastruktur gestaltet sich so als deutlich weniger aufwendig als bei einer manuellen Implementation dieser Anteile.

Die weiteren genannten Herausforderungen der Anbindung an die Referenznetzsimulation lassen sich gut dadurch realisieren, dass auch in Java geeignete Frameworks für die Anbindung an Webtechnologien existieren. Um das Management-System jedoch so kompatibel mit Webtechnologien zu halten wie möglich, liegt es nahe, den Overhead der Anbindung zwischen Referenznetzsimulation und Webtechnologie direkt auf der laufenden Plattform zu realisieren anstatt im (separaten) Management-System. Auf diesen Umstand geht noch einmal detaillierter der folgende Abschnitt 10.3 ein.

Zusammenfassend ist der Einsatz eines Webservices für das Management-System ein vielversprechender Ansatz, der jedoch noch weitere Integrationsarbeit erfordert. Die Erfolgsaussichten sind höher einzuschätzen als beim Einsatz einer Referenznetzsimulation als Management-System.

10.2.3. Realisierung als formfreie Java-Anwendung

Neben den beiden zuvor vorgestellten Ansätzen soll an dieser Stelle ein allgemein gehaltener dritter Ansatz genannt werden. Das maßgebliche Ziel bei diesem Ansatz liegt darin, die Komplexität der Anwendung durch den Einsatz gleichartiger Technologien, wie sie auch in der Referenznetzsimulation zum Einsatz kommen, gering zu halten. Dabei soll insbesondere keine Form durch Frameworks o. ä. vorgegeben werden, um die Lösung direkt auf die abzubilden Anforderung zuschneiden zu können. Der Hauptvorteil besteht darin, dass auf lange Sicht keine Abhängigkeiten zu anderen Frameworks aufgebaut werden, für die Wartungen auslaufen könnten.

Eine Eingliederung in die bisherigen Ansätze wäre realisierbar, wenn auch mit einer gewissen Menge an Aufwand verbunden. Ein Hauptnachteil besteht darin, dass unter Umständen viele bereits in anderer Form gelösten Probleme erneut implementiert werden müssten. Auch die Versorgung von Systemen mit geeigneten Startparametern für den Start eines Renew Simulators mit Distribute Plugin gestaltet sich als nicht trivial.

Implementationsarbeit ist wie bereits erwähnt in erster Linie eine handwerkliche Tätigkeit. Da wie eben beschrieben dieser Ansatz in erster Linie einen großen Teil an Implementationsarbeit mit sich bringen würde, ist der Ansatz insbesondere aufgrund seines Umfangs nicht pragmatisch für das Ziel der Arbeit eine Simulationsumgebung bereitzustellen, die Skalierung bestmöglich unterstützt. Daher soll hier keine weitere Betrachtung dieses Ansatzes erfolgen, sondern der Fokus auf die beiden verbleibenden Ansätze gerichtet werden.

10.2.4. Kombination mit bestehenden Clustermanagementtools

Wie bereits in den vergangenen Abschnitten motiviert, kann es sinnvoll sein, einen externen Clustermanager in das Management-System zu integrieren. Die Motivation liegt darin begründet, dass die Verwaltung von Infrastruktur kein für Referenznetzsimulationen spezifisches Problem darstellt. Auf diese Weise kann der Aufwand, individuell verschiedene Infrastruktur in ein Gesamtsystem zu integrieren deutlich reduziert werden. Jedoch existiert auf der anderen Seite auch der Nachteil, dass die Einbindung eines externen Systems eine weitere Abhängigkeit in das Gesamtsystem integriert. Je nach Wahl der verwendeten Technologie kommen zusätzlich Lizenzierungsprobleme und etwaige Kosten hinzu. Eine weitere Herausforderung beim Einsatz eines externen Systems liegt darin, dass ein Adapter zur Schnittstelle des externen Systems konstruiert werden muss. Der Aufwand das externe System zu integrieren generiert nur dann einen Mehrwert, wenn sich die Schnittstelle vom Referenznetzsystem ausreichend direkt ansprechen lässt, sodass der Aufwand, den Adapter zu konstruieren nicht den Aufwand übersteigt, den eine direkte Implementation des Infrastrukturmanagements sich bringen würde. Daher sollte die Wahl eines verwendeten Clustermanagers sehr differenziert und überlegt erfolgen.

Die effektive Entscheidung darüber, ob ein derartiges externes System eingesetzt werden soll oder nicht, entspricht der Entscheidung, in welcher Dimension die Untersuchungen am Gesamtsystem erfolgen sollen. Da der Fokus der Arbeit allerdings insbesondere auf der Skalierbarkeit der betrachteten Systemteile liegt, ist es unerlässlich, zumindest die technische Möglichkeit bereitzustellen, mit wenig manuellem Aufwand auch große Simulationen zu starten. Dies erfordert auf

technischer Ebene jedoch enorme Mengen an Implementations- und Testarbeit, die nicht direkt die Forschungsfrage adressieren würden. Aus diesem Grund soll im Rahmen der Arbeit ein entsprechender Clustermanager eingesetzt werden. Er bildet aus Sicht der MUSHU-Architektur die Schnittstelle zwischen der Funktionalität des Plattformmanagements, Plattformen zu erzeugen und zu zerstören und der technischen Infrastruktur.

Es existieren hierfür bereits viele erprobte und teilweise kommerzielle Lösungen, wie beispielsweise Apache Mesos, OpenStack, OpenShift, Kubernetes und weitere. Da sich die Überlegung bezüglich der Realisierung an dieser Stelle auf ein konzeptuelles Level beschränkt, liegt es nahe, bereits Auswahlkriterien für spätere Prototypen in Kapitel 14 zu betrachten. Die zunächst dabei erwähnten Kriterien sind weitestgehend nicht spezifisch für Clustermanager, sondern gelten generell für die Einbindung von Software von Drittanbietern.

In erster Linie steht die Frage nach der *Lizenz der Software* im Raum, da diese weitreichend entscheidet in welcher Form und in welchem Umfang sie verwendet werden kann. Anschließend an die Frage nach der Lizenz schließt sich unmittelbar die Frage nach den *Kosten der Software* an. Hierbei sind klar frei verfügbare Lösungen zu präferieren, auch wenn diese im Funktionsumfang gelegentlich hinter proprietären Lösung anstehen aber nicht müssen. Als weitere Anforderung ist eine *aktive Entwicklung* zu nennen, um sichergehen zu können, dass Schwachstellen und Bugs zeitnah behoben werden können. Des Weiteren wäre es wünschenswert, wenn eine *Organisation ohne direktes kommerzielles Interesse* hinter der Software steht, um ungünstige Entwicklungen in eine kommerziell nachvollziehbare aber gegebenenfalls wissenschaftlich ungünstige Richtung zu vermeiden. Eine *aktive Community* ist ebenfalls eine sinnvolle Anforderung, um Hilfe bei der Einrichtung und der Pflege zu erhalten.

10.2.5. Einsatz von Zustandslosigkeit

Wie bereits in Abschnitt 2.5.2 diskutiert, ist Zustandslosigkeit eine sehr günstige Eigenschaft eines Systems in Bezug seine Skalierbarkeit. Ein zustandsloses System speichert, wie der Name suggeriert, keinen Zustand. Das mehrfache Ansprechen eines Teils seiner Schnittstelle mit identischen Daten, führt stets zum selben Ergebnis, da die Berechnung von keinem internen Zustand des Systems abhängt. Dabei ist zu beachten, dass Daten aus einer etwaig angeschlossenen Datenbank auch zu den Eingabedaten zählen. Zustandslose Systeme können auch als Sammlung von Funktionen betrachtet werden. Da keine Daten direkt vom System verwaltet werden müssen, ist eine Skalierung wesentlich unproblematischer möglich als in klassischen zustandsbehafteten Systemen.

Die Frage, welche in diesem Abschnitt behandelt werden soll, lautet daher nicht, ob Zustandslosigkeit ein wünschenswerter Aspekt des Management-Systems ist. Es soll vielmehr eine Diskussion darüber erfolgen, wie aufwändig sich die Umsetzung als zustandsloses System je nach gewählter Grundrealisierung gestaltet. Die Erkenntnisse bezüglich der Realisierungsmöglichkeiten von Zustandslosigkeit können in die finale Entscheidung einbezogen werden.

Zustandslose Referenznetzsimulation

Die Gestaltung einer zustandslosen Referenznetzsimulation gestaltet sich als nicht trivial. Um die entsprechende Funktionsmächtigkeit umsetzen zu können, bietet es sich an, auf der Implementation des Simulators Renew aufzusetzen. Renew befindet sich bereits seit langem in Entwicklung und die Zustandslosigkeit war an wenigen bis keinen Stellen des Entwicklungsprozesses eine formulierte Anforderung.

Soll nun doch nachträglich Zustandslosigkeit in den Simulator integriert werden, so existieren mehrere mögliche Ansätze für die Implementation.

Eine einfache Variante der Implementation von Zustandslosigkeit könnte auf der fachlichen Ebene erfolgen, indem ein Netz konstruiert wird, welches bei der Abarbeitung einer Anfrage zwar temporär einen Zustand aufweist, jedoch nach Ende der Abarbeitung der Anfrage alle Daten (Marken) aus der Anfrage löscht (verbraucht). Das Netz müsste eine entsprechende externe API bereitstellen, an die Anfragen gestellt werden können. Es müsste auch so aufgebaut sein, dass garantiert werden kann, dass eine Anfrage an die von ihm erzeugte API bei jeder Wiederholung zu gleichen Ergebnissen kommt und keine Zustände in Form von übrig bleibenden Marken auf Plätzen gespeichert bleiben. Die Zustandslosigkeit wäre in diesem Fall allerdings nur auf Ebene der Anfrage realisiert, da die Simulation durch Renew nach wie vor sehr wohl zustandsbehaftet durchgeführt wird und der Simulator so einen internen Zustand hat.

Dieser Umstand motiviert es, den Blick auf eine niedrigere Ebene zu lenken und die Implementation einer zustandslosen Referenznetzsimulation selbst zu überdenken. Die folgenden Absätze diskutieren diese Idee zunächst unabhängig von ihrem Einsatz als Management-System.

Da auch zustandslose Systeme häufig zustandsbehaftete Systeme wie Datenbanken oder andere Systemteilen referenzieren und verwenden, können zustandslose Systeme häufig auch erst im verteilten Fall wirklich sinnvoll eingesetzt werden, da sonst keinerlei persistente Daten vom Gesamtsystem verarbeitet werden könnten. Folglich sollte eine zustandslose Referenznetzsimulation auch immer als verteilt angesehen werden. Die vollständige Entkopplung von Zuständen und der Extraktion eines zustandslosen Anteils innerhalb der Anwendung bedarf einer kompletten

Umstrukturierung des Simulationskerns im Kontext der zustandslosen Simulation von Referenznetzen.

Da Referenznetze ein sehr komplexer Formalismus mit mächtigen Bestandteilen sind, ist dies eine nicht zu unterschätzende Aufgabe. Eine Trennung an geeigneter Stelle wäre notwendig, um die Netze zu verteilen, wie dies im Kontext der Arbeit entlang der Mulan- und Mushu-Agentenmetapher erfolgt ist. Zustandslose Systeme sind jedoch nicht zwangsläufig auf den Agentenkontext beschränkt. Eine derartige Stelle könnte allgemein der Aufruf von synchronen Kanälen darstellen, da hier häufig auch die Hierarchieebene bzw. andere Netzinstanzen angesprochen werden. Zustandslose Systeme weisen häufig idempotente[2] Schnittstellen auf. Eine entsprechende Implementation müsste gewährleisten, dass nach Feuern eines synchronen Kanals die angesprochene andere Netzinstanz ein gleiches Verhalten auch bei erneuter Ausführung aufweist. Da Petrinetze aber inhärent Nichtdeterminismus abbilden können, muss diese Bedingung insofern aufgeweicht werden, als das dahinterliegende Netz nur einen ebenfalls erreichbaren äquivalenten Zustand annimmt für eine erneute Ausführung.

Einer der Vorteile von zustandslosen System besteht darin, dass ein (replizierter) Systemteil zu jeder Zeit abgeschaltet werden kann und etwaige auf Antwort wartende Kommunikationspartner Ihre Anfrage einfach erneut an einen äquivalenten Systemteil senden können. Dadurch ergibt sich aber das Problem, dass ein lokaler Knoten den Zustand der lokalen Simulation beim Aufruf eines entfernten synchronen Kanals im Zweifel vorhalten müsste, um den Aufruf ein weiteres Mal auf identische Weise ausführen zu können. Je nachdem, wie viele Synchronisationspartner an dem Aufruf beteiligt sind und wie komplex der Vorbereich der synchronen Transition gestaltet ist, kann dadurch ein nicht unerheblicher Overhead entstehen.

Zusammenfassend lässt sich festhalten, dass eine vollwertige Implementation eines zustandslosen Referenznetzsimulationssystems einen sehr hohen Aufwand mit sich bringen würde. Die Umsetzung kann problemlos Gegenstand einer eigenen Arbeit sein. Erste Untersuchungen in Hinblick auf die Unterstützung von funktionalen und seiteneffektfreien Anschriften bietet (M. Simon, Moldt u. a., 2019). Diese können einen hervorragenden Ausgangspunkt für die Konstruktion zustandsloser Plattformen bilden. Die Variante eine API einzusetzen wie zunächst beschrieben, wäre möglich, würde jedoch die Idee, durchgehend eine Referenznetzsimulation einzusetzen untergraben. Da an dieser Stelle lediglich ein Management-System für die Skalierungsumgebung von verteilten Referenznetzsimulationen konstruiert werden soll, würde eine zustandslose referenznetzbasierte Lösung den Aufwand der Skalierung der Simulation selbst höchstwahrscheinlich deutlich übersteigen.

[2] Eine idempotente Operation kann beliebig oft ausgeführt werden und führt dabei zu keiner Zustandsveränderung bzw. zu keinem Ergebnis, welches von der einmaligen Ausführung der Operation abweicht.

Zustandsloser Webservice

Die Konstruktion eines zustandslosen Webservices ist im Gegensatz zu der Variante eine Referenznetzsimulation einzusetzen in vielen Gesichtspunkten einfacher. Zunächst existiert kein komplexer Formalismus als Grundlage, welcher ebenso komplex umgesetzt werden müsste. Darüber hinaus ist das umgesetzte Kommunikationsprotokoll häufig HTTP oder HTTPS, welches durch seine Header bereits bestimmte Methoden vorsieht, deren intendierte Semantik Idempotenz umfasst. Webservices sind häufig im Hintergrund an eine klassische relationale Datenbank oder aber auch neuartigere Datenbanken angeschlossen. Ihre vorgehaltenen Routinen werden zumeist mit entsprechenden Informationen zu der Session mit dem Client vom Client aus versorgt und können so durch den Datenbankzugriff selbst äußerst generisch gehalten werden. Eine vielfach von Webservices eingesetzte Technologie ist der »Representational State Transfers« kurz REST, wie er bereits in zugehörigen Grundlagenkapitel in Abschnitt 2.5.2 diskutiert wurde. REST wird häufig mit dem HTTP-Protokoll in Verbindung gebracht, ist konzeptuell aber eigenständig.

Durch diesen Umstand lassen sich Webservices relativ leicht zustandslos implementieren. Aus diesem Grund existiert für die Entwicklung derartiger Webservices eine Vielzahl von technologischen Lösungen in Form von Frameworks oder eigenständigen Systemen. Eine Implementation als Webservice hätte somit den Vorteil, dass auch hier auf externer Technologie aufgesetzt werden kann.

Zustandslose formfreie Java-Anwendung

Bei der formfreien Konstruktion einer Java-Anwendung steht die Flexibilität der Implementation im Vordergrund. Daher ist es mit der entsprechenden Vorbereitung problemlos möglich diese Anwendung zustandslos zu entwerfen. Sofern die Implementation zustandslos erfolgt, muss gewährleistet werden, dass etwaige Daten an geeigneter Stelle vorgehalten werden, wie beispielsweise in einer Datenbank. Die Wahl des Kommunikationsprotokolls hat an dieser Stelle einen entscheidenden Einfluss auf die Möglichkeit der Implementation von Zustandslosigkeit innerhalb der Anwendung. Die Wahl des Kommunikationsprotokolls zum Management-System hin wird noch einmal detailliert in Abschnitt 10.4 behandelt. Wie jedoch bereits in Abschnitt 10.2.3 ausgeführt wurde, ist im hiesigen Kontext der Aufwand einer vollständigen Eigenimplementation nicht zu vertreten. Daher soll auch die Diskussion über die Zustandslosigkeit hier nicht weiter im Detail geführt werden.

Art der Umsetzung	*Ref.netzsim.*		*Webservice*		*Frei (Java)*	
Einsatz eines Clustermanagers	ja	nein	ja	nein	ja	nein
Kontrolle über die Skalierung	+	+	+	+	+	+
Nahtlose Eingliederung	++	+	o/+	o	+	+
Distribute Plugin starten können	+	-	++	o	+	o
Anpassungen am Referenznetzsimulator	-/+	-/+	o/+	+	o/+	o/+
Schlankes und flexibles System	o	-	++	o/+	+	+
Implementationsaufwand gering	o/+	-	+	-/o	- -	- -
Umsetzbarkeit von Zustandslosigkeit	-	-	++	o	+	o

Tabelle 10.1.: Verschiedene Ansätze für das Design des Management-Systems. Bewertung negativ bis positiv: - -, -, -/o, o, o/+, +, ++. Das Symbol -/+ weist auf eine kontextabhängige Bewertung hin.

10.2.6. Vergleich der Ansätze, Bewertung und Entscheidung

Abschließend werden in diesem Abschnitt noch einmal die drei Ansätze zusammengefasst und in Hinblick auf die in Abschnitt 9.3 formulierten Anforderungen miteinander verglichen. Ziel ist es dabei die konzeptuelle Grundlage und Systemarchitektur für die Prototypen in Kapitel 14 zu schaffen. Dazu sind die verschiedenen Ansätze und ihre jeweiligen Bewertungen noch einmal zusammenfassend in Tabelle 10.1 aufgeführt. Dabei symbolisiert das Zeichen **- -** eine sehr schlechte Erfüllung der Anforderung, **-**, **-/o**, **o**, **o/+**, und **+** Zwischenstufen in dieser Reihenfolge und **++** eine sehr gute Erfüllung. Dem Symbol **-/+** kommt eine Sonderstellung zu: Die Anforderungserfüllung ist abhängig von zusätzlichen Designentscheidungen. Diese werden in der textuellen Ausführung erläutert.

Neben den Anforderungen, wie sie in Abschnitt 9.3 formuliert wurden, wurde der Punkt »Umsetzbarkeit von Zustandslosigkeit« hinzugefügt.

Bei der Betrachtung der Gesamtergebnisse fällt ein Favorit auf: Die meisten Anforderungen können von der Implementation als Webservice mit Integration eines Clustermanagers gut bis sehr gut abgedeckt werden. Ebenfalls attraktiv wirkt die Implementation als formfreie Java-Anwendung, allerdings sollte hierbei das Augenmerk auf den Implementationsaufwand gerichtet werden. Die Implementation ohne den Einsatz nennenswerter Frameworks würde mit einem nicht zu vertretenen handwerklichen Aufwand einhergehen und damit für den Rahmen der Arbeit nicht zielführend sein.

Die Lösung durch ein Referenznetzsystem kann wie erwartet durch nahtlose Eingliederung in die bestehende Lösung punkten. Je nach Art der Umsetzung sind jedoch einige Anpassungen am Referenznetzsimulator selbst notwendig, wenn beispielsweise auf Simulationsebene Zustandslosigkeit implementiert werden soll. Weitere Ausführungen hierzu finden sich in Abschnitt 10.2.5. Für die Variante ohne einen Clustermanager ist darüber hinaus der Implementationsaufwand deutlich erhöht, da alle Interaktionen mit der Infrastruktur durch Referenznetze abgebildet werden müssen.

Zusammengefasst konnte durch die hier dargestellten Abwägungen die Entscheidung gefällt werden, dass das Management System auf Basis eines Webservices mit Anwendung eines Clustermanagers realisiert werden sollte. Die genaue Ausprägung und ein Beispiel mit konkreten technischen Komponenten dazu werden in Kapitel 14 diskutiert.

10.3. Management-System: Konstruktion der Simulationsanbindung

Anschließend an die Ausführungen zur Konstruktion der Management-Komponente ist es notwendig eine Anbindung in die Simulation zu schaffen. Den ersten Anhaltspunkt hierzu liefert die abstrakte Struktur der Mushu-Architektur. Die hier referenzierte Funktionalität wird durch die reaktiven Komponenten des Plattformmanagements abgebildet, welche auf interne Agentennachrichten reagieren können, um neue Plattformen zu erzeugen und zu zerstören. Während so die generelle Struktur vorgegeben wird, verlässt die Anbindung – auf Basis der Entscheidung einen Webservice einzusetzen – die Referenznetzsimulation. Folglich ist die Konstruktion einer entsprechenden Schnittstelle zwischen Simulation und Management-Komponente nötig.

Dabei soll in diesem Abschnitt weniger die Art der Kommunikation zwischen Management-Komponente und laufender Simulation betrachtet werden, sondern zunächst die lokale Architektur der Simulationsanbindung selbst. Da im Rahmen der Arbeit Referenznetzsimulationen mit dem Simulator Renew betrachtet werden, liegt es nahe, die Simulationsanbindung selbst in Renew zu integrieren. Da Renew in seiner aktuellen Form als Plugin-System implementiert ist, ergibt sich als logische Form die Konstruktion eines Plugins für Renew.

Aus Sicht des Modellierers stellt die Anbindung an die Simulation die zentrale Schnittstelle dar, durch welche die physikalische Ausdehnung der Simulation beeinflusst werden kann. Eine logische erste Frage ist daher, welcher Funktionsumfang dem Modellierer durch diese Schnittstelle bereitstehen soll.

Aufbauend auf dem Umfang gilt es danach zu erörtern, in welcher Form die Interaktion stattfinden soll. Das Modell der MUSHU-Architektur gibt generell vor, dass Nachrichten bezüglich der Anzahl an Plattformen beim Plattformmanagement eingehen können sollen, auf deren Grundlage das Plattformmanagement Plattformen hinzufügt oder entfernt. Der Inhalt dieser Nachrichten ist in der Petrinetz-Darstellung jedoch nicht weiter ausformuliert. Abschnitt 7.3.3 motiviert für die Umsetzung bereits den Einsatz des Reconciler-Patterns, welches die Formulierung dieser Nachrichten als idempotente Zustandsbeschreibungen vorgibt.

10.3.1. Umfang der Schnittstelle

Für die konkrete Ausgestaltung der Schnittstelle und der Realisierung des Reconciler-Patterns ist es hilfreich erneut gedanklich die Position des späteren Modellierers des Systems einzunehmen. Da bei der Beschreibung des Modells ansonsten vor allem fachliche Aspekte relevant sind und weniger die technischen Implementationsdetails, sollte die Schnittstelle, die vom Modellierer zu bedienen ist, in ihrer Komplexität gering gehalten werden.

Es klingt verlockend, Informationen zur Infrastruktur auszulesen und diese direkt und ungefiltert an den Modellierer zurückzuspiegeln, sodass dieser eine Entscheidung für die Akquisition neuer Ausführungsknoten modellieren kann. Jedoch hätte dieser Ansatz eine zu geringe Abstraktion von den technischen Gegebenheiten. Vielmehr sollte die Kontrolle darüber, welche exakten physikalischen Knoten in welcher Situation akquiriert werden, automatisch durch das Plattformmanagement erfolgen. In jedem Fall aber sollte dem Modellierer Kontrolle darüber eingeräumt werden, eine Anzahl weiterer Knoten in die Simulation aufzunehmen bzw. diese wieder freizugeben, ohne die technischen Details beachten zu müssen.

Ein direkter Start von Plattformen erfordert trotz vieler Optimierungen der eingesetzten externen Tools Zeit. Daher muss die aktuelle Menge an verfügbaren Plattformen nicht unbedingt den jüngsten Wünschen des simulierten Modells bzw. dessen Agenten entsprechen. Dazu ist es hilfreich, wenn die Diskrepanz zwischen gewünschter Anzahl an Plattformen welche an der Simulation beteiligt sind und der tatsächlichen Menge ersichtlich ist, bis der Reconciler im Management-System diese Werte einander angeglichen hat.

Insgesamt ergeben sich somit die drei Kernbestandteile der Schnittstelle:

- Anfrage zur Änderung auf eine bestimmte Anzahl an Plattformen
- Abrufen der tatsächlichen Anzahl der verfügbaren Plattformen
- Abrufen der gewünschten Anzahl an verfügbaren Plattformen

10.3.2. Form der Implementation

Eine der Überlegungen umfasst die Syntax der Anbindung, bzw. deren generelle Struktur. Da die verwendete Verteilungslösung für Renew das Distribute Plugin ist, welches seinerseits wie in Abschnitt 2.8.8 beschrieben den Referenznetzformalismus um einzelne Elemente erweitert, liegt es nahe, für die Simulationsumgebungssteuerung ebenfalls eine Erweiterung des Formalismus vorzunehmen.

Jedoch muss bei solch einer Änderung jeder formale Aspekt des zugrunde liegenden Formalismus erneut bedacht und ausgewertet werden. Neue Beweise sind gegebenenfalls zu formulieren und Verhaltensmuster könnten sich durch eine Formalismusänderung jenseits der geplanten Anpassung ändern. Darauf aufbauend ergibt sich die Frage, ob nicht die Bereitstellung von speziellen Funktionen innerhalb einer Transitionsanschrift ausreichend wäre, um die geforderte Funktionalität abzudecken, ohne den Formalismus und damit die Semantik zu ändern.

Beide Lösungen haben gemein, dass sie innerhalb der abgebildeten Referenznetze optisch sichtbar sind. Es ist zu diskutieren, ob dies ein gewünschter Aspekt der Erweiterung ist, oder ob eine unsichtbare Integration wünschenswerter ist. Eine für die simulierten Referenznetze unsichtbare Lösung könnte beispielsweise mit der Positionierung weiterer Metadaten in Form von Dateien neben den Referenznetz-Dateien realisiert werden.

In den folgenden Abschnitten werden diese verschiedenen Ansätze nacheinander erläutert und im Kontext der Anforderungen abgewogen. Auf Basis der Ausführung aus dem letzten Abschnitt ist die Umsetzung von Zustandslosigkeit an dieser Stelle durch die Begrenzung auf eine Referenznetzsimulation nur schwer zu realisieren und soll daher als Gegenstand weiterer, über die Arbeit hinausgehender, Untersuchungen verbleiben.

Erweiterung des Referenznetzformalismus

Die Integration des Distribute Plugins in Renew erfolgte durch eine Erweiterung des Referenzformalismus. Diese Erweiterung lässt sich logisch dadurch begründen, dass durch die Verringerung der durchgeführten Kommunikationsrunden bei der Unifikation in der Auswertung eines synchronen Kanals ein anderes Verhalten des Referenznetzes auftritt.

Es könnte auf den ersten Blick argumentiert werden, dass durch eine Inkorporation von physikalischen Simulationsknoten ebenfalls die Semantik eines Referenznetzes geändert wird. Dadurch, dass eine zusätzliche entfernte Berechnungsinstanz zur Verfügung steht, sind Schaltungen möglich, die vorher unter Umständen nicht möglich waren. Betrachtet man diesen Umstand allerdings genau, hätte diese Änderung ebenfalls durch das lokale Erzeugen einer weiteren Netzinstanz, die

der entfernten Netzinstanz gleicht, erreicht werden können. Daher handelt es sich beim Zuschalten weiterer Knoten zur Simulation um einen mit dem originalen Referenznetzformalismus bzw. den durch das Distribute Plugin abgewandelten Formalismus konzeptuell abbildbaren Vorgang.

Wie eingehend erläutert hätte eine Erweiterung des Formalismus eine Reihe negativer Implikationen zur Folge. Etwaige formulierte Beweise müssten gegebenenfalls neu formuliert werden, Untersuchungen zum Verhalten von Referenznetzsystemen müssten wiederholt werden und es würde eine neue Klasse an Referenznetzen erzeugt werden, die noch weitgehend ununtersucht ist. Eine derartige Änderung sollte nur dann erfolgen, wenn eine stark begründete Notwendigkeit dazu existiert. Dies ist beispielsweise gegeben, wenn sich das Verhalten nicht mehr mit der Semantik des alten Formalismus abbilden lässt. Eine Erweiterung des Referenznetzformalismus wäre somit nicht nur kompliziert, sondern auch nicht korrekt bzw. notwendig.

Zusammenfassend überwiegen die Nachteile der Einführung eines eigenen Subformalismus deutlich gegenüber den vermeintlichen Vorteilen. Daher wird diese Umsetzung nicht weiter verfolgt, solange keine grundlegenden Änderungen im Verhalten der Simulation erzwungen werden, die dies notwendig machen würden. Eine derartige Änderung im Verhalten der Simulation wäre beispielsweise die Integration der Annahme, dass während einer Synchronisation Knoten verschwinden bzw. ausfallen könnten. Da diese Überlegung jedoch kein Aspekt der Skalierung selbst ist, sondern vielmehr der Verteilung von Referenznetzen, wird in diesem Kontext auf nachfolgende Untersuchung jenseits dieser Arbeit verwiesen.

Bereitstellung über integrierte Funktionen

Eine nicht ganz so tiefgreifende Änderung, wie die Abwandlung des zugrunde liegenden Formalismus, bildet die Einführung von speziellen Funktionen. Die Implementation kann somit als einfache Java Funktionen erfolgen und auf die Funktionalitäten kann durch statische Aufrufe an einer Transition zugegriffen werden. Um im Rahmen der sonstigen Implementation des Simulators Renew zu verbleiben, könnten diese Funktionen direkt am neu zu erzeugenden Plugin für Renew aufrufbar sein.

Während die Implementation so relativ einfach umgesetzt werden kann, ist diese Erweiterung im Modell sichtbar. So muss für die Steuerung eine Referenz auf das Plugin-Objekt gehalten werden und dieses in den Aufrufen stets referenziert werden. Werden jedoch weitere im Kontext des Simulators entwickelte Plugins betrachtet, handelt sich dabei durchaus um eine gängige Praxis. Die Sichtbarkeit wäre zwar gegeben, jedoch nicht übermäßig groß. Eine weitere Abmilderung des Problems der Sichtbarkeit kann dadurch erzielt werden, dass die gesamte

Skalierungssteuerung in ein separates Referenznetz ausgelagert wird, welches lediglich durch synchrone Kanäle aus der Hauptsimulation erreichbar ist. Wird zusätzlich auch noch mit Uplinks in der Hauptsimulation und Downlinks in der Skalierungssteuerung gearbeitet, kann die Skalierungssteuerung für die Hauptsimulation nahezu unsichtbar erfolgen, mit dem Nachteil der Konzentration von verschiedensten Referenzen auf Netzinstanzen im Skalierungssteuerungsnetz. Die konkrete Gestaltung in diesem Fall obliegt hierbei jedoch nicht der Realisierung der MUSHU-Architektur, sondern dem Modellierer konkreter Netze für die Ausführung im hier beschriebenen System. Folglich ist der hier beschriebene Aufbau als Anregung für eine Kapselung zwischen Fachlichkeit und reaktiver Skalierung zu betrachten.

Darüber hinaus existieren für diese Variante keine weiteren gravierenden Nachteile, die den Vorteilen der einfachen Integration, Implementation und einfachen Benutzbarkeit gegenüberstehen würden.

Deskriptive Skalierungsdefinition als Metaelement

Eine weitere mögliche Bereitstellungsform ist die als Metaelement zu Referenznetzen. Dabei könnte neben einer oder jeder Datei, die ein statisches Referenznetz repräsentiert, eine zusätzliche Datei bestehen, welche die Bedingungen für eine Änderung der Skalierung beschreibt. Das Ziel dabei ist in erster Linie, die Referenznetzdarstellung möglichst schlank zu halten und auf fachliche Abbildungen zu reduzieren. Die Grundidee bei diesem Ansatz liegt darin, in der separaten Datei keine Referenznetzdarstellung einzusetzen, da die Variante eine separate Datei in Referenznetzdarstellung zu erzeugen mit dem zuvor vorgestellten Ansatz wie beschrieben bereits möglich ist. Die innere Struktur dieser Metadatei sollte einen deskriptiven Charakter besitzen und Bedingungen formulieren, bei welchen eine Hoch- bzw. Herunterskalierung notwendig ist. Für diesen Abschnitt sollen die Inhalte dieser Metaelement-Datei folgend als »Skalierungsdefinition« bezeichnet werden.

Eine der ersten dabei auftretenden Fragen ist, wie die Integration zwischen Skalierungsdefinition und Referenznetz stattfinden soll. Während die im letzten Abschnitt beschriebene Integration über synchrone Kanäle mit den Transitionen primär die aktiven Elemente des Netzes anspricht, ist zu überlegen, ob durch die Auswertung von Platzinhalten eine bessere Aussage über die Notwendigkeit von Skalierung getroffen werden kann. Ähnlich wie eine Platz-Invariante könnte so über die Menge an Marken oder die Art der Marken in bestimmten Plätzen eine notwendige Menge an ausführenden Instanzen definiert werden.

Der Vorteil bei dieser Methode liegt klar darin, dass sehr differenziert Bedingungen für die Skalierung vergleichsweise einfach formuliert werden können. Durch

den deskriptiven Charakter können gewünschte Zustände formuliert werden im Gegensatz zu einzelnen Aufträgen. Auch dies würde eine gute Integration mit der Reconciler-Form der beabsichtigten weiteren Implementation ergeben. Ein weiterer Vorteil dieser Methode ergibt sich dadurch, dass die Abhängigkeit nur von Skalierungsdefinition zu Referenznetz existiert (Das Referenznetz referenziert die Skalierungsdefinition nicht). Daher kann bei der Modellierung die Konstruktion der Skalierungsdefinition an einen anderen Entwickler oder Modellierer abgetreten werden.

Auf der nachteiligen Seite ist zu nennen, dass weitere Dateien die Datenhaltung weniger übersichtlich gestalten. Auch bei Anwendern, welche nicht mit Skalierungsdefinitionen vertraut sind, könnte das System unerwartetes Verhalten aufweisen oder gänzlich in einen Fehlerzustand laufen, falls etwaige Plätze nach einer Umstrukturierung nicht mehr gefunden werden. Darüber hinaus kann die Implementation der Skalierungsdefinition aufwändig ausfallen, da etwaige Zusammenhänge zwischen mehreren Plätzen betrachtet werden, ähnlich wie bei der Berechnung von Invarianten in Netzen.

Auf der Implementationsseite ergibt sich eine wesentlich komplexere Arbeit für die Umsetzung des beabsichtigten Verhaltens. Zunächst ist es notwendig, dass eine geeignete Sprache für Skalierungsdefinitionen geschaffen wird. Auf dieser Grundlage muss ein Parser implementiert oder verwendet werden, welcher diese Art von Skalierungsdefinitionen einlesen kann und darauf aufbauend ein Überwachungsalgorithmus, welcher jede Änderung an den Inhalten von Plätzen erkennt und gegen die Skalierungsdefinitionen abgleicht.

An dieser Stelle stellt sich auch die Frage, wie mit dem Konzept der Netzinstanzen umgegangen werden soll. Wird die Skalierungsdefinition gleichzeitig auf alle Netzinstanzen ohne Austausch angewandt, ist zu erwarten, dass fast immer konfligierende Skalierungsanforderungen entstehen werden. Dies gilt insbesondere, wenn die Referenznetzsimulation mit True-Concurrency-Semantik ausgeführt wird. Mögliche Implementationen hierbei könnten eine führende Netzinstanz bestimmen, bei der Plätze entsprechend überwacht werden sollen, oder aber die Summe aller Netzinstanzen und der darin in Platzinstanzen liegenden Marken des zugehörigen Netzes berechnen. Auch bei der Formulierung von Bedingungen, sofern diese sich nicht nur auf die reine Anzahl an Marken beziehen, muss bedacht werden, dass im Fall der in Renew implementierten Referenznetze auch komplexe Objekte von Marken in Plätzen referenziert werden können sowie weitere Netzinstanzen. All diese Eigenschaften erschweren die Implementation ungemein und führen auch zu einem zusätzlichen Overhead bei jedweder Ausführung der Referenznetzsimulation.

Eine letzte Herausforderung ergibt sich dadurch, dass die Notwendigkeit der Hochskalierung nicht zwangsläufig immer aus dem Zustand der Referenznetzsimulation ablesbar sein muss, welcher sich durch die Gesamtheit aller Marken in allen Platz-

Art der Umsetzung	*Formalismus*	*Funktion*	*Deskriptiv*
Kontrolle über die Skalierung	+	+	++
Nahtlose Eingliederung	-	++	+
Distribute Plugin starten können	n/a	n/a	n/a
Anpassungen am Referenznetzsimulator	-	++	- -
Schlankes und flexibles System	o	+	o
Implementationsaufwand gering	-	+	- -

Tabelle 10.2.: Verschiedene Ansätze fürs Design des Anschlusses an die Simulation

instanzen ergibt. Gerade, wenn berechnungsintensive Bibliotheken in Verwendung sind, kann der Zustand des Netzes kaum von einem nicht ausgelasteten Zustand unterschieden werden, da Berechnungen im Feuern von Transitionen verborgen sind. Der Gesamtansatz, aus Plätzen Skalierungsbedarf abzuleiten, muss daher vorsichtig betrachtet werden. Eine weitere Möglichkeit würde darin bestehen, spezielle dafür vorgesehene Plätze einzuführen, die beispielsweise die Anzahl der Feuervorgänge einer bestimmten Transition in einem gegebenen Zeitrahmen oder aber aufwändige Bibliotheksaufrufe zählen. Der Vorteil der Unsichtbarkeit der Skalierungsdefinition innerhalb des Referenznetzes wäre dadurch allerdings zumindest zum Teil wieder relativiert.

Zuletzt kann argumentiert werden, dass auch die Erweiterung durch Skalierungsdefinitionen eine Formalismusänderung in dem Sinne darstellt, dass Plätzen eine zumindest zum Teil aktive Eigenschaft zugeschrieben wird. Zur Abgrenzung sei daher festgehalten, dass der erste Ansatz, welcher am Anfang dieses Abschnitts vorgestellt wurde, eine Änderung der Syntax innerhalb der Referenznetze bezeichnet.

10.3.3. Auswahl einer geeigneten Umsetzung

Zur Auswahl einer geeigneten Umsetzung sollen erneut die bereits bekannten Anforderungen an die Realisierung herangezogen werden. Analog zur Auswahl einer geeigneten Konstruktion für das Management-System werden die Überlegungen der letzten Abschnitte auch hier in Form von Tabelle 10.2 konsolidiert.

Der erste zu betrachtende Punkt ist die Kontrolle über die Skalierung, die durch die jeweilige Simulationsanbindung bereitgestellt werden kann. Hierbei ist wie zuvor beschrieben die deskriptive Version äußerst stark, da komplexe Zusammenhänge und Konstellationen von Marken zur Manipulation des Skalierungsgrades eingesetzt werden können. Etwas aufwändiger umzusetzen, jedoch ähnlich mächtig, ist die Variante, Funktionen bereitzustellen und die Skalierung an Transitions-

feuervorgänge zu koppeln. Eine Erweiterung des Formalismus wäre auch in der Lage, ein ähnliches Verhalten abzubilden, jedoch wären ggf. mehrere Änderungen dem Formalismus hinzuzufügen, um komplexes Verhalten zu implementieren. Fortgeschrittene Steuerung wie mit der deskriptiven Methode sind jedoch auch hier nicht leicht abbildbar.

Die nahtlose Eingliederung in die bestehende Referenznetzsimulation kann durch die Verwendung von Funktionen am leichtesten erreicht werden. Hierbei werden lediglich weitere Funktionen als Transitionsanschriften eingesetzt. Der Einsatz unterscheidet sich somit nicht von anderen Lösungen mit Referenznetzen. Der Einsatz deskriptiver Dateien würde sich nicht direkt eingliedern, sondern nach Möglichkeit weitestgehend unsichtbar ablaufen. Somit bleibt die Simulation zwar unverändert, streng genommen kann aber auch nicht von einer Eingliederung gesprochen werden. Durch eine Formalismusänderung sind unter Umständen konventionelle (oder Distribute-)Referenznetze nicht mehr direkt einsetzbar im skalierenden Kontext. Somit kann durch eine Formalismusänderung keine wirkliche Integration erzielt werden.

Keine der Änderungen hat Einfluss auf die Startbarkeit weiterer Instanzen von Renew mit Distribute Plugin. Diese Anforderung muss im Management-System umgesetzt werden und ist daher hier nicht anwendbar.

Nachfolgend sollen die notwendigen Änderungen am Referenznetzsimulator diskutiert werden. Es fällt unmittelbar auf, dass die Bereitstellung von Funktionen zur Steuerung der Skalierung keine eingreifenden Änderungen am Simulator erfordern. Lediglich die Implementation der Funktionen ist hinzuzufügen. Anders verhält sich dabei die Einführung eines neuen Formalismus. Hierzu muss ein geeigneter Parser, eine geeignete Syntaxdefinition sowie ein entsprechender Netzcompiler hinzugefügt werden, wie es auch schon für das Distribute-Plugin erfolgt ist. An der Spitze der notwendigen Änderungen steht jedoch die Umsetzung mittels deskriptiver Skalierungsdefinitionen. Neben einem Parser für derartige Dateien muss auch die Verarbeitung der Informationen der aktuellen Markierung im Netzsystem hinzugefügt werden. Um nicht bei jeder Änderung den gesamten Zustand neu scannen zu müssen, sollte lediglich die Änderung zum vergangenen Zustand betrachtet werden. Diese Differenz entspricht genau den konsumierten und produzierten Marken bei einem Feuervorgang. Folglich müsste der grundlegende Simulator entsprechend angepasst werden, um bei jedem Feuervorgang die Auswertung der aktuell gewünschten Skalierung vorzunehmen.

Direkt aus diesem Umstand ergibt sich somit auch, dass das System, welches aus der Umsetzung der deskriptiven Variante resultiert, mit dem Overhead der ständigen Auswertung der Skalierungsinformationen umgehen muss. Dies ist schwierig mit der Anforderung nach einem schlanken System zu vereinbaren, bietet jedoch eine gute Menge Flexibilität in der Formulierung der Anforderungen an die Skalierung. Zusammenfassend ist diese Eigenschaft also als mittelmäßig erfüllt zu

betrachten. Ähnlich hierzu verhält sich der eigene Formalismus, da er das System vermutlich nur wenig erweitern würde. Die transitionsbasierte Umsetzung mittels Funktionen schränkt den Overhead jedoch auf ein Minimum ein. Sie bietet dennoch eine gute Menge Flexibilität, da sie an jeder Stelle in der Simulation (als Funktion) verfügbar ist.

Als letztes Kriterium ist der Implementationsaufwand zu betrachten. Analog zu den vorherigen Ausführungen benötigt die Bereitstellung als Funktion den geringsten Aufwand, indem nur die bereitgestellten Funktionen implementiert werden müssen. Ein eigener Formalismus benötigt wie beschrieben Parser, Syntaxdefinition und Netzcompiler, während die deskriptive Variante ein Umschreiben bzw. Ergänzen des Simulators erfordert.

Abschließend kann festgehalten werden, dass im Kontext der Anforderungen die Umsetzung mittels Funktionsbereitstellung die besten Bewertungen erhält. Daher wird eine Umsetzung mittels Funktionsbereitstellung angestrebt.

Während die Umsetzung als deskriptive Skalierungsdefinition einen nicht zu unterschätzenden Eingriff darstellt und nicht ohne Nachteile ist, ist der Ansatz mächtig und vielversprechend und sollte in jedem Fall Gegenstand weiterer Untersuchungen jenseits dieser Arbeit sein.

10.4. Management-System: Kommunikation zur Simulation

Nachdem in den vergangenen Abschnitten sowohl die Architektur des Management-Systems als auch die Architektur der Simulationsanbindung erörtert wurde, liegt es nahe, ein weiteres zentrales Element zu adressieren: die Kommunikation zwischen diesen beiden Komponenten. Es wurde bereits motiviert, dass einzelne Kommunikationsaktionen zwischen den Systemteilen durch zwei einfache Kommunikationen abbilden lassen. Der Fokus an dieser Stelle soll daher auf zwei Aspekten bzw. Fragestellungen liegen: »Wie werden Daten ausgetauscht?« und »Welche Daten werden ausgetauscht?«. Die Erörterung dieser Fragen nimmt in gewissem Maße einige Inhalte von Kapitel 12 vorweg. Kapitel 12 fokussiert insbesondere jedoch auch (komplexe) Kommunikation zwischen Agenten, deren voller Umfang in diesem Kontext nicht nötig ist.

Die erste Frage lässt sich in ihrer sehr allgemeinen Fassung in der Informatik in Form von Protokollen beantworten (Tanenbaum und Wetherall, 2010). Über die Jahre wurden eine Vielzahl an verschiedenen Lösungen auf verschiedenen Ebenen bzw. Abstraktionsniveaus für die Gestaltung von Kommunikationsprotokollen von der wissenschaftlichen und technischen Gemeinschaft vorge-

schlagen. Eine altbekannte Hierarchie dieser Abstraktionsniveaus beschreibt das OSI-Schichtenmodell[3]. Da der von dieser Arbeit adressierte Forschungsrahmen auf der Anwendungsebene (nach OSI) stattfindet, liegt es nahe, auch nur dort Protokolle zu betrachten. Ziel dieses Abschnitts ist es, ein geeignetes Protokoll auszuwählen, welches eine Kommunikation erlaubt, die bestmöglich die Anforderungen aus Kapitel 9 abbildet.

Ein weiterer Aspekt der Modalitäten des Datenaustausches ist die Form der Datenserialisierung. Auch hierbei existiert eine Vielzahl an verschiedenen Ansätzen. Einige Protokolle erfordern ein spezielles Datenformat während andere freier bei der Auswahl sind. Die verschiedenen Formate wurden mit verschiedenen Zielen implementiert, sodass ein Abgleich dieser Ziele im Rahmen der Kommunikation zwischen den hier beschriebenen Systemkomponenten nötig ist. Auf der hier eingenommenen konzeptuellen Ebene wird in erster Linie die Wichtigkeit der Umsetzung verschiedener wünschenswerter Eigenschaften von Datenformaten diskutiert.

Darüber hinaus ist wie anfangs eingeführt der Aspekt zu erörtern, was Bestandteil der ausgetauschten Daten sein muss und welche Daten gegebenenfalls nicht benötigt werden. Jede Form von unnötig übertragenen Daten und unnötigen Kommunikationen verringern im Zweifel die Performance des Systems. Auch hierbei soll das oberste Ziel die bestmögliche Erfüllung der Anforderungen aus Kapitel 9 sein. Weitere Details zur Frage, welche Daten übertragen werden müssen, diskutiert Abschnitt 10.4.3.

10.4.1. Format der Datenserialisierung

Zunächst soll das Format der Serialisierung betrachtet werden. Bei der Repräsentation von Daten unterscheiden sich die Repräsentationsformen zunächst zwischen Formaten, die für die menschliche Lesbarkeit optimiert sind (meist auf ASCII oder Unicode Basis) und solchen, die für die Lesbarkeit durch Applikationen optimiert sind. Da ein menschlicher Leser im Fall von einfachen Nachrichten wesentlich mehr Informationen und Kontext benötigt, um die übermittelte Nachricht verstehen zu können, liegt es nahe, dass diese Art der Übermittlung aus Performancesicht weniger effizient ist.

Das Gegenstück hierzu wird gemeinhin als binäres Format bezeichnet, wobei auch in binären Formaten mitunter Bruchstücke der Daten durch menschliche Leser verstanden werden können. Dies ist insbesondere der Fall, wenn es sich bei den übertragenen Daten ohnehin um Text handelt. Bei der Gestaltung eines skalierenden Systems entstehen viele komplexe Zusammenhänge der Komponenten. Die Fehlersuche und -behandlung ist in verteilten Systemen ohnehin erschwert, daher

[3] vgl. hierzu beispielsweise (Tanenbaum und Wetherall, 2010)

soll für den Einsatz im Rahmen der Arbeit zunächst ein menschenlesbares Format zum Einsatz kommen. Da Kontext dieser Betrachtung auch nur die Kommunikation zur Steuerung der Simulationsskalierung im Management-System ist, bei der naturgemäß etwas weniger Datenverkehr zu erwarten ist als bei der Simulation selbst, wiegt die Performance-Eigenschaft weniger stark. Es ist denkbar, später in späteren Entwicklungsstadien eine binäre Alternative anzubieten.

Ein weiterer Aspekt bei der Serialisierung beläuft sich auf die mögliche Standardisierung des Formats. Folgt das Datenformat einem Standard, so steigt die Wahrscheinlichkeit, dass einsetzbare Programmbibliotheken existieren und auf eine bestehende und robuste Implementation zurückgegriffen werden kann. In diesem Kontext ist es auch fraglich, ob der Standard oder das Format frei verfügbar oder durch beispielsweise Patente und einschränkende Lizenzen o. Ä. geschützt ist. Da die Verwendung eines Formats mit problematischer Lizenz Schwierigkeiten im Bezug auf eine Veröffentlichung bedeutet, soll dies für den Rahmen der Arbeit nach Möglichkeit vermieden werden. Insgesamt sollte die Verwendung eines Standards angestrebt werden, um unnötige handwerkliche Implementationsarbeit zu reduzieren und Kompatibilität mit Drittsystemen zu gewährleisten.

Zuletzt sind noch einige vorteilhafte Aspekte zu benennen, welche aber nur wenig Auswirkung auf die Wahl des Datenformats haben sollen. Im Kontext von Referenznetzen wäre es hilfreich, wenn das ausgewählte Datenformat ebenfalls das Konzept der Referenzierung unterstützt. Somit könnte eine zu übertragene Datenstruktur originalgetreuer beschrieben werden und gleichzeitig die Menge der zu übertragenen Daten reduziert werden. Da die Übertragung von Referenznetzen allerdings nur beim durchgehenden Einsatz von Referenznetzen in der Simulationsanbindung und dem Management-System erforderlich wäre, ist diese Anforderung gering zu gewichten. Innerhalb der Simulation wäre eine derartige Anforderung jedoch wesentlich interessanter und sollte bei weiteren Arbeiten in der Richtung der Neugestaltung der Verteilung von Referenznetzen in jedem Fall Beachtung finden.

Ein zusätzlicher wünschenswerter Aspekt ist die Möglichkeit, die übertragenen Daten ohne Umstrukturierung, Konvertierung oder Änderung direkt nutzen zu können. In der Literatur wird dies als Zero-Copy[4]-Fähigkeit des Datenformats bezeichnet. Um die Performance des Protokolls weiter zu steigern, ist dies immer ein erstrebenswerter Vorteil.

Zusammengefasst lässt sich somit festhalten, dass für den Einsatz ein Datenformat gesucht wird, welches menschenlesbar, standardisiert und frei verfügbar ist sowie falls möglich Referenzen und Zero-Copy unterstützt.

[4] vgl. beispielsweise (Khalidi und Thadani, 1995), (Chu, 1996) sowie (Kang u. a., 2006)

10.4.2. Kommunikationsprotokolle

Die Auswahl der Kommunikationsprotokolle spielt eine entscheidende Rolle bei der Gestaltung des Gesamtsystems. In Abgrenzung zu den Protokollen der MUSHU- / MULAN-Architektur kommt hier die Bezeichnung »Kommunikationsprotokolle« für Abläufe der Netzwerkkommunikation auf technischer Ebene zum Einsatz. Bei der technischen Umsetzung müssen diese Kommunikationsprotokolle entweder selbst implementiert oder ein geeignetes Framework ausgewählt werden, welches im Idealfall diese Kommunikationsprotokolle nativ unterstützt. Da bereits die Entscheidung gefällt wurde, das Management-System als Webserver zu implementieren, liegt es nahe, ebenfalls ein webbasiertes Kommunikationsprotokoll einzusetzen. Dies passt auch zum generellen Ansatz, das Gesamtsystem als Webservice-Struktur zu interpretieren.

Dennoch sollten bestehende Möglichkeiten abgegrenzt werden, um eine fundierte Auswahl treffen zu können und die Entscheidung zu Webtechnologie zu validieren.

Da eine Vielzahl an Kommunikationsprotokollen existiert, ist ebenfalls ein geeignetes Vorgehen zum Selektieren der als Kandidaten infrage kommenden Kommunikationsprotokolle notwendig. Dabei wird in diesem Abschnitt lediglich auf die konzeptuelle Herangehensweise zur Auswahl des Kommunikationsprotokolls eingegangen und welche Vor- und Nachteile dieses bietet und weniger auf die konkreten Kommunikationsprotokolle. Der Abschnitt dient damit als Vorbereitung auf die finale Auswahl von einem Kommunikationsprotokoll in der technischen Umsetzung, welche in Kapitel 14 beschrieben ist.

Ähnlich wie bei der Konstruktion des Management-Systems ist ein naheliegender Ansatz, verschiedene Ziele als primär zu betrachten und danach ideale Kommunikationsprotokolle vorzustellen. Werden die technischen Aspekte des Systems betrachtet, liegt es nahe, ebenfalls ein technisch verwandtes Format einzusetzen. Im Falle des Simulators RENEW stammen diese in erster Linie aus der Java-Welt.

Ein anderer Ansatz, welcher in einer Linie mit der Auswahl das Management-System als Webservice zu realisieren liegt, ist ein Fokus auf Universalität des Kommunikationsprotokolls. Dies bedeutet, dass das erklärte Ziel eine Kompatibilität mit möglichst vielen anderen ggf. dritten Komponenten ist.

Zuletzt ist wie eingangs erwähnt Effizienz eine wünschenswerte Eigenschaft bei der Wahl des Kommunikationsprotokolls. Ein weiterer Ansatz widmet sich also dem Ziel ein möglichst effizientes Kommunikationsprotokoll auszuwählen, bei dem Daten und Kommunikationsrunden minimiert werden.

Technologisch naheliegendes Format

Der erste Ansatz umfasst den Einsatz eines Kommunikationsprotokolls, welches technologisch möglichst nah an den bereits bestehenden Komponenten liegt. Mit der Programmiersprache Java als Grundlage setzen die sonstigen Teile der Simulation eine imperative, zustandsverändernde Sprache ein. Damit wäre ein direktes technologisch verwandtes Format der Kommunikation der Aufruf von entfernten Methoden. Diese Technologie wird gemeinhin als Remote Procedure Call bzw. Remote Method Invocation bezeichnet und wird so bereits im Distribute Plugin eingesetzt. Für die Kommunikation wäre wie bereits beschrieben eine Registry erforderlich, welche die Aushandlung zwischen lokaler und entfernter Instanz des Programmes übernimmt. Lokal müssten für entfernte Methoden entsprechende Stubs vorbereitet werden, welche dann im Hintergrund auf den entfernten Methodenaufruf abgebildet werden. Die Repräsentation der Daten erfolgt im Normalfall durch geeignete Serialisierungmethoden, welche im Rahmen der Programmiersprache angeboten werden. Die Daten werden somit meist in einem Binärformat verschickt und sind damit zwar kompakt, aber nicht einfach verständlich für menschliche Betrachter.

Die Vorteile bei diesem Ansatz liegen klar darin, dass das Programmierparadigma nicht gewechselt werden muss und lokale wie entfernte Programmfragmente gleich behandelt werden können. Die Integration ist damit aus Sicht der Programmierung und aus Sicht der Simulation sehr hoch. Jedoch liegt die Kehrseite dieses Ansatzes darin, dass eine weitere Komponente in Form der Registry zum System hinzugefügt wird. Da jedoch das Distribute Plugin in seiner klassischen Form bereits auf der Basis der Remote Method Invocation arbeitet, könnte die Registry der Simulation mitverwendet werden. Der Lifecycle der Registry für die Skalierungskontrolle und der Lifecycle der Registry für die Simulation selbst müssen jedoch nicht immer zwangsläufig identisch sein. Durch die Zusammenlegung der Registries würde somit im allgemeinen Fall eine zusätzliche Einschränkung eingeführt werden. Diese Lösung kann dann gut funktionieren, wenn in der betrachteten Gesamtinfrastruktur stets nur *eine* (potenziell sehr große) Simulation ausgeführt wird. Sie kann jedoch zu Problemen führen, wenn mehrere gleichberechtigte Simulationen auf derselben Infrastruktur existieren. Darüber hinaus wird im Rahmen von Kapitel 11 und Kapitel 12 noch motiviert, dass es wünschenswert sein kann, zukünftig das Kommunikationsprotokoll des Distribute Plugins abzulösen. Ein weiterer Einsatz der Remote Method Invocation innerhalb des Gesamtsystems würde dieses Vorhaben jedoch behindern.

Ein weiterer Nachteil der Umsetzung mittels Remote Method Invocation liegt in der Integrierbarkeit mit Komponenten, welche nicht auf der gleichen Programmiersprache basieren. Diese müssten durch aufwändig zu konstruierende Adapter

Zugriff auf die Kommunikation erhalten. Somit wäre eine universelle Erweiterbarkeit im Rahmen weiterer Untersuchungen der Methode deutlich erschwert.

Die technische de facto Empfehlung bei Umsetzung dieses Ansatzes wäre der Einsatz von Java RMI als Kommunikationsprotokoll. Es setzt auf der Serialisierung von Java Objekten auf und verwendet somit ein binäres, Java-eigenes Datenformat.

Unabhängiges Format

Um den Einsatz eines unabhängigen dritten Kommunikationsprotokolls gezielt und ausreichend einfach gestalten zu können, ist in erster Linie wichtig, dass dieses Kommunikationsprotokoll ähnlich wie das Datenformat einem Standard folgt und zusätzlich auch in Form von Programmbibliotheken einfach verfügbar ist. Darüber hinaus ist es ebenfalls wünschenswert, dass keine beschränkende Lizenz oder Patente mit der Nutzung des Kommunikationsprotokolls einhergehen.

Aus Sicht der Anwendung müssen bei dem Kommunikationsprotokoll zwar Daten aber keine großen Datenmengen übertragen werden. Daher ist es nicht notwendig, ein auf Datendurchsatz von großen Datenmengen optimiertes Kommunikationsprotokoll einzusetzen. Die zu ermittelnden Daten werden in Abschnitt 10.4.3 noch einmal im Detail erörtert. Zusätzlich kann es eine wünschenswerte Eigenschaft sein, wenn von dem Kommunikationsprotokoll Datenkompression unterstützt wird.

Weitere Aspekte, die in ihrer Wichtigkeit im Kontext der Arbeit nachgelagert sind, umfassen zusätzlich die Fähigkeit des Kommunikationsprotokolls Verschlüsselung einzusetzen, sowie an mehrere Kommunikationspartner gleichzeitig Nachrichten zu übertragen (Multicasting). Der Sicherheitsaspekt der Anwendung ist gerade bei einer Infrastruktursteuerung hochgradig relevant, jedoch fokussiert sich die Arbeit auf die Realisierbarkeit der Mushu-Architektur und nicht im Speziellen auf den Sicherheitsaspekt. Daher wird für alle Ausführungen eine sichere Ausführungsumgebung angenommen, bei der keine böswilligen Agenten vorhanden sind. Verschlüsselung wäre daher primär für spätere weitere Untersuchungen in Richtung IT-Sicherheit jenseits dieser Arbeit von Interesse.

Die Möglichkeit, an mehrere Kommunikationspartner gleichzeitig Nachrichten zu übertragen, kann beim gleichzeitigen Hochfahren vieler neuer Berechnungsinstanzen von Interesse sein. Jedoch ist dieser Anwendungsfall selten so groß ausgelegt, dass der Einsatz von Multicasting gegenüber dem Versand einzelner Nachrichten wirkliche Vorteile bringen könnte.

Eigenes Format

Der Einsatz eines eigens für diesen Zweck konstruierten Kommunikationsprotokolls weist den Vorteil auf, dass hier notwendige Besonderheiten gezielt in das Kommunikationsprotokoll integriert werden könnten. Die Integration in die für die jeweiligen Systemteile verwendeten Sprachen wäre direkt realisierbar. Die offensichtliche Schwierigkeit bei der Methode liegt jedoch darin, dass das vollständige Design eines eigenen Kommunikationsprotokolls je nach angestrebter Mächtigkeit sehr aufwändig und kompliziert sein kann. Die zentrale Frage, welche hierbei also beantwortet werden muss, ist die, ob das Designs eines eigenen Kommunikationsprotokolls ausreichend viele Vorteile bietet, um den Aufwand zu rechtfertigen.

Eine Beantwortung der Frage hängt maßgeblich davon ab, ob im Prototypen-Kapitel ein Kommunikationsprotokoll gefunden werden kann, welches die Anforderungen an ein unabhängiges Format, wie sie im letzten Abschnitt definiert wurden, ausreichend erfüllen kann. Da der Implementationsaufwand absehbar nicht unerheblich wäre, sollte diese Lösung nur im absoluten und begründeten Ausnahmefall gewählt werden und bedarf einer großen Menge Vorbereitung und Überlegung.

10.4.3. Ausgetauschte Daten

Zum Abschluss der Ausführungen zur Kommunikation zwischen Management-System und Simulationsanbindung soll die Frage erörtert werden, welche Daten zwischen Simulation und Skalierungsmanagement ausgetauscht werden müssen. In erster Linie steht dabei neben den bloßen Informationen über die gewünschte Anzahl an Plattformen die Frage, mit welchen eine konkrete Simulation betreffenden Daten entfernte Instanzen des Simulators Renew ausgestattet werden müssen, um sich als weitere Plattform in die Simulation integrieren zu können.

Die Simulation von Referenznetzen erfordert das Vorhandensein der statischen Netzdaten. Aus der Sicht der Mushu-Architektur handelt es sich dabei um die Agentendefinitionen. Folglich sind die Netze, bei denen die Absicht vorliegt, Netzinstanzen im Laufe der Simulation zu erzeugen, in jedem Fall an das Skalierungsmanagement zu übertragen. Renew bietet die Möglichkeit, eine minimalisierte Variante von diesen Netzen zu generieren, bei denen grafische Repräsentationen ausgenommen sind. Diese minimalen Varianten von Netzen werden als Schattennetze bzw. von ganzen Netzsystemen als »Shadow Net Systems« bezeichnet, wie sie bereits in Abschnitt 2.8.7 eingeführt wurden.

Beim Start einer Referenznetzsimulation erfolgt die Instanziierung einer einzelnen Netzinstanz. Daher muss eine neue Plattform beim Start darüber informiert werden, welches Netz initial mit einer Netzinstanz zu instanziieren ist. In späte-

ren Untersuchungen bezüglich des Plattformdesigns wird sich zeigen, dass diese Anforderung unter bestimmten Bedingungen aufgeweicht werden kann. So ist es durch das Design der Plattform der Mushu-Architektur möglich, auch später weitere Netzdaten nachzuliefern.

Bei diesen Daten (Shadow Net System und initial zu instantiierendes Netz) handelt es sich um die minimal benötigte Menge an Daten, um eine entfernte Referenznetzsimulation zu starten. Die Voraussetzung dabei ist, dass der entfernte Simulator sowohl über eine Einbindung des Distribute Plugins verfügt, als auch eine etwaige physikalische Anbindung an die Skalierungskontrolle selbst.

Darüber hinaus kann in erweiterten Szenarien die entfernte Renew Instanz zusätzlich spezialisierte Plugins und Java Bibliotheken benötigen. Diese Anforderung wird später noch einmal detaillierter im Rahmen der Funktionalitätserweiterung von Plattformen in Kapitel 11 diskutiert. In (Engelhardt, 2020) wurde als Beispiel für eine Funktionalitätserweiterung die Verteilung des Analysewerkzeugs »MoMoC« (Willrodt, 2019) unter dem Namen »Distributed Analysis« in dem zum hier beschriebenen System gehörenden Prototypen realisiert.

10.5. Management-System: Überlegungen im Kontext des Gesamtsystems

Dieses Kapitel dient der Vorstellung von Überlegungen, welche erst im Bilde der Gesamtarchitektur der Steuerung der Simulationsumgebung sinnvoll adressiert werden können. Dabei handelt es sich beispielsweise um die Frage nach einer grafischen Oberfläche für das System, die Frage, welche Systemkomponenten eine Skalierung vornehmen dürfen und auch wie mehrere nebenläufige Simulationen im Gesamtsystem gehandhabt werden.

10.5.1. Grafische Oberflächen

Verschiedene der vorgestellten Komponenten könnten potenziell mit grafischen Oberflächen versehen werden. Eine Oberfläche des Management-Systems könnte einfache Auskünfte über die Anzahl an Plattformen ermöglichen. Da die Skalierungskontrolle automatisiert durch eine laufende Simulation erfolgt, hätte eine derartige grafische Oberfläche ausschließlich informativen Charakter und könnte eingesetzt werden, um Fehler im Modell und ungewolltes Verhalten besser identifizieren zu können. Zu den dargestellten Daten könnte unter anderem die Menge an aktuell in der Benutzung befindlichen Knoten zählen, wie viele Knoten in welcher Zeitspanne hinzugeschaltet wurden und wie viel Datenverkehr zwischen den einzelnen Knoten besteht.

Da wie in den ersten Abschnitten dieses Kapitels vorgestellt der Einsatz eines Clustermanagers beabsichtigt ist, kann der Grenznutzen der Implementation einer eigenen grafischen Oberfläche allerdings als überschaubar angenommen werden. Viele Clustermanager besitzen größere Projekte der Community, mit denen eine grafische Oberfläche zur Verfügung gestellt werden kann. Die Bereitstellung einer derartigen Oberfläche soll daher der Entscheidung des Administrators obliegen, welcher für die Installation des Clustermanagers zuständig ist.

Ein anderer Ansatz ist eine Oberfläche für die Simulation selbst. Diese könnte die einzelnen Feuervorgänge innerhalb der Simulation darstellen und die Nachvollziehbarkeit der verteilten Simulation verbessern. Der Simulator Renew verfügt über eine mitgelieferte grafische Oberfläche auf Java-Basis. Diese ist im aktuellen Zustand relativ eng an viele Komponenten gekoppelt und dadurch nur schwer alleinstehend bzw. entkoppelt ausführbar. Die Idealvorstellung für eine verteilte Simulation wäre ein globaler Simulationsfeed, welcher durch eine unabhängige grafische Oberfläche an beliebiger Stelle interpretiert werden könnte. Während es im Rahmen der Modularisierung von Renew laufende Bestrebungen gibt, die grafische Oberfläche vom Simulationskern zu entkoppeln, wurde eine derartige unabhängige grafische Oberfläche jedoch noch nicht entwickelt. Eine erste Entwicklung in diese Richtung bietet die webbasierte Oberfläche, welche in (Kilian, 2019) und (Richter, 2019) vorgestellt wurde. Diese Oberfläche ist jedoch nicht an eine laufende Simulation anschließbar, da auf der Seite des Simulationskerns die entsprechenden Möglichkeiten (beispielsweise in der Form eines webbasierten Simulationsfeeds) nicht gegeben sind.

In Abschnitt 11.6 wird auf eben diese Erstellung eines Simulationsfeeds von Seiten der Simulation umfangreich eingegangen. Aus diesem Grund wird die detaillierte Umsetzung an dieser Stelle nicht weiter verfolgt, sondern im Folgenden eine einfache, umsetzbare Alternativlösung skizziert, die jedoch nur begrenzte Einsicht in das System erlaubt und einigen Einschränkungen unterliegt.

Ausgehend von der Annahme, dass eine (verteilte) Referenznetzsimulation immer aus mindestens einem Knoten besteht, ist es vertretbar, einen primären Knoten in der Simulation zu benennen. Demnach liegt es nahe, dem primären Knoten ebenfalls eine grafische Oberfläche zuzugestehen, damit ein etwaiger Modellierer (oder Nutzer) an einer festen Stelle Einsicht in das laufende System erhalten kann. Neue, der Simulation hinzugefügte Knoten entstehen dynamisch durch die Simulation selbst, je nach simuliertem Inhalt in höherer oder niedrigerer Anzahl.

Auf diese Weise könnte auf einer Workstation eine mit grafischer Oberfläche ausgestattete Instanz des Simulators gestartet werden, dort eine Simulation initiiert werden, welche dann via Management-System weitere Knoten ohne grafische Oberfläche startet und sich mit ihnen verbindet. Ein Nachteil hierbei ist selbstverständlich, dass so nur Netzinstanzen auf dem lokalen Knoten inspiziert werden können und keine Einsicht in die Zustände der entfernten Netzinstanzen genom-

men werden kann. Dieser Umstand ist unschön, jedoch in dieser Implementation ab einer gewissen Skalierungsgröße unvermeidlich, da sonst der Knoten mit grafischer Oberfläche zu einem Flaschenhals werden würde und die Skalierung insgesamt relativieren könnte.

In späteren Überlegungen kann es sinnvoll sein, dass der primäre Knoten mit der Zeit wechselt. Dies kann insbesondere bei langlaufenden Simulationen Sinn ergeben. Da die isolierte Ausführung mit bereitgestelltem Simulationsfeed wie oben beschrieben (noch) nicht möglich ist, soll diese Variante für spätere Betrachtung vorbehalten bleiben.

10.5.2. Datenhaltung in der Skalierungskontrolle

Ein zu adressierender Punkt ist die Datenhaltung im Kontext der Skalierungskontrolle. Dazu ist zunächst zu erörtern, welche Form von Daten gesichert werden muss. Da die Skalierung dem Reconciler Pattern folgen soll, ist es in erster Linie wichtig die »Soll« und »Ist«-Größen der gewünschten Plattformen vorzuhalten. Darüber hinaus müssen die lokalen Adressdaten der adressierten (physikalischen) Plattformen gesichert werden, damit Teile der Simulationen andere Teile erreichen können. Bezugspunkt sind dabei die reinen Informationen zum physikalischen Routing, nicht etwa ein Auffinden bzw. Namensauflösung von Plattformen, welches u.A. im Rahmen der Agentenkommunikation in Kapitel 12 aufgearbeitet werden wird. Zusätzlich müssen für eine laufende Simulation Daten wie die eingesetzten Netze, das initial zu instanziierende Netz und ggf. Plugins und Bibliotheken vorgehalten werden. Darüber hinaus müssen etwaige Zugangsberechtigungen für die Steuerung der lokalen Infrastruktur vorgehalten werden.

Da das zentrale Management-System als zustandsloser Webservice realisiert werden soll, können keine veränderlichen Daten an dieser Stelle vorgehalten werden. Da einige Daten direkt im Zusammenhang mit dem Zustand der Infrastruktur stehen, können und sollten sie nicht rein auf Clientseite –in diesem Fall in der Simulation –vorgehalten werden. Durch den Einsatz eines Clustermanagers können viele Daten direkt live aus der Infrastruktur abgefragt werden.

Die Bereitstellung der Daten zur Einbindung neuer Plattformen sind spezifisch für eine bestimmte laufende Simulation und müssen beim Start von neuen Plattformen diesen bereitgestellt werden können. Bei diesen simulationsspezifischen Daten handelt es sich weder um infrastrukturspezifische Daten, noch um Daten, die von einem einzelnen Client verantwortet werden. Folglich wäre die Speicherung innerhalb des Management-Systems eine naheliegende Lösungsmöglichkeit. Dem entgegen steht jedoch die angestrebte Zustandslosigkeit des Management-Systems. Dazu gibt es zwei denkbare Lösungen: Zum einen könnte die Anforderung von Zustandslosigkeit aufgeweicht werden und nur im Kontext einer laufen-

den Simulation gefordert werden. Dabei würde die Information, wie zusätzliche Plattformen in die Simulation einzubinden sind, als für den gesamten Lebenszyklus des Management-Systems konstante Information angesehen werden. Damit könnte innerhalb einer laufenden Simulation die Semantik der Zustandslosigkeit erreicht werden.

Die andere Lösungsstrategie würde den Einsatz einer dedizierten Datenbank hinter dem Management-System erfordern, sodass die zustandsbehafteten Daten ausgelagert werden. Diese Variante hätte in der Praxis Stabilitätsvorteile, würde jedoch die Komplexität der Anwendung weiter steigern. Die speziellen Stabilitätsvorteile werden im folgenden Abschnitt noch einmal dedizierter erläutert und in den Kontext weiterer Risiken eingegliedert.

10.5.3. Risiken und Ausfälle

Ein weiterer zu adressierender Punkt bezieht sich auf die Stabilität und den Umgang mit Ausfällen. Da es sich bei der skalierenden Referenznetzsimulation insbesondere auch um ein verteiltes System handelt, sollten in jeder Systembetrachtung auch Ausfälle thematisiert werden. Dabei kann es sich um verschiedene Arten der Ausfälle handeln, wie beispielsweise der Absturz eines der am System beteiligten Prozesse, dem Ausfall einer physikalischen Netzwerkverbindung oder aber der fehlerhafte Zustand einer Komponente. Die Idealvorstellung einer verteilten Referenznetzsimulation umfasst die vollständige Dezentralisierung des Systems, bei der jeder Knoten eine gleichberechtigte Rolle innehat und keine zentralen Steuerungskomponenten existieren. Dieser Aufbau wäre ebenfalls für die Skalierbarkeit des Systems ideal, sofern nicht einzelne Komponenten feste, nicht verschiebbare Aufgaben haben. Aus den vergangenen Ausführungen ergibt sich jedoch, dass das System diesem Ideal nicht immer gerecht werden kann, um eine generelle Realisierbarkeit zu erhalten.

Das Management-System ist als eine zentralisierte Komponente konstruiert und sein Ausfall besitzt daher naturgemäß ein hohes Risiko, das Gesamtsystem zum Erliegen zu bringen. Oberstes Ziel sollte in jedem verteilten System stets die Verhinderung von kaskadierenden Fehlschlägen sein, bei dem der Ausfall einer Komponente den Ausfall weiterer Komponenten bewirkt. Dies wurde im Rahmen der Cloud-Nativity in Abschnitt 6.1.5 unter dem Aspekt der Abhärtung angesprochen. Diese Eigenschaft wird später im Rahmen der Plattformrealisierung in Kapitel 11 noch ausführlicher diskutiert. Die Anbindung an das Management-System erfolgt durch ein zustandsloses Protokoll und da das Management-System selbst ebenfalls zustandslos umgesetzt wird, können und müssen die Simulationsanbindungen entsprechend robust gegenüber Ausfällen des Management-Systems implementiert sein.

Noch kritischer als das Management-System selbst ist der Einsatz der Java RMI Registry durch das Distribute Plugin, da diese die Information und Aushandlung aller an der Simulation beteiligten Komponenten an einem zentralen Ort bündelt. Darüber hinaus ist sie zustandsbehaftet und das Distribute Plugin baut die Verbindung bereits beim Start des Renew Simulators und nicht erst beim Start der Simulation selbst auf. Dies hat zur Folge, dass ein Ausfall der Java RMI Registry empfindliche Konsequenzen für den weiteren Verlauf der Simulation hat und fast sicher zu ihrem Erliegen bzw. Fehlschlagen führen wird. Durch die zustandsbehaftete Implementation seitens Java kann die Registry auch nicht einfach hochskaliert oder in sonstiger Art und Weise redundant betrieben werden. Eine mögliche Lösung wurde jedoch in (Baratloo u. a., 1998) beschrieben in Form einer replizierten Registry für Java RMI.

Mit viel Aufwand wäre es denkbar bzw. möglich, ständige Backups und Speicherabbilder der Registry zu extrahieren und im Problemfall eine andere Registry darauf aufsetzen zu lassen. Da dies allerdings bei jeder Änderung erfolgen müsste, wären die Performanceeinbuße gravierend und der Implementationsaufwand immens.

Eine Abkehr vom Distribute Plugin als Grundlage der Kommunikation hätte eine Neuimplementation der Verteilung der Referenznetzsimulation selbst zur Folge und würde ein völlig neues Feld der Untersuchungen aufschließen. Um wirkliche Fehlertoleranz zu erzielen wäre dies jedoch vermutlich notwendig. Eine andere und weniger drastische Variante wäre der Austausch der Java RMI Registry durch eine robustere Komponente, während dabei jedoch der implementierte Algorithmus von Distribute im Wesentlichen erhalten bleibt. Grundlegende konzeptuelle Überlegungen zu diesem Vorhaben finden sich im Abschnitt zur Realisierung der Agentenkommunikation in Kapitel 12.

10.5.4. Handhabung mehrerer Simulationen

Bei der Betrachtung der bisher vorgestellten Komponenten der Gesamtarchitektur fällt auf, dass in jeder Überlegung lediglich von der Existenz einer einzelnen Simulation ausgegangen wurde. Jedoch ist die Infrastruktur grundlegend in der Lage, potenziell mehrere nebenläufige Systeme bzw. Simulationen gleichzeitig auszuführen. Auf der architekturellen Ebene müssten diese jedoch alle die Kontrolle über dieselbe Infrastruktur innehaben. Die Situation erinnert an eine objektorientierte Klasse, deren Instanzen alle auf dieselbe statische Infrastruktur zugreifen und dadurch in Konflikt geraten. Dieser Umstand wird daher an dieser Stelle im Kontext der Gesamtarchitektur der (reaktiven) Skalierung gesondert adressiert.

Eine Referenznetzsimulation selbst ist ein hoch nebenläufiges System, sofern auf Elemente verzichtet wird, die einen globalen Zustand abfragen müssen, wie bei-

spielsweise Inhibitorkanten. Dadurch kann zunächst argumentiert werden, dass für eine einzelne Simulation mittels Renew mehrere Instanzen von Renew auf demselben physikalischen Knoten nur begrenzt Nutzen generieren. Bei mehreren Simulationen hingegen sollte in jedem Fall die Möglichkeit eingeräumt werden, eine weitere Plattform auf demselben physikalischen Rechnerknoten hochzufahren, um etwaige Leerläufe gering halten zu können. Dabei können jedoch allerlei Arten von Konflikten entstehen, welche in erster Linie auf der technischen Ebene zu verorten sind, wie beispielsweise Portkonflikte, Zuordnungsprobleme oder Deadlocks im Zuge des Zugriffs auf kritische Infrastrukturteile.

Ein möglicher Ansatz, um diese Problematik zu lösen, wäre es den Simulationen jeweils Wissen voneinander zuzugestehen. Die Simulationen könnten somit eine gemeinsame Nutzung der Infrastruktur aushandeln und so meist friedlich koexistieren. Dabei erhält jede Simulation jedoch eine erhöhte Menge an Komplexität und je nach Menge der vorhandenen Simulationen, könnte der Overhead beträchtlich ausfallen.

Da die Probleme wie beschrieben primär auf technischer Ebene auftreten sollten, liegt es nahe, die Form des Deployments der Plattformen in die Betrachtung einzubeziehen. Die Problematik von potenziell in Konflikt stehenden Applikationen ist nicht spezifisch für Referenznetzsimulationen. Daher wurden in der Vergangenheit vielfältige konzeptuelle und technische Lösungen konstruiert und eingesetzt. Ein Hauptansatz dabei umfasst die Virtualisierung, deren Ziel unter diesem Gesichtspunkt ist, jedem Prozess zumindest die Illusion eines eigenen nur für ihn geschaffenen Hostsystems zu geben.

Abschnitt 2.6.4 im Grundlagenkapitel hat das Grundprinzip der Virtualisierung und Containerisierung eingeführt, sodass die vorgestellten Technologien und Konzepte hier Anwendung finden können. Da die Details zu diesen Aspekten maßgeblich zur Konstruktion der im Rahmen der Arbeit vorgestellten Lösung beitragen, ist ihnen der eigene Abschnitt 10.6 zum Deployment gewidmet.

10.6. Update der Plattformdefinition und Deployment

Es kann argumentiert werden, dass das Deployment der Einzelkomponenten spezifisch für die Plattformen und die Komponenten des Plattformmanagements ist. Da jedoch eine dynamische Bereitstellung von Plattformen durch das Plattformmanagement angestrebt wird, muss das Deploymentkonzept zwangsläufig im Rahmen des Plattformmanagements als Gesamteinheit diskutiert werden.

Wie bereits im Abschluss von Abschnitt 10.5.4 zur Handhabung mehrerer Simulationen motiviert wurde, ergeben sich durch die Form des Deployments der Software für die Simulationsumgebungssteuerung und der darin ablaufenden Simulation selbst verschiedene Chancen und Möglichkeiten für die Gestaltung.

Dabei wird durch die Form des Deployments sowohl die Art und Weise bezeichnet, wie die Software von Quellcode bis zur aktiven Laufzeitumgebung gebracht wird, als auch die eingesetzte Technologie zur Strukturierung der Laufzeitumgebung. Wie bereits im Grundlagenkapitel erläutert, haben die Grundlagen des Deployments in den 2010er Jahren einen grundlegenden Wandel durchlaufen. Ein Hauptaspekt ist dabei der Fokus auf Virtualisierungslösungen, welche zwar selbst eine gewisse Menge an Overhead mit sich bringen, jedoch im Kern zwei Probleme adressieren: zum einen die Problematik der benötigten Komponenten der Laufzeitumgebung (beispielsweise verwendete Bibliotheken) zum anderen die Portabilität der Anwendung.

Zur Milderung des benötigten Overheads, der hauptsächlich in der Vorbereitung der Umgebung innerhalb der virtualisierten Lösung besteht, haben sich die Ansätze der Continuous Integration (CI) und Continuous Delivery / Continuous Deployment (CD) etabliert. Im Kern dieser Konzepte steht die Verwendung automatisierter Skripts zum Erzeugen von Laufzeitumgebungen, welche auf die individuelle Software angepasst sind. Dies ist eine Umkehr des klassischen Ansatzes, bei dem die Software an ihre Umgebung angepasst wird. Die dabei anvisierte Laufzeitumgebung muss nicht zwangsläufig virtualisiert sein, die virtuelle Form ist jedoch in heutiger Zeit sehr oft anzutreffen. Ein weiterer Aspekt bei CI/CD umfasst die Geschwindigkeit des Deployments durch Entwicklungsteams, da das Deployment weitestgehend automatisiert abläuft. Während dieser Aspekt in profitorientierten Einsatzbereichen maßgeblich ist, spielt er bei den Proof-of-Concept-Implementationen im Rahmen dieser Arbeit eine untergeordnete – aber nicht irrelevante – Rolle.

Im Kontext der Ausführungsumgebung einer skalierenden Referenznetzsimulation gilt es somit zu untersuchen, ob der Einsatz von CI/CD und Virtualisierung den benötigten Mehraufwand für ihre Implementation rechtfertigen. Dabei ergeben sich gewisse Überschneidungen zum Agilitätsaspekt der Mushu-Plattform, wie sie bereits in Abschnitt 6.1.4 aufgeführt wurden. Dabei soll zunächst auf klassisches Deployment per Hand eingegangen werden, dann auf Deployments mittels Konfigurationswerkzeugen und schlussendlich mittels Virtualisierung bzw. Containerisierung. Abschließend werden darüber hinaus noch die Verwendung und der Einsatz von CI/CD adressiert.

10.6.1. Vorüberlegungen zu Deploymentformen

Das Ziel der folgenden Abschnitte liegt darin, die verschiedenen Kombinationsmöglichkeiten von Systemaspekten mit Deploymentformen zu erörtern und zu bewerten.

Die Systembestandteile rund um das Management-System wurden in den vorangegangenen Abschnitten dargelegt und zählen direkt zu den deploybaren Systemaspekten. Die vorgestellten Systembestandteile umfassen ein Kommunikationsmedium, wie beispielsweise die *RMI Registry*, das *Management-System* selbst und den *Clustermanager*. Darüber hinaus wird eine exemplarische *GUI Komponente* als Systembestandteil betrachtet, welche keine weitere Besonderheit aufweist, außer, dass sie in der Lage ist, den für sie lokalen Ausschnitt der Gesamtsimulation darzustellen. Zusätzlich betrachtet wird der Hauptbeitrag der Bereitstellung *zusätzlicher Simulationsinstanzen*, welche durch das Management-System programmatisch hochgefahren werden können.

Die betrachteten Deploymentformen umfassen *vollständig manuelles* Deployment, Deployment in *automatisch konfigurierte Umgebungen*, Deployment durch *Virtualisierung* und Deployment durch *Containerisierung*.

Darüber hinaus werden zusätzlich zu jeder Möglichkeit der Kombination verschiedene Bewertungs- bzw. Einordnungskriterien vorgestellt, um eine Vergleichbarkeit der einzelnen Aspekte und Deploymentformen zu gewährleisten. Dabei soll pro Aspekt/Deploymentform-Kombination sowohl der *Vorabaufwand* als auch die *dynamischen Möglichkeiten* untersucht werden. Dynamische Möglichkeiten beschreiben hier den Grad der Umsetzbarkeit der Skalierbarkeit durch die Gesamtsimulation bei einer gegebenen Deploymentform für einen Aspekt. Der Vorabaufwand bezieht sich im Wesentlichen auf die notwendige programmatische Vorbereitung für einen reibungslosen Ablauf und ist mit dem Grad der Notwendigkeit des Einsatzes von CI/CD verknüpft, welcher detaillierter in Abschnitt 10.6.7 diskutiert wird.

Darüber hinaus bietet sich als relevanter Aspekt pro Deploymentform die *Zeit* an, welche *zum Hinzuschalten weiterer Simulationsinstanzen* benötigt wird. Als letzter betrachteter Aspekt soll der *Grad der Bindung an konkrete bzw. spezielle Infrastruktur* durch die Deploymentform dienen. Zusätzlich spielt der *Grad der Isolation* der Simulation eine wichtige Rolle, wie am Ende von Abschnitt 10.5.4 erläutert wurde. Zuletzt ist ein entscheidender Aspekt, wie viel (für die Referenznetzsimulation) *individueller Aufwand auf neuen physikalischen Knoten* betrieben werden muss, um sie als Ziel potentieller Simulationserweiterungen nutzen zu können.

Die folgenden Abschnitte sind nach Deploymentformen organisiert. Dabei ist explizit nicht jeder der Abschnitte als umfassende Alternative zu den jeweils anderen

Kriterium	*Vorabaufwand*	*Möglichkeiten*
Kommunikationsmedium	+	o
Management-System	++	+
Clustermanager	o	++
GUI Komponente	+	+
Zusätzliche Sim.instanzen	- -	-

Tabelle 10.3.: Bewertung von manuellem Deployment

zu interpretieren. Jeder Abschnitt vermittelt eine Einsicht dazu, welche konkreten Auswirkungen eine Umsetzung des jeweiligen Aspekts mit dem jeweiligen Ansatz hätte. Schlussendlich kann sich jedoch eine Mischform verschiedener Deploymentformen als am vorteilhaftesten herauskristallisieren.

Insgesamt ist ein deutlicher Vorteil durch komplexere Deploymentformen an Stellen des Systems zu erwarten, welche dynamisch skaliert werden sollen. Die Möglichkeiten der singulären oder gering skalierten Komponenten sollten sich als weitestgehend unabhängig von der Form des Deployments erweisen.

10.6.2. Volles manuelles Deployment

Ein volles manuelles Deployment stellt sich als die unkomplizierteste Variante der möglichen Deploymentformen dar. Auf jedem physikalischen Knoten werden die Ausführungsumgebung, benötigte Bibliotheken wie beispielsweise die Java JVM, gegebenenfalls benötigte Startskripte und weiteres per Hand installiert. Dabei fällt sofort auf, dass bei dieser Methode kein weiterer Implementationsaufwand notwendig ist, was sich günstig auf den Vorabaufwand auswirkt im Falle jeder einzelnen Komponente. Für die Komponenten der verteilten Referenznetzsimulation stellt sich die Einbindung und der Start wie in der bestehenden Literatur dar. Daher sei an dieser Stelle auf individuelle Beispiele verzichtet und auf die entsprechende Literatur wie beispielsweise (M. Simon und Moldt, 2016) verwiesen.

Bezogen auf das Kommunikationsmedium (beispielsweise RMI-Registry) ist die Vorbereitung der benötigten Startparameter notwendig. Beim Deployment selbst müssen somit die entsprechenden Parameter hinzugefügt werden, um das Kommunikationsmedium im lokalen Infrastrukturkontext starten zu können. Die manuelle Deploymentform ist ebenfalls die einzige, welche vor den Ergebnissen dieser Arbeit so bereits existierte. Die Möglichkeiten durch den Einsatz einer RMI Registry sind auf natürliche Weise eingeschränkt durch die zustandsbehaftete und zentralisierte Implementation von RMI.

Das Management-System gestaltet sich in seinem Deployment hingegen wesentlich einfacher, da keine spezifischen Parameter bereitgestellt bzw. herausgefunden werden müssen. Seine Möglichkeiten können trotz manuellem Deployment weitestgehend angeboten werden.

Die Installation eines Clustermanagers ist nie ein triviales Unterfangen, da Clustermanager im Normalfall tief in jeweils lokale Systeme integriert werden. Falls dieser manuell deployed wird, sind im Zweifel viele Systemanpassungen und manuelle Schritte bzw. Installationsskripte notwendig. Da ein Clustermanager üblicherweise die Steuerung der konkreten Infrastruktur innehat und eine Abstraktionsschicht zur konkreten Ausprägung der Infrastruktur bildet, kann dieser seine Möglichkeiten maximal entfalten, wenn ein komplexes (erneut selbst abstrahierendes) Deployment entfällt und die Installation nativ auf der Infrastruktur erfolgt. Darüber hinaus bezieht sich ein Clustermanager immer direkt auf eine statische Infrastruktur, weshalb ein mehrfaches Vorhalten eines Clustermanagers nicht sinnvoll ist. Dies betrifft jedoch nicht eine etwaige Redundanz und damit Ausfallsicherheit innerhalb eines deployten Clustermanagers.

In der illustrierten Variante der Steuerung der Ausführungsumgebung bildet die GUI Komponente die primäre Simulationsinstanz. Durch diese Eigenschaft erwächst keine Notwendigkeit eines dynamischen Deployments und somit kann sie problemlos manuell deployed werden, ohne dabei nennenswert an Möglichkeiten einzubüßen. Der Vorabaufwand umfasst die Bereitstellung des Simulators (in diesem Fall Renew), der Installation einer JVM, sowie die Bereitstellung aller benötigter Bibliotheken und Plugins.

Das Starten zusätzlicher Plattformen benötigt eine erhöhte Menge an Vorabaufwand. Alle Komponenten, welche bereits für das Starten der Plattform mit GUI Anbindung notwendig waren, müssen ebenfalls für zusätzliche Plattformen verfügbar sein. Entlang der Mushu-Architektur bezieht sich dies jedoch nur auf Basiskomponenten (die Plattformdefinition), da Plattformfunktionalitäten auch dynamisch zur Laufzeit nachgerüstet werden können.

Dennoch sind die dynamischen Möglichkeiten an dieser Stelle im Falle des manuellen Deployment sehr eingeschränkt. Die konkreten Binaries, Laufzeitumgebungen und die Bibliotheken müssten fest auf dem physikalischen Rechner installiert werden. Es kann nicht davon ausgegangen werden, dass eine solche Umgebung beispielsweise in kommerziellen Lösungen, auf welche die Simulation skaliert werden könnte, direkt anzutreffen ist. Mit einer händischen Installation ist eine umfangreiche Skalierbarkeit auf viele Instanzen somit nur mit nicht zu vernachlässigendem Vorabaufwand möglich und daher unhandlich bis nicht realisierbar.

Zusätzlich ist der Grad der Bindung an die konkrete Infrastruktur sehr hoch, da eben nur auf den vorbereiteten physikalischen Knoten Simulationen ausgeführt

Kriterium	*Vorabaufwand*	*Möglichkeiten*
Kommunikationsmedium	o	o
Management-System	o/+	+
Clustermanager	-	++
GUI Komponente	o	+
Zusätzliche Sim.instanzen	o	o

Tabelle 10.4.: Bewertung von Deployment mit automatischer Konfiguration

werden können. Beim Ausfall eines Knotens, müsste ein Ersatzknoten entweder bereits vorbereitet sein oder aber aufwändig per Hand nachgerüstet werden.

Darüber hinaus ist der Grad der Isolation der einzelnen Prozesse auf den Knoten sehr eingeschränkt, da keine Kontrolle darüber besteht, welche Prozesse welche anderen Prozesse auf demselben physikalischen Knoten beeinträchtigen können. Somit kann es auf einfache und häufige Weise zu Portkonflikten oder Problemen mit Datensicherheit und Ähnlichem kommen.

Auf der positiven Seite ist jedoch zu vermerken, dass das Hinzuschalten einer weiteren Simulationsinstanz auf einem bereits vorbereiteten und laufenden Knoten sehr wenig Zeit in Anspruch nimmt, da praktisch keinerlei Overhead zum Starten der Instanz erforderlich ist.

Die Ergebnisse bezüglich der einzelnen Systemkomponenten sind in Tabelle 10.3 zusammengefasst, während die globalen Ergebnisse gesammelt in Tabelle 10.7 zu finden sind.

10.6.3. Automatische Konfiguration

Der nächste zu untersuchende Ansatz für die Wahl der Deploymentform ist die Einrichtung automatischer Konfigurationsskripte. Dieser Ansatz wurde im Grundlagenkapitel unter 2.6.4 eingeführt. Es existieren vielfältige technische Lösungen aus dem professionellen Umfeld der Serveradministration. Anstatt jedoch auf eine konkrete Umsetzung mit einer dieser Lösung einzugehen, soll auch dieser Abschnitt den Einsatz einer automatischen Konfiguration auf einer konzeptuellen Ebene diskutieren.

Der Kern beim Einsatz automatischer Konfigurationsskripts und -tools besteht darin, einen neuen, bisher unbenutzten Server auf automatisierte Weise mit der benötigten Laufzeitumgebung für eine konkrete Software auszustatten. Dabei kann diese automatische Konfiguration sowohl auf klassischen Servern eingesetzt werden als auch auf Workstations, welche von Nutzern direkt verwendet werden.

Automatische Konfiguration adressiert somit direkt einige Probleme, welche im vergangenen Abschnitt genannt wurden. Das Ziel dabei ist vor allem die Reduktion von manuellem Aufwand bei der Bereitstellung neuer Rechnerknoten.

Insgesamt muss dafür bei allen Systemkomponenten die benötigte Umgebung analysiert werden und mithilfe eines Skripts diese dann automatisch auf neuen Instanzen bereitgestellt werden. Dabei handelt es sich in erster Linie um eine Vorbereitung. Die Ausführung selbst weicht nicht grundlegend vom manuellen Deployment ab.

Aus diesem Grund gestalten sich die dynamischen Möglichkeiten des Kommunikationsmediums, dem Management-System, dem Clustermanager und der GUI Komponente kaum anders als bei der manuellen Variante.

Die Bereitstellung von RMI Registry und GUI Komponente benötigen jeweils eine Laufzeitumgebung für den Simulator (hier Renew mit Distribute Plugin). Dafür ist die Java JVM, sowie Renew selbst samt Plugins auf den Zielsystemen zu installieren. Eine Bereitstellung der Binärdaten im (Netzwerk-)Zugriff der neu aufzusetzenden Server ist somit notwendig. Dies erfordert einige manuelle Vorarbeit, sodass der gesamte Vorabaufwand für das Kommunikationsmedium und die GUI Komponente als mittelmäßig einzustufen ist. Aus konzeptueller Sicht entspricht die Bereitstellung der Binärdaten der Plattformdefinition des Mushu-Plattformmanagements.

Zusätzlich besteht der Aufwand der Bereitstellung der Laufzeitumgebungskomponenten für das Management-System sowie das Management-Systems selbst. Ferner muss die Anbindung an den Clustermanager bedacht werden. Der Vorabaufwand ist daher ähnlich einzuschätzen wie bei Kommunikationsmedium und GUI Komponente.

Deutlich aufwändiger fällt die Vorbereitung jedoch bei der Installation eines Clustermanagers aus. Wie bereits an anderer Stelle erwähnt sind Clustermanager sehr komplexe Systeme, welche die konkrete Infrastruktur abstrahieren können. Eine automatisierte Konfiguration muss somit zumindest immer einen Teil der Infrastruktur kennen, auf welcher der Clustermanager installiert werden soll. Wenigstens die Zahl der physikalischen Knoten, die so vorbereitet werden sollen, müsste bekannt sein.

Da der Clustermanager stets nur einmal pro Infrastruktursystem vorhanden sein muss, ist es wahrscheinlich, dass die Vorbereitung automatisierter Installationsskripte den Aufwand der manuellen Installation des Clustermanagers übersteigt. Die Installationsskripte können direkt auf der Infrastruktur ausgeführt werden. Dadurch entsteht keine weitere Komplexität der Installationsskripte, wie sie beispielsweise durch eine Indirektion wie bei der Virtualisierung auftreten würde. Somit kann der Aufwand noch etwas im Rahmen gehalten werden. Anders gestaltet es sich bei den in den folgenden Abschnitten beschriebenen Deploymentformen.

Lediglich das Hinzuschalten weiterer Instanzen zur Simulation kann auf ganzer Linie vom Einsatz automatischer Konfiguration profitieren. Die Grundumgebungen, welche für jede weitere Instanz der Simulation bereitstehen müssen, ähneln sich jeweils sehr. Somit muss ein Renew Simulator und Java JVM zugegen sein und die benötigen Bibliotheken bereitstehen. Während bei einer manuellen Installation für jeden neu zu akquirierenden Knoten manuelle und damit extrem langsame und teure Arbeit notwendig ist, kann durch eine automatische Konfiguration diese weitestgehend entfallen.

Dabei ist jedoch nach wie vor der Installationsschritt vor dem Hochfahren der neuen Simulationsinstanz notwendig. Dieser Umstand kann etwas abgemildert werden, indem dem System bekannt gemacht wird, auf welchen Knoten bereits die notwendige Umgebung installiert ist. Auf diesen Systemen kann die Installation logischerweise entfallen.

Durch die volle Automatisierung entsteht eine ganz neue Dimension der dynamischen Möglichkeiten in Bezug auf die Simulationsgröße. Dass die volle Automatisierung dennoch nur mit einer mittelmäßigen Wertung versehen werden kann, ist in erster Linie den Möglichkeiten der noch folgenden Deploymentformen geschuldet. Die Wertung resultiert ebenfalls aus der nun folgenden Betrachtung der globalen Aspekte der Deploymentform mit automatischer Konfiguration.

Die komponentenorientierten Ergebnisse sind abschließend noch einmal in Tabelle 10.4 zusammengefasst.

Bei der Betrachtung der globalen Aspekte fällt sofort die deutliche Verbesserung im individuellen Aufwand auf. Statt manuellen Eingriffen sorgen automatische Konfigurationsskripte für die programmgesteuerte Einrichtung von lokalen Ausführungsumgebungen.

Dadurch erhöht sich ebenfalls die Infrastrukturunabhängigkeit, da durch die Skripte andere Infrastruktur in überschaubarer Zeit für die Ausführung vorbereitet werden kann. Im Idealfall sind dafür nur wenig manuelle Anpassung an den Skripten notwendig. In realen Situationen muss dies jedoch nicht immer gegeben sein, sodass gerade die Infrastrukturunabhängigkeit immer noch als zu schwach bezeichnet werden muss.

Ein weiterer Nachteil erwächst durch eine andere Betrachtung des Wortes »Infrastrukturunabhängigkeit«, da analog zum manuellen Deployment die Infrastruktur ihrerseits zunächst fest an die zu deployende Applikation gebunden wird. Eine Nutzung der Infrastruktur für weitere Zwecke kann unter Umständen zu einer Konkurrenzsituation in Bereichen, bei denen dies vermeidbar wäre, zwischen den beiden Anwendungen führen. Dass nebenläufig ausgeführte Prozesse auf derselben Berechnungsinstanz um Prozessorzeit der CPU konkurrieren, ist unvermeidlich, jedoch Details wie beispielsweise Portkonflikte oder das Benötigen unterschiedlicher Versionen einer Bibliothek können zumindest prinzipiell vermieden werden.

Dieses letzte Argument weist ebenfalls darauf hin, dass der Grad der Isolation zwischen mehreren Prozessen bei dieser Deploymentform als unzureichend zu bewerten ist. Lediglich der Aufwand des Zuschaltens weiterer Instanzen auf vorkonfigurierten Maschinen ändert sich im Vergleich zum manuellen Deployment nicht, da lediglich ein Prozess auf dem jeweils lokalen System gestartet werden muss.

Diese Ergebnisse sind ebenfalls in Tabelle 10.7 festgehalten.

10.6.4. Virtualisierung

Virtualisierung ist eine lange bestehende Technologie. Ihr Kern basiert auf der Bereitstellung eines vollwertigen Betriebssystems mit virtueller (softwarebasierter) Hardware. Wie zu Beginn der Arbeit in Abschnitt 2.6.4 eingeführt, erreichte die Virtualisierung durch ihre Vorteile der Portabilität und der Möglichkeit Snapshots zu erzeugen, zunehmende Beliebtheit in professionellen Szenarien. Insbesondere beim Deployment auf Servern können so deutliche Vorteile erzielt werden. Darüber hinaus bieten virtuelle Maschinen (VMs) exzellente Isolationsmöglichkeiten der darin laufenden Prozesse, wodurch wiederum Mandantenfähigkeit von Anbietern dynamisch mietbarer Rechenkapazität begünstigt wird. Ein entscheidender Vorteil in modernen Cloud-Architekturen.

Diese Eigenschaften sind durchaus auch für die Simulation von Referenznetzen von Interesse, da so eine erhöhte Portabilität der Simulationsinstanzen erreicht werden kann. Ein Berechnungsknoten muss lediglich das Hostsystem für virtuelle Maschinen mittels spezialisierter Software stellen und ist weitestgehend unabhängig von der darin ausgeführten Anwendung. Eine Ausnahme bilden an dieser Stelle Anwendungen, welche auf spezialisierte Hardware wie beispielsweise Sensoren zugreifen müssen. Diese spezialisierte Hardware muss dann virtualisiert in der virtuellen Maschine bereitgestellt werden. Andernfalls ist die Anwendung schlicht nicht in einer rein virtualisierten Umgebung einsetzbar. Je nach Umfang und Aufwand ist eine Trennung zwischen hardwarespezifischem Code und Weiterverarbeitung der gewonnenen Daten denkbar. Ein nicht virtualisierter Teil könnte mittels einer Kommunikationstechnik in den verbleibenden Teil der Simulation eingebunden werden.

In Abbildung 10.1 findet sich eine ursprüngliche konzeptuelle Zeichnung für den Einsatz virtueller Maschinen im Kontext von verteilten Referenznetzsimulationen. Sie entstammt der Veröffentlichung des Autors (Moldt, Röwekamp und M. Simon, 2017). Wie eingehend erläutert muss die Deploymentform nicht einheitlich für alle Komponenten gewählt werden. Daher dient die Darstellung hier nur der Vorstellung von Möglichkeiten und Nachteilen der Form des Deployment als virtuelle Maschine bezogen auf die Einzelbestandteile des Systems und nicht

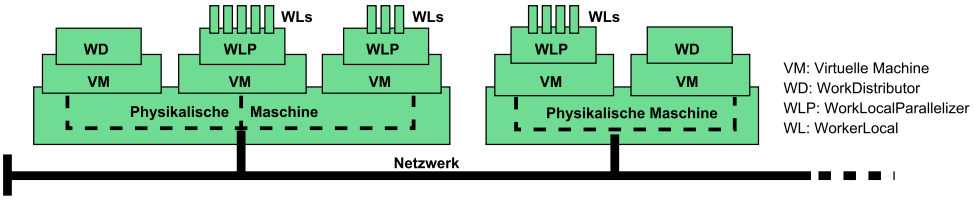

Abbildung 10.1.: Altes Modell zum VM Einsatz

als Gesamtvorschlag für das System. Darauf aufbauend wird die Diskussion am Ende dieses Abschnitts auch bezogen auf die einzelnen Komponenten geführt. Es handelt sich bei der Darstellung um den ersten Prototypen für den Einsatz virtueller Maschinen bei der Simulation von Referenznetzen. Dieses Modell wird auch noch einmal im Kapitel 14 zu Prototypen beim Beispiel zu Primfaktorzerlegung in Abschnitt 14.1.1 zum Einsatz kommen.

Die Kernidee dabei besteht darin, dass eine Arbeitslast in einer virtuellen Maschine lokal in eine nebenläufige Form überführt wird und sämtliche Steuerung durch die globale Referenznetzsimulation erfolgt. Eine schematische Darstellung dieser ersten Idee zeigt Abbildung 10.1. Dieser Schritt wird in der Grafik durch den »WorkLocalParallelizer« umgesetzt. Nach der Parallelisierung wurden in dieser früheren Fassung mehrfache »WorkerLocal« eingesetzt, um die jeweiligen Aufgaben abzuarbeiten. Es existierte pro physikalischer Maschine ein »WorkDistributor«, welcher die jeweiligen WorkLocalParallelizer koordinierte und mit Arbeitspaketen versorgte. Eine Kommunikation der verschiedenen WorkDistributor gewährleistete so eine knotenübergreifende Kooperation.

Die vergangenen Abschnitte dieses Kapitels haben jedoch den Einsatz weiterer Systemkomponenten wie dem Management-System eingeführt, erörtert und begründet. Folglich liegt es nahe, das in Abbildung 10.1 vorgestellte Modell in den neuen Kontext zu überführen. Eine entsprechende schematische Darstellung, welche den Einsatz von Clustermanager und Management-System berücksichtigt, ist in Abbildung 10.2 zu finden. Zunächst lässt sich festhalten, dass die erweiterte Trennung zwischen dem lokalen Einführen von Nebenläufigkeit und Arbeitslastverteilung in weiteren Untersuchungen[5] und Überlegungen im Allgemeinen nicht die gewünschten Vorteile ergeben hat. Insbesondere der Einsatz mehrerer virtueller Maschinen pro physikalischem Knoten ergab nur selten Vorteile durch die intrinsische Nebenläufigkeit des Renew Simulators. Daher wurde in der aktualisierten Fassung auf den Einsatz eines dreistufigen Konzepts in der Verteilungssicht pro physikalischer Maschine verzichtet, sodass nicht mehr eine *WorkDistributor* Netzinstanz pro physikalischer Maschine existiert sondern global eine. In sehr

[5]Ein weiterer Prototyp ohne diese Dreiteilung wird in Abschnitt 14.1.2 beschrieben.

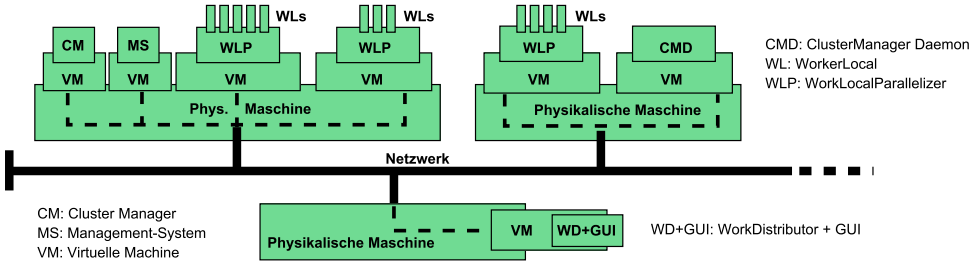

Abbildung 10.2.: Schematische Darstellung vom Einsatz von virtuellen Maschinen

großen Simulationen wäre es denkbar diese Stufe wieder einzuführen, sobald die *WorkDistributor* Netzinstanz an Belastungsgrenzen stößt. In diesem Fall müssten als Unterschied jedoch Mengen von physikalischen Knoten unter je einer der WorkDistributor-Instanzen zusammengefasst werden.

Ein weiterer Punkt, welcher in der Originalfassung nicht adressiert wurde, ist die Interaktion eines Nutzers mit dem System durch eine grafische Oberfläche. Es wurde angenommen, dass jede virtuelle Maschine eine grafische Oberfläche bereitstellt. Wie bereits erläutert liegt es nahe, im zu konstruierten System eine primäre Instanz zu benennen und diese mit einer grafischen Oberfläche auszustatten, während die anderen Instanzen nicht zwangsläufig eine grafische Oberfläche besitzen müssen. Der Verzicht auf eine grafische Oberfläche hat speziell im Simulator RENEW noch weitere Vorteile, da bestimmte Klassen so nicht geladen werden müssen und keine Last durch die Anzeige der Simulation generiert wird. Für den Verzicht auf grafische Oberflächen spricht außerdem, dass gerade in einem Szenario, bei dem die dynamische Skalierung der Simulation definiertes Ziel ist, die dynamisch akquirierten Instanzen häufig ohne Überwachung bzw. direkte Betrachtung agieren. Daher ist davon auszugehen, dass die lokale Bereitstellung einer grafischen Oberfläche auf diesen Plattformen sogar gänzlich ohne Vorteile bleiben würde, da sie in vielen Fällen nie durch menschliche Beobachter betrachtet wird.

Im Kontext des Einsatzes von virtuellen Maschinen ergibt sich als erster Vorteil die Portierbarkeit der Instanzen der virtuellen Maschinen. Somit könnten beim Ausfall einer physikalischen Maschine die darauf ausgeführten Simulationsanteile gebündelt an anderer Stelle hochgefahren werden. Hierzu wäre es notwendig, einen entsprechenden Snapshot der virtuellen Maschine auf dem ausgefallen physikalischen Knoten zu besitzen. Ausfälle kündigen sich im Normalfall jedoch nicht planbar an, sodass nicht zwangsläufig ein aktueller Snapshot mit dem neuesten Zustand bestehen muss. Die Erzeugung von Snapshots ist insbesondere bei True-Concurrency Simulationen nicht trivial, hat Implikationen zum Persistieren des Simulationszustandes und wird detailliert noch einmal in Abschnitt 11.6 zum Simulationsfeed aufgegriffen. Aus diesem Grund muss angenommen werden, dass

auf entfernten Simulatoren Datenverluste auftreten können. Dieser Umstand sollte in kritischen Simulationsanteilen vonseiten des Modellierers bedacht werden, indem Redundanzen vorgesehen werden und verloren gegangene Arbeitsanfragen erneut angefragt werden können.

Nach den generellen Ausführungen bezüglich dem Einsatz von virtuellen Maschinen folgt nun die Bewertung dieser Deploymentform für die einzelnen Systembestandteile in Bezug auf Vorabaufwand und dynamische Möglichkeiten durch die Deploymentform. Trotz der durch die Virtualisierung ermöglichten Portabilität ist der Transfer einer virtuellen Maschine zum Akquirieren eines neuen Rechnerknotens aufwändig.

Die Begründung hierbei liegt darin, dass eine virtuelle Maschine sämtliche Systemkomponenten des simulierten Systems in einer realen Kopie vorhält. Dadurch müssen nicht nur anwendungsspezifische Bestandteile, sondern alle Bestandteile verschoben werden. Darüber hinaus muss eine virtuelle Maschine vollständig hochgefahren werden, bevor die darin gekapselte Anwendung gestartet werden kann. Je nach Komplexität des verwendeten Basissystems und der Performance des Hostsystems können hierbei einige Sekunden bis Minuten vergehen. Werden selten neue Instanzen des Systems hochgefahren, handelt sich dabei meist um vertretbare Zeitdifferenzen. Da jedoch gerade in skalierbaren Systemen insbesondere die Geschwindigkeit eine Rolle spielt, in der auf Änderungen der Umgebung reagiert werden kann, kann eine Differenz von einigen Sekunden bereits als kritisch bewertet werden.

Die Bewertung der Einzelaspekte orientiert sich daher maßgeblich daran, wie sehr der jeweilige Aspekt zu den soeben beschriebenen Anforderungen passt. Die Vorbereitung umfasst im Normalfall stets die Auswahl einer geeigneten Betriebssystemumgebung, dem Installieren aller notwendigen Bibliotheken und dem Speichern der Anwendungsdaten selbst innerhalb des virtuellen Betriebssystems.

Sowohl Kommunikationsmedium als auch GUI Komponenten müssen (zunächst) lediglich in einfacher Ausführung vorgehalten werden. Der Vorabaufwand umfasst die zuvor erläuterten Bestandteile und hat als Anforderung die Bereitstellung einer Java JVM und eines Renew Simulators mit entsprechenden Bibliotheken. Mit der Möglichkeit der überhaupt existierenden Portierbarkeit ergeben sich damit Vorteile im Gegensatz zu der einfachen Ausführung, wie sie bei manuellem Deployment oder automatischer Konfiguration auftritt. Die Vorabkonfiguration bezüglich Ports richtet sich nach der Einbindung der virtuellen Maschinen ins Hostsystem. Bei der Verwendung virtueller Adressen bestehen keine Konflikte, wohingegen eine Zusammenschaltung unter der Adresse des Hostsystems und der Verwendung VM-spezifischer Ports aufgrund von möglichen Konflikten problematischer – und daher nicht zu empfehlen – ist.

Kriterium	*Vorabaufwand*	*Möglichkeiten*
Kommunikationsmedium	-/o	o
Management-System	o	+
Clustermanager	- -	-
GUI Komponente	-/o	+
Zusätzliche Sim.instanzen	-/o	+

Tabelle 10.5.: Bewertung von VM basiertem Deployment

Das Management-System gestaltet sich ähnlich, mit dem Unterschied, dass hier der Aufwand zur Konfiguration von Ports geringer ausfällt. Das Zuschalten von Simulationsinstanzen benötigt im Vorabaufwand ähnliche Überlegungen wie das Kommunikationsmedium und die GUI Komponente. Durch die Portierbarkeit und die Gleichartigkeit der weiteren Simulationsinstanzen sind die Möglichkeiten in dieser Deploymentform jedoch den zuvor betrachteten Varianten deutlich überlegen. Wie zuvor ausgeführt ist die Portierbarkeit zwar gegeben, jedoch teilweise nur mit erhöhtem Zeitaufwand und stärkerer Verzögerung zu realisieren.

Lediglich das Deployment des Clustermanagers würde sich als grundlegend schwierig gestalten. Ein Hauptaspekt dabei liegt darin, dass der Clustermanager Zugriff auf alle physikalischen Infrastrukturkomponenten benötigt. Dadurch müsste trotz eines Deployments in einer virtuellen Maschine ein großer Teil des Hostsystems und weiterer physikalischer Systeme an den Clustermanager angeschlossen werden, indem Zugangswege durch die Virtualisierungsschicht geöffnet werden. Daraus ergibt sich insgesamt ein immenser Vorabaufwand, welcher sich aus Konfiguration und gegebenenfalls weiterer Programmierung zusammensetzt. Die Möglichkeiten, sofern eine derartige Realisierung überhaupt möglich ist, wären nicht besser als die unter anderen Deploymentformen. Da der Clustermanager häufig selbst bereits ein verteiltes System ist, kann beispielsweise nur unzureichend auf die Funktion zurückgegriffen werden, Snapshots von virtuellen Maschinen herzustellen und zu laden. Es ist jedoch dabei denkbar, dass kommerzielle Produkte zur Virtualisierung Möglichkeiten bereitstellen, um mehrere Maschinen übergreifende Snapshots zu sichern und zu laden. Insgesamt ist es schwierig, ein Deployment des Clustermanagers durch virtuelle Maschinen zu rechtfertigen oder zu empfehlen.

Die gesammelten Bewertungen können auch in diesem Abschnitt in der zugehörigen Tabelle 10.5 gesammelt nachgeschlagen werden.

Abschließend sind noch die Bewertungen der globalen Aspekte zu betrachten. Zunächst ist zu bemerken, dass der Zeitaufwand, welcher benötigt wird, um eine neue Instanz auf einem vorbereiteten Knoten hinzuzuschalten, deutlich höher ausfällt als in den übrigen Deploymentformen. Die Begründung liegt darin, dass nun neben der Anwendung zusätzlich auch die gesamte sie umschließende Betriebssystemumgebung gestartet und hochgefahren werden muss.

Durch die generell mögliche Portierbarkeit der Anwendung innerhalb ihrer virtuellen Maschine entsteht eine Loslösung von der darunterliegenden Infrastruktur. Eine Ausnahme hierbei wäre lediglich das VM-basierte Deployment des Clustermanagers. Durch die Größe der Deploymenteinheiten ist jedoch trotz der generellen Möglichkeit ein Wechsel der Infrastruktur mit deutlichem Aufwand verbunden. Daher kann die Infrastrukturbindung nicht als minimal gesehen werden.

In einer virtuellen Maschine arbeitet die Anwendung in einem vollwertigen virtuellen Betriebssystem komplett isoliert von allen sonstigen Prozessen des Hostsystems. Dadurch ist der Grad der Isolation innerhalb einer virtuellen Maschine als exzellent zu bewerten. Zusätzlich können aufgrund der Isolation viele virtuelle Maschinen auf einem Hostsystem bereitliegen und nebenläufig oder sequenziell betrieben werden. So kann die Anwendungsbindung der Infrastruktur gering gehalten werden.

In einem eigens dafür vorgesehenen Betriebssystem können alle für die Ausführung der Anwendung notwendigen Bestandteile vorab in automatische Startsequenzen codiert werden. Die meisten Virtualisierungslösungen lassen ebenfalls eine automatisierte Steuerung bezüglich des Hoch- und Herunterfahrens von Instanzen zu. Der manuelle Aufwand, eine solche Lösung im Normalbetrieb zu unterhalten, nachdem der initiale Aufwand die Umgebung zu präparieren abgeschlossen ist, kann somit als äußerst gering bewertet werden.

Zusammenfassend können diese Ergebnisse ebenfalls als Spalte in Tabelle 10.7 nachgeschlagen werden.

10.6.5. Containerisierung

Das letzte betrachtete Deploymentformat ist die Containerisierung. Während Container diverse Ähnlichkeiten zu virtuellen Maschinen aufweisen, sind die beiden Konzepte jedoch grundlegend verschieden. Container wurden im Allgemeinen bereits im Grundlagenkapitel in Abschnitt 2.6.4 vorgestellt. Der zentrale Unterschied zu virtuellen Maschinen liegt darin, dass Container lediglich eine Isolation von Prozessen im Hostsystem darstellen, anstatt einem vollwertig virtualisierten Betriebssystem. Dadurch sind sie leichtgewichtig sowie einfach und schnell zu starten, bieten jedoch nicht so voll umfassende Isolationsmöglichkeiten wie virtuelle Maschinen, da zentrale Systemressourcen geteilt werden. Zusätzlich erfordert die Isolation eine sehr robuste und abgehärtete Implementation seitens der Containerisierungssoftware.

Gerade durch die Leichtgewichtigkeit und durch die schnelle Startmöglichkeit von Containern sind sie ein interessanter Ansatz für das Deployment von Referenznetzsimulationen. Da Container viele Eigenschaften mit virtuellen Maschinen teilen, treffen viele der Ausführungen aus dem vorangegangenen Abschnitt auch auf

Container zu. Daher werden im Folgenden primär die jeweiligen Unterschiede diskutiert.

Die Bewertung der einzelnen Kriterien orientiert sich im Wesentlichen an den Bewertungen des Deployments mittels virtueller Maschine. Dabei kann jedoch angenommen werden, dass sich der jeweilige Vorabaufwand für die Containervariante konzeptuell ungefähr mit dem eine entsprechende virtuelle Maschine bereitzustellen deckt. Die Implementation ist abhängig von der konkret eingesetzten technischen Containerisierungslösung, ist meist aber zugänglich und einfach umgesetzt, wie beispielsweise durch eine Definitionsdatei, welche den Aufbau des Containers definiert. Dabei kommt häufig ein Schichtensystem zum Einsatz, sodass auf bereits erzeugten Basisinstanzen aufgesetzt werden kann und sich der Aufwand somit insgesamt etwas geringer gestaltet.

Dies kann beispielsweise so genutzt werden, dass eine Basisfassung vom Referenznetzsimulator eingesetzt wird, auf welche dann nur noch die jeweiligen Spezialisierungen (Bibliotheken und Plugins) hinzugefügt werden müssen. Hierbei gilt es jedoch zu beachten, dass nach wie vor eine Lösung für die Plattformdefinition gestaltet wird, welche nur Basiskomponenten bereitstellt. Es ist jedoch denkbar für verschiedene Simulationen mit Containerebenen verschiedene Basisversionen der Plattform vorzuhalten. Auch ein Update der Definition könnte durch das Hinzufügen einer weiteren Ebene gelöst werden, welche die Differenz der Versionen beschreibt. Praktisch ist eine derartige Umsetzung jedoch einigen Nachteilen unterworfen, welche detaillierter im Abschnitt 10.6.7 diskutiert werden.

Die Umsetzung für das Kommunikationsmedium unterscheidet sich in ihren Möglichkeiten nicht wesentlich von der virtualisierten Variante. Genauso kann dieser Umstand für das Management-System, wie es als zustandsloser Webservice geplant ist, beschrieben werden.

Analog zum Deployment mittels virtuellen Maschinen ist von der Bereitstellung des Clustermanagers auf Basis von Containern definitiv abzuraten. Die Gründe hierfür ergeben sich maßgeblich auch daraus, dass eine direkte Steuerung der Infrastruktur möglich sein muss für den Clustermanager. Es existieren für Containertechnologien dafür zwar Lösungen, diese erhöhen allerdings den Aufwand und die Komplexität deutlich, sodass im Rahmen dieser Arbeit davon abgesehen wird.

Ein echter Unterschied ist bei der GUI Komponente zu erwarten, da Container im Normalfall keine Anbindung an ein grafisches Nutzerinterface vorsehen. Es ist theoretisch möglich auch Container mit einer grafischen Oberfläche zu versehen. Dies ist jedoch häufig mit einem stark erhöhten Aufwand verbunden. Da mit dem Einsatz eines Containers für die grafische Oberfläche lediglich der einmalige Installationsaufwand auf dem primären Knoten der Simulation entfällt, sind die Vorteile durch den Einsatz eines Containers nicht bahnbrechend. Nachteilig

Kriterium	*Vorabaufwand*	*Möglichkeiten*
Kommunikationsmedium	o	o
Management-System	+	+
Clustermanager	- -	-
GUI Komponente	- -	-
Zusätzliche Sim.instanzen	o	++

Tabelle 10.6.: Bewertung von Deployment mittels Container

wirkt sich jedoch zusätzlich aus, dass der Nutzer bei jeder Änderung der GUI Installation eine Änderung indirekt im Container vornehmen muss.

Die Bereitstellung neuer Simulationsinstanzen kann stark von der Containertechnologie profitieren. Da wie bereits ausgeführt die Notwendigkeit für eine grafische Oberfläche bei den dynamisch erzeugten Instanzen häufig entfällt, entsteht durch deren Abwesenheit bei Containern auch kein Nachteil. Das dynamische Starten erfolgt in sehr geringer Zeit und durch die schichtbasierte Architektur müssen bei einer Änderung nur kleinste Datenmengen auf die Zielsysteme übertragen werden. Die Vorteile, die virtuelle Maschinen bei dieser Deploymentform aufweisen konnten, nämlich die Portabilität und das Entfallen einer individuellen Installation auf dem Zielsystem, erhalten auch Container vollumfänglich. Für weitere Simulationsinstanzen muss jedoch nach wie vor ein Container bzw. ein Image davon vorbereitet werden.

Alle Ergebnisse sind analog zu den anderen Deploymentformen in der Tabelle 10.6 festgehalten.

Bezogen auf die globalen Eigenschaften können sich Container durch ein schnelles Hinzuschalten neuer Instanzen gegen virtuelle Maschinen behaupten. Da jedoch der Overhead eines Containers immer noch größer ist als keinerlei Overhead, reicht die Zeiteffizienz nicht ganz an den nativen Start eines Simulators heran.

Durch die einfache Portierbarkeit und die Schichtenarchitektur in der Containertechnologie entsteht eine außerordentlich geringe Bindung an die Infrastruktur. Basisebenen können von schnellen externen Quellen bezogen werden oder existieren bereits auf dem Zielsystem, sodass lediglich kleine Differenzen zu den allgemeinen Basisebenen übertragen werden müssen und so Simulationsteile leicht auf neue Systeme verschoben werden können. Dies bezieht sich sogar auf Systeme, auf denen noch nie ein Teil der Simulation lief. Der Grad der Isolation ist gut, jedoch nicht zwangsläufig auf dem Level einer virtuellen Maschine, bei der sämtliche Systembestandteile einzig und allein für die Anwendung vorgehalten werden.

Durch gemeinsame Nutzung von Systemressourcen bestehen potentielle Überschneidungen, auch wenn Anbieter von Containertechnologie die Isolation ihrer

Instanzen verstärkt fokussieren. So kann es sich mit umfangreicher aus Testung und dem Reifen der Technologie ergeben, dass der Grad der Isolation nicht mehr von der in virtuellen Maschinen zu unterscheiden ist. Das Problem der schwierigeren Trennung auf konzeptueller Ebene bleibt jedoch bestehen und containerisierte Lösungen sind daher inhärent aufwendiger zu isolieren.

Der individuelle Aufwand, welcher zum Deployment eines neuen Containers notwendig ist, ist sehr gering. Im Idealfall muss lediglich der Name der Basisebene bekannt sein, sodass ein entsprechender neuer Container hinzugezogen werden kann. Dies kann problemlos automatisiert erfolgen.

All diese Ergebnisse sind neben den anderen Ergebnissen in Tabelle 10.7 festgehalten und werden sogleich im folgenden Abschnitt noch einmal diskutiert und gegeneinander abgewogen.

10.6.6. Eigenschaften und Entscheidungen

Nachdem in den vergangenen vier Abschnitten die jeweiligen Deploymentformen detailliert vorgestellt wurden, reflektiert dieser Abschnitt noch einmal die globalen Eigenschaften und betrachtet ideale Wahlen für die jeweiligen Komponenten.

Zunächst kann festgehalten werden, dass der Zeitaufwand, welcher benötigt wird, um eine weitere Instanz der Simulationen hinzuzuschalten ohne den Overhead einer Virtualisierung bzw. eines Containers logischerweise am geringsten ausfällt. Während jedoch bei einer virtuellen Maschine ein ganzes virtuelles Betriebssystem hochgefahren werden muss, beschränkt sich der Overhead beim Container auf eine Form der Prozessisolation auf dem Hostsystem und kann sehr schnell erfolgen.

Die Unabhängigkeit von konkreter Infrastruktur ist bei manuellem Deployment sehr niedrig, da jedes System komplett individuell vorbereitet werden muss. Durch Autokonfiguration kann diese Bindung im gewissen Maße reduziert werden, indem Umgebungen automatisch vorbereitet werden können. In einer virtuellen Maschine kann diese Umgebung gleich vollständig mitgeliefert werden. Dadurch wird bei der virtuellen Maschine der Umzugsaufwand auf eine neue Infrastruktur allerdings auch beträchtlich größer. Lediglich der Container besitzt im Allgemeinen eine sehr hohe Flexibilität bezüglich der Infrastruktur.

Der Grad der Isolation liegt in erster Linie an der Form der Laufzeitumgebung. So ist diese bei einer direkten Ausführung auf dem Hostsystem im Normalfall nicht gegeben. Durch eine komplette Virtualisierung entsteht hingegen eine (fast) ideale Isolation. Auch bei Containern kann durch die Kapselung der Prozessstruktur von einer starken Isolation ausgegangen werden. Container teilen sich jedoch die selben Systemressourcen und verwenden systemseitige Prozessisolation um sie gewährleisten zu können. Sie ist somit deutlich komplexer zu realisieren als in

Deploymentform	*Manuell*	*Autoconfig*	*VM*	*Container*
Zeitaufwand Hinzuschalten	++	++	-	+
Infrastr. unabhängigkeit	- -	-	+	++
Grad der Isolation	- -	- -	++	+/++
Individueller Aufwand	- -	+	++	++

Tabelle 10.7.: Bewertung der globalen Aspekte pro Deploymentform

virtuellen Maschinen und benötigt eine solide und ausgefeilte Implementation um wirksam zu sein.

Der individuell notwendige Aufwand fällt logischerweise beim manuellen Deployment am größten aus. Durch automatische Konfiguration kann individueller Aufwand stark reduziert werden. Am geringsten jedoch fällt der Aufwand bei virtuellen Maschinen und Containern aus, da hierbei auf automatisierte Art und Weise das gesamte Umgebungssystem beim Anwendungsdeployment mitgeliefert wird.

All diese Ergebnisse sind ebenfalls übersichtlich in Tabelle 10.7 festgehalten. Nach der Betrachtung der einzelnen Eignungen der Deploymentformen sowie der globalen Eigenschaften besteht nun die Möglichkeit, eine begründete Entscheidung für die Deploymentform pro Komponente zu treffen.

Bei einem einfachen Blick auf die Tabelle erscheint die Containerisierung als vielversprechendste Deploymentform. Dabei sollte jedoch beachtet werden, dass die Tabelle nur die *globalen* Eigenschaften der Deploymentformen beschreibt, welche für den Gesamtkontext skalierender Referenznetzsimulationen im Rahmen der Arbeit als relevant betrachtet werden. Die individuellen Überlegungen der vergangenen Abschnitte sind in der Tabelle hingegen nicht abgebildet. Daher erfolgt die Betrachtung und Entscheidung pro Komponente.

Das Vorhalten des Kommunikationsmediums ist entsprechend der MUSHU-Architektur notwendig. Andere Lösungen können hier abweichen. Da jedoch die Implementation des Distribute Plugins auf Java RMI aufbaut, lohnt sich an dieser Stelle eine Betrachtung einer RMI bzw. Distribute Registry.

Im Falle von Java RMI ist eine einzige Registry vorgesehen. Eine Skalierung der Registry ist daher weitgehend unerheblich und die Deploymentformen unterscheiden sich nicht großartig in ihren Möglichkeiten. Die Vorabaufwände zur Bereitstellung unterscheiden sich ebenfalls nicht sehr stark. Zusätzlich sollte die RMI Registry auf einfache Art und Weise manuell neustartbar sein. Gerade bei der Entwicklung von verteilten Referenznetzsimulationen kann es zu Abstürzen kommen und somit die RMI Registry in einen unsauberen Zustand geraten. Dies kann jedoch in jedem Deploymentformat bewerkstelligt werden, beispielsweise durch den Einsatz von Skripten. Lediglich der Container hat hier den einfachen Vorteil, dass er einfach und schnell neu gestartet werden kann und der Aufwand

den Start der RMI Registry zu kodieren bereits in der Vorbereitung für das Containerimage erfolgt sein muss. Daher sollte ein Deployment der RMI Registry via Container angestrebt werden.

Beim Management-System stehen von den Individualbetrachtungen Containerisierung und manuelles Deployment in etwa auf gleicher Stufe. Auf der Basis der globalen Kriterien ist jedoch die Bereitstellung als Container vorzuziehen.

Entgegen der bisherigen sonstigen Übereinstimmung der individuellen mit den globalen Betrachtungen gestaltet sich die Entscheidung bezüglich der Form des Deployments des Clustermanagers als weniger eindeutig. In den vergangenen Abschnitten wurde erörtert, warum ein Deployment mittels Container oder virtueller Maschine nur sehr schwer zum Erfolg führen kann. Auch der Einsatz vollautomatischer Konfiguration wurde als nicht empfehlenswert identifiziert, da die Bereitstellung der automatischen Konfiguration vermutlich den Aufwand der manuellen Installation übersteigt. Folglich ergibt sich als naheliegendste Deploymentform in der Tat die manuelle. Der damit verbundene Aufwand ist nicht unerheblich. In kommerziellen Angeboten von Cloudprovidern kann jedoch glücklicherweise häufig auf einen bereits vorinstallierten Clustermanager zurückgegriffen werden.

Durch die Abwesenheit einer umfassenden GUI-Unterstützung in Containerlösungen kann der Einsatz von Containertechnologie für die GUI-Komponente nicht empfohlen werden. Da die Möglichkeiten mit den verbleibenden Deploymentformen nicht stark unterschiedlich zu bewerten sind, kann dieses Deployment letztendlich an die Wünsche des Nutzers angepasst werden. Während der native manuelle Weg für viele Nutzer einen natürlichen Umgang mit Software darstellt, sind viele gerade professionelle Nutzer auch an den Einsatz von virtuellen Maschinen gewöhnt. Der Einsatz einer virtuellen Maschine könnte in Fällen gewünscht sein, bei denen möglichst wenig Änderungen am Hauptsystem der Nutzer vollzogen werden sollen. Der Ansatz von automatischer Konfiguration ist möglich, jedoch vielen Nutzern nicht vertraut. Daher kann an dieser Stelle die Empfehlung der Nutzung einer nativen Installation oder aber einer virtuellen Maschine ausgesprochen werden.

Letztendlich folgt die Betrachtung von zusätzlichen dynamisch erzeugten Simulationsinstanzen. Bei diesen überwiegt sowohl global als auch individuell deutlich der Einsatz von Containern. Zusätzliche Simulationsinstanzen stellen den Hauptaspekt der skalierbaren Simulation von Referenznetzen dar und genießen daher ein besonderes Augenmerk. Durch die hohe Infrastrukturunabhängigkeit und schnelle Startbarkeit kann die Simulation effizient erweitert werden. Eine gezielte Steuerung und die organisierte Bereitstellung von Images über Webschnittstellen runden das Gesamtbild ab. Somit ist für die Erzeugung zusätzlicher Simulationsinstanzen die Containertechnologie eindeutig zu empfehlen.

Systemkomponente	*Deploymentform*
RMI Registry	Container
(Anderes Komm.medium)	Technologieabhängig
Management-System	Container
Clustermanager	Nativ/Manuell
GUI Komponente	Nativ/Manuell oder VM
Zusätzliche Sim.instanzen	Container

Tabelle 10.8.: Entscheidung der Deploymentform je Aspekt

Die Einzelentscheidungen sind abschließend noch einmal in Tabelle 10.8 festgehalten.

10.6.7. Einsatz von Continuous Integration

Die Betrachtungen der letzten Abschnitte haben ergeben, dass ein Deployment durch Containerisierung für einige der Bestandteile der Ausführungsumgebung sinnvoll sind. Während Container an sich auch manuell konstruierbar sind, synergieren sie hervorragend mit automatisierter Erzeugung. An dieser Stelle ist es sinnvoll, den Ansatz von Continuous Integration (CI) und insbesondere Continuous Delivery (CD) zu betrachten. Das Konzept wurde bereits im Grundlagenkapitel in Abschnitt 2.6.3 eingeführt und erläutert.

Das Ziel dieses Abschnitts besteht darin, für die jeweiligen Komponenten zu entscheiden, inwiefern der Einsatz von CI/CD einen substantiellen Mehrwert bietet und angestrebt werden sollte. CI/CD ermöglicht (u.a.) die deutlich beschleunigte - da automatisierte - Bereitstellung von containerisierten Anwendungen. Insbesondere bei der Anwendung mit komplexen Konfigurationen kann dadurch eine Menge Zeit eingespart werden. Dies ist vor allem dann wichtig, wenn die Anwendung nach einem agilen Grundprinzip weiterentwickelt werden soll.

Wird dieser Grundgedanke weiter verfolgt, so ergibt sich direkt ein Vorteil für das Management-System. Die RMI Registry und zusätzliche Simulationsinstanzen verwenden einen mehr oder weniger fixierten (headless) Renew Simulator als Grundlage. Renew selbst ist selbstverständlich auch der Weiterentwicklung unterworfen und durch aktuelle Neuerungen bewegen sich die Entwicklungsmodalitäten bei Renew hin zu wesentlich kürzeren Releasezyklen. Die CI/CD Aspekte werden auch durch die Agilität der Plattform im Sinne der Cloud-Nativity adressiert, wie sie in Abschnitt 6.1.4 eingeführt wurde.

Ein weiterer Punkt beim Einsatz von CI/CD besteht darin, dass durch die implementierte Pipeline eine Form der Dokumentation des Kompilier- und Bereitstellungsvorgangs erfolgt. Daher ist es gerade im universitären und wissenschaftlichen

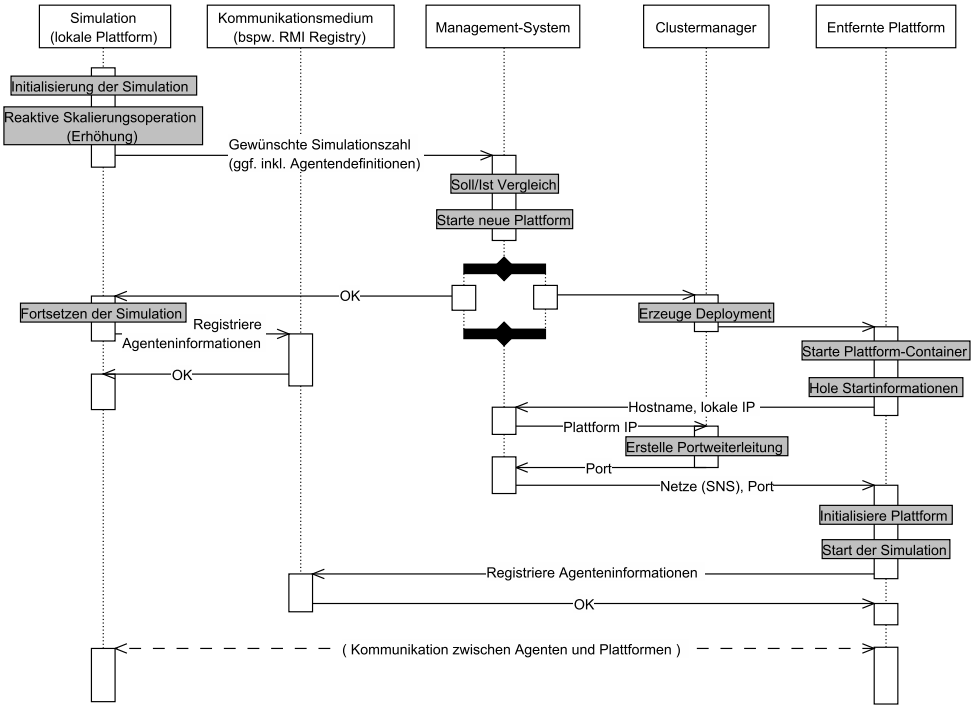

Abbildung 10.3.: Schematische Darstellung vom Ablauf eines Simulationsstarts

Kontext hilfreich, wenn der Deploymentprozess auch für andere Betrachter und für spätere Untersuchungen eindeutig nachvollziehbar ist.

Aus diesem Grund kann sich für alle Bestandteile des Systems für den Einsatz von CI ausgesprochen werden. Im Falle der containerbasierten Bereitstellung bietet sich darüber hinaus der Einsatz von CI/CD an.

10.6.8. Ablauf des Simulationsstarts in der Umgebung

Zum Abschluss der Betrachtung der Deploymentformen soll das Ablaufdiagramm aus Abbildung 7.6 in Abschnitt 7.3.3 konkretisiert werden. Abbildung 10.3 zeigt eine Variante, welche die Überlegungen dieses Kapitels mit einbezieht.

Die Abbildung beschreibt den Ablauf eines Simulationsstarts, bei dem direkt nach dem Start ein weiterer entfernter Simulator angefordert wird und sich die lokale Plattform erst nach der Anforderung selbst beim Kommunikationsmedium registriert. Dieser Ablauf erfordert eine entsprechend robuste Implementation der jeweiligen Plattformen, damit mit einer etwaigen, noch nicht registrierten Plattform umgegangen werden kann.

Nach der Initialisierung der Simulation wird besagte Skalierungsoperation an das Management-System gerichtet. Aus Sicht des MUSHU-Plattformmanagements handelt es sich dabei um eine reaktive Skalierungsoperation, welche von einer Plattform auf dem Plattformmanagement ausgeht. Das Management-System vergleicht entsprechend des Reconciler-Patterns die aktuell verfügbare Menge an Plattformen mit der gewünschten Menge an Plattformen und kommt in diesem Beispiel zu dem Schluss, dass eine weitere Plattform gestartet werden muss. Der lokalen Simulation wird zurückgemeldet, dass die Anfrage eingegangen ist. Gleichzeitig wird der angeschlossene Clustermanager beauftragt, ein neues Deployment einer Plattform auszurollen. Die entfernte Plattform wird in Form eines Simulator-Containers gestartet und bezieht mit ihren lokalen Informationen die Informationen zur aktuellen Simulation vom Management-System.

An dieser Stelle kann es notwendig sein, speziell für die neue Plattform Firewall- und/oder Portregeln zu etablieren. Diese Anfrage wird an den Clustermanager gerichtet, welcher einen entsprechenden verwendbaren Port zurückliefert. Etwaige übertragene Agentendefinitionen sowie der Port werden an die entfernte neue Plattform übertragen. Diese startet auf der Basis dieser Informationen den lokalen Simulator, welcher fortan als Plattform fungiert und sich sogleich beim Kommunikationsmedium registriert. An dieser Stelle ist eine Anfrage der lokalen Plattform bezüglich der entfernten Plattform erfolgreich und die Kommunikation zwischen den beiden Plattformen kann erfolgen. Je nach Kommunikationsmodalität wird hierbei auch das Kommunikationsmedium einbezogen. Details zu dieser Kommunikationsmodalität werden detaillierter in Kapitel 12 beschrieben.

10.7. Darstellung des Gesamtsystems zur Skalierungskontrolle

Dieser Abschnitt dient der rekapitulierenden Darstellung aller in diesem Kapitel gewonnenen Erkenntnisse und aller getroffenen Entscheidungen bezüglich der Konzeption einer Ausführungsumgebung für skalierbare Referenznetzsimulationen. Dabei wird zunächst kurz beschrieben, welche Bestandteile das System besitzt, welche dedizierten Aufgaben diese wahrnehmen und wie sie miteinander interagieren.

10.7.1. Kurzfassung der wesentlichen Systembestandteile

Zum vereinfachten Verständnis der Bestandteile ist es hilfreich, auf eine bildliche Darstellung der Systembestandteile zurückgreifen zu können. In Abbildung 10.4 ist die (technische) Architektur der Komponenten um das Management-System

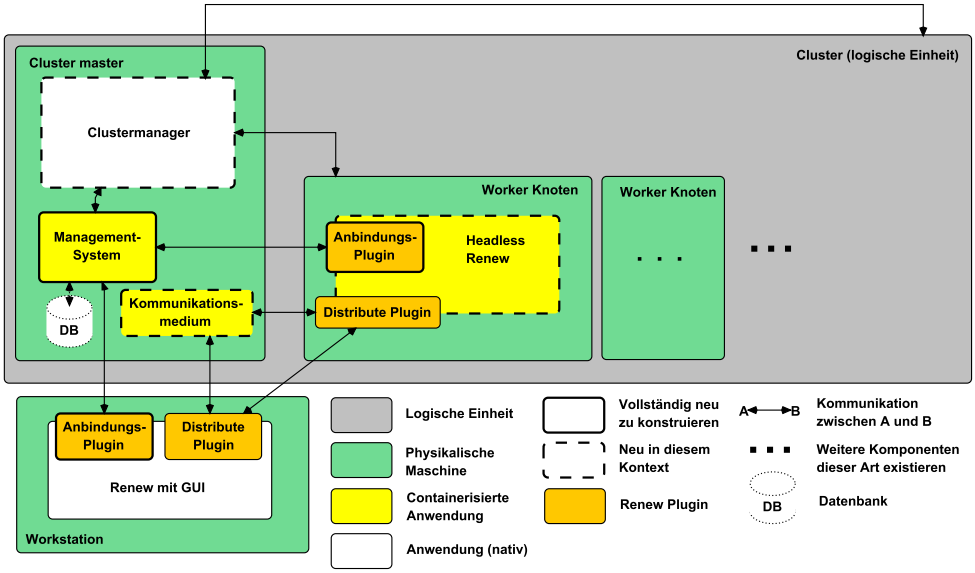

Abbildung 10.4.: Schematische Darstellung der Simulationssteuerung

herum schematisch dargestellt. Dabei sind die Komponenten, deren Umsetzung die (neuen) Beiträge des Autors darstellen, durch fettgedruckte Umrandungen kenntlich gemacht. Das Gesamtkonzept ist ebenfalls einer der Beiträge des Autors im Rahmen der Arbeit. Dieses Diagramm wird später in Abschnitt 12.8 auf die gesamte schematische Darstellung zur technischen Realisierung des Plattformmanagements in Abbildung 12.15 erweitert.

Die geplanten Komponenten umfassen im Wesentlichen zwei Anteile: Ein Management-System, welches als zustandsloser (Web-)Service implementiert werden soll und eine Simulationsanbindung in Form eines Plugins für den Simulator Renew, welches mit dem Management-System über ein zustandsloses standardisiertes Protokoll kommuniziert. Dabei soll ein menschenlesbares, standardisiertes, frei verfügbares Datenformat zum Informationsaustausch eingesetzt werden. Die Anbindung an die laufende Referenznetzsimulation erfolgt durch ein entsprechendes Anbindungsplugin, welches orange in der Grafik dargestellt ist.

Neben den genannten neu zu konstruieren Komponenten, welche in der Abbildung durch fettgedruckte durchgehende Linien dargestellt sind, benötigt die gesamte Ausführungsumgebung noch weitere Bestandteile. Als Basisbestandteil wird das Distribute Plugin für die Simulationserweiterung und Kommunikation innerhalb der Simulation eingesetzt. Des Weiteren ergab es sich als notwendig, Renew innerhalb von Containertechnolgie zu betreiben. Erstmalig wurde dies in den Untersuchungen des Autors umgesetzt, welche dieser Arbeit vorausgingen. In der

Darstellung bezieht sich dies auf die Komponenten Headless Renew sowie das Kommunikationsmedium, sofern dieses als RMI Registry umgesetzt wird.

Zusätzlich zu den Renew-spezifischen Komponenten wurde die Notwendigkeit eines Clustermanagers dargelegt. Derartige Software wurde bisher noch nicht im Kontext von Referenznetzsimulation eingesetzt und die Eignung musste daher untersucht werden. Alle durch den Autor der Arbeit erstmalig *in diesem Kontext* eingesetzten Komponenten sind mit Fett gezeichneten unterbrochenen Linien dargestellt. Der Clustermanager steuert alle physikalischen Knoten (grüne Kästen), welche zu einem logischen Cluster (grauer Kasten) zusammengeschlossen sind.

10.7.2. Aufgabenverteilung

Das Anbindungsplugin stellt die Schnittstelle für alle reaktiven Skalierungsanfragen von innerhalb einer bestimmten Plattform bereit. Es bietet in Form von Funktionsaufrufen die Möglichkeit, neue Plattformen auf Berechnungsknoten für die Simulation zu akquirieren oder diese freizugeben. Beim Start einer Simulation ist es dafür verantwortlich, dass die zu simulierenden Netze bzw. initial vorhandenen Agentendefinitionen gepackt und ans Management-System übergeben werden.

Das Management-System stellt Informationen für neue Plattformen bereit, welche sich in eine laufende Simulation integrieren möchten. Diese Informationen umfassen eine Adressierung des Kommunikationsmediums, eins oder einige (statische) Netzdefinitionen für die Simulation, sowie das Netz mit dessen Instanz die entfernte Simulation gestartet werden soll. Darüber hinaus abstrahiert das Management-System das Interface für Skalierungsanfragen an den Clustermanager und hilft bei der Informationsverteilung bezüglich verwendeter Ports durch die einzelnen Plattformen. Es ist ebenfalls für ein kontrolliertes Abbauen aller entfernten Plattformen verantwortlich, sobald die Simulation auf dem Primärknoten kontrolliert terminiert wird. Anfragen zur Skalierung können durch das Management-System grundsätzlich von jeder Plattform angenommen werden, welche an der Simulation teilnimmt.

Das Management-System ist als zustandslose Anwendung konzipiert. Daher ist das Vorhalten von Daten, welche die Ausführung betreffen, minimal zu halten. Da das Management-System jedoch aber auch als Verteilungspunkt für zu simulierende Netze und als zentraler Zugangspunkt zum Clustermanager dient, ist die Sicherung einiger Daten unvermeidlich. Zu diesem Zwecke kann eine Datenbank eingesetzt werden. In Testszenarien kann alternativ auch auf die (externe) persistente Speicherung der Daten verzichtet werden. In diesem Fall verliert das Management-System jedoch formal seine Zustandslosigkeit. Das Management-System wird als Container bereitgestellt.

Der Clustermanager ist eine externe Software, welche die Steuerung der physikalischen Knoten innehat. Der primäre Auftrag ist dabei das Starten von neuen containerisierten Plattformen auf anderen physikalischen Maschinen sowie das Bereitstellen einer geeigneten softwarebasierten Netzwerklösung (Routing, Firewalls, etc.), sodass der Cluster als logische Einheit existieren kann. Um die Möglichkeit der Steuerung der physikalischen Hardware so einfach wie möglich zu gestalten, wird der Clustermanager traditionell manuell installiert.

Das Kommunikationsmedium wird als Container bereitgestellt, um einfache Neustartbarkeit herzustellen. Das Kommunikationsmedium ist naturgemäß zustandsbehaftet, daher sollte eine persistente Speicherung angestrebt werden. Für das Funktionieren bisheriger Komponenten im Kontext von Renew ist es erforderlich, auf die für Java RMI notwendige RMI Registry zurückzugreifen. Sie ist zentral und zustandsbehaftet und damit eine der problematischsten Komponenten für eine Skalierung. Kapitel 12 wird sich der zentralen Frage der (Agenten-) kommunikation widmen und schlägt eine entsprechende Alternativlösung vor.

»Headless Renew« ist eine vorbereitete Version von Renew ohne grafische Oberfläche und wird in Containern bereitgestellt. Es handelt sich dabei um die Plattformdefinition im Sinne der Mushu-Architektur. Auf ihrer Basis werden neue Plattformen für die Simulation durch das Plattformmanagement erzeugt und vernichtet.

Die Renew Instanz mit GUI dient als primäre Plattform, welche eine Interaktion mit dem Nutzer vorsieht und als Start bzw. Endpunkt der verteilten Simulation fungiert. Sie wird mittels klassischer manueller Installation bereitgestellt. Es wurde bereits motiviert, dass mit dem aktuellen Stand die Notwendigkeit einer primären Plattform naheliegend ist, diese jedoch mit weiteren Überlegungen später entfallen könnte. Maßgeblich hierzu sind die Überlegungen bezüglich des Simulationsfeeds, welcher in Abschnitt 11.6 eingeführt wird sowie die bereits angesprochenen Überlegungen bezüglich der Kommunikation in Kapitel 12.

10.8. Weitere Komponenten des Plattformmanagements

Neben der zentralen Simulationsskalierung sieht die Mushu-Architektur weitere Komponenten des Plattformmanagements vor: Eine Möglichkeit, Zustandsabfragen zu erzeugen, Ablaufmanager und Nachrichtenbroker, sowie proaktives Verhalten des Plattformmanagements selbst.

Der Konstruktion von Ablaufmanager und Kommunikationsmedium widmet sich Kapitel 12. Für die weiteren Betrachtungen ist daher insbesondere relevant, wie

sich diese Komponenten bei einem entsprechenden Deployment verhalten würden. Der Ablaufmanager ist im Wesentlichen eine neu zu konstruierende Komponente und sollte daher nach Möglichkeit ebenfalls so konstruiert werden, dass er im containerisierten Umfeld einsatzfähig ist. Das Kommunikationsmedium wird aller Wahrscheinlichkeit nach als bereits bestehendes Projekt bzw. bestehende Lösung realisiert. Dadurch richtet sich die Wahl des Deployments nach den Vorgaben der konkreten technischen Umsetzung. Nach Möglichkeit sollte jedoch auch eine grundlegende Form der Virtualisierung bzw. Containerisierung verfolgt werden, um besser auf sich ändernde Situationen reagieren zu können.

Zustandsabfragen umfassen im Wesentlichen die Abfrage einzelner Plattformen. Sie sollten an einer definierten Stelle im Plattformmanagement verortet werden und können problemlos repliziert bereitgestellt werden, da sie keinen eigenen persistenten Zustand aufweisen und daher inhärent zustandslos sind.

10.8.1. Proaktives Verhalten des Plattformmanagements

Die proaktive Skalierung durch das Plattformmanagement kann auf der Basis von externen Messungen an den Plattformen realisiert werden. Im Rahmen der Überlegungen bezüglich der Realisierung des Management-Systems wurde bereits nahegelegt, einen entsprechenden Clustermanager einzusetzen. Damit dieser seine Aufgaben erfüllen kann, ist anzunehmen, dass auf den physikalischen Maschinen entsprechende Daemons des Clustermanagers installiert sind. Diese könnten entsprechende Messungen übernehmen und extern erhebbare Zustände an die Clustermanager und damit an das Management-System übermitteln.

Auf der Basis dieser übermittelten Daten können entsprechende Skalierungssollzustände durch das Management-System oder den Clustermanager abgeleitet werden. Ab diesem Punkt entspricht das weitere Vorgehen vollständig dem der reaktiven Skalierung, welches in den vergangenen Abschnitten des Kapitels ausführlich diskutiert wurde.

10.8.2. Zustandsinformationen

Die Abfrage von Zustandsinformationen ist aufgrund der dezentralen Natur der Realisierung des Plattformmanagements nicht trivial. Zunächst sollte geklärt werden, welche Daten das Plattformmanagement überhaupt aggregieren können soll.

Dafür ist es hilfreich, sich entlang der Ebenen der MUSHU-Architektur zu orientieren. Auf der obersten Ebene des Plattformmanagements selbst sind Metainformationen wie die Anzahl aller Plattformen oder Zustände der Kommunikationsinfrastruktur denkbar. Auf Basis der Ausführung bezüglich des Reconciler

Patterns innerhalb des Management-Systems wurde bereits diskutiert, dass eine Schnittstelle zur Abfrage der Ausdehnung bereits an dieser Stelle umgesetzt wird. Informationen bezüglich der Kommunikationsinfrastruktur können besser im Kontext der Agentenkommunikation behandelt werden und sollen daher in Kapitel 12 aufgegriffen werden.

Die nächste Ebene der MUSHU-Architektur umfasst die der Plattform. Dabei können vielerlei Daten von Interesse sein, wie beispielsweise Anzahl der Agenten, verfügbare Plattformfunktionalitäten, Agentendefinitionen, lokale Auslastungen und generell viele weitere Metriken, welche für spezifische Anwendungsfälle von Interesse sein können. Hierbei ist eine Umsetzung eines allgemeinen Interfaces für die Bereitstellung von Informationen denkbar. Dieses Thema wird noch einmal in Abschnitt 11.5 aufgegriffen und detailliert diskutiert werden.

Letzten Endes folgt die Ebene der Agenten. Hierbei handelt es sich im Wesentlichen um die Ebene, auf der fachliche Berechnungen ausgeführt bzw. Interaktionen simuliert werden. Folglich sind an dieser Stelle entweder fachliche Teilergebnisse von Interesse oder aber der vollständige Ablauf der Simulation. Von einem vollständigen Ablauf kann beispielsweise ein externes System profitieren, welches einem Nutzer grafisches Feedback zur verteilten Simulation liefern will. Dies wurde ebenfalls bereits in Abschnitt 10.5.1 motiviert. Aufgrund der Nebenläufigkeit und insbesondere beim Einsatz von True-Concurrency-Semantik ist dieses Unterfangen jedoch sehr komplex. Aus diesem Grund adressiert der Abschnitt 11.6 diese Thematik im Detail.

Wie bereits in der Einleitung erwähnt, sollte der Aggregationsservice selbst keinen Zustand aufweisen und sollte daher problemlos als containerisierte Anwendung bereitgestellt werden können.

10.9. Zusammenfassung - Umsetzung des Plattformmanagements

In diesem Kapitel wurden konkrete Realisierungsvorschläge für die abstrakten Komponenten des MUSHU- Plattformmanagements vorgestellt. Reaktive und proaktive Skalierung erfolgen durch einen Clustermanager und ein neu zu konstruierendes, nach dem Reconciler-Pattern arbeitendes Management-System, welches die Form eines Webservices annehmen soll. Die Integration in die Simulation erfolgt durch das Angebot einfacher Schnittstellenfunktion in Form von spezialisierten Funktionen, welche einzelnen Transitionen hinzugefügt werden können. Das Deployment aller Komponenten soll nach Möglichkeit als Container erfolgen, wodurch unter anderem die dynamische Aktualisierung der Plattformdefinition realisiert werden kann. Einzelne Komponenten weichen jedoch begründet von der

Containerisierung ab. Darüber hinaus umreißt das Kapitel die generelle Vorbereitung für die Plattform- und Kommunikationsrealisierung durch die bereitgestellte Umgebung.

11. Realisierungskonzept zur Plattform

Als zentrales Novum im Rahmen dieser Arbeit wird in diesem Abschnitt das Konzept eines Cloud-Native-Renew vorgestellt. Dabei umfasst die Grundidee den Renew Simulator in eine Struktur zu überführen, die dem Cloud-Native Paradigma entspricht. Die entsprechenden Anforderungen wurden bereits umfassend im zugehörigen Abschnitt 6.1 dargelegt.

Ein Grundgedanke des Cloud-Native-Ansatzes besteht darin, Abhängigkeiten von der zugrunde liegenden Infrastruktur zu entfernen. Die hauptsächliche Herausforderung dabei liegt darin, eine Anwendung zu konstruieren, die nicht darauf angewiesen ist, dass ein Administrator auf dem lokalen System, auf dem die Anwendung ausgeführt wird, die Anwendung verwalten kann. Dazu muss die Anwendung aus sich heraus selbst eine Administrationsschnittstelle bereitstellen, mit der sie konfiguriert und überwacht werden kann.

Die Veröffentlichung (Röwekamp, Taube u. a., 2021) des Autors umfasst die zentralen Erkenntnisse aus diesem Kapitel und wurde auf der zugehörigen Tagung mit dem Fachpublikum diskutiert.

11.1. Umsetzung der Komponenten der Plattform

Auf der Basis der Mushu-Architektur ergeben sich Aspekte, welche durch eine Plattform erfüllt werden sollten. Diese wurden bereits in den entsprechenden Kapiteln dargestellt und begründet. Dies erfolgte jedoch nur auf abstrakter Ebene. Aus diesem Grund ist es notwendig, die entsprechenden Ideen für eine konkrete Umsetzung zu entwickeln.

Bevor die einzelnen Aspekte weiter ausformuliert werden, diskutiert dieser Abschnitt zunächst die grobe Auswahl des generellen Ansatzes des jeweiligen Aspekts im Kontext von Referenznetzsimulationen und dem Simulator Renew. Diese Ansätze dienen als Grundlage für die weiteren Ausführungen innerhalb dieses Kapitels.

11.1.1. Plattformfunktionalitäten

Plattformfunktionalitäten sind im Wesentlichen Eigenschaften der Plattform. Im Kontext von Renew bestehen mehrere Möglichkeiten, um solche Funktionalitäten einzubinden.

Zunächst besteht die Möglichkeit, Plattformfunktionalitäten als entsprechendes Referenznetzen zu modellieren und durch den Renew Simulator in die Simulation eingliedern zu lassen. Ein weiterer Ansatz besteht darin, die Funktionalitäten im Quelltext des Simulators festzuschreiben. Darüber hinaus bietet Renew einen Plugin-basierten Ansatz, sodass es auch denkbar wäre, Funktionalitäten als Plugin(s) zu modellieren.

Die Darstellung als Referenznetz weist den Vorteil auf, dass sie auf derselben konzeptuellen Grundlage basiert wie die übrigen Komponenten des Simulators. Diese Darstellung ist zwangsläufige Voraussetzung für die Simulation durch den Renew Simulationskern. Wird der Renew Simulator jedoch unter dem Aspekt der Mulan- bzw. Mushu-Architektur betrachtet, entspricht die Instanziierung von Netzen der Plattform untergeordneten Entitäten. Somit wäre Implementation von Funktionalitäten als Referenznetz ein Bruch in der Umsetzung der Architektur.

Die Integration in den Quelltext des Simulators beschreibt die direkteste Integration in die Plattform. Dabei ist dieser Aspekt jedoch auch schon der Hauptvorteil des Ansatzes, da es mit einer festen Integration zu Schwierigkeiten kommt, die Heterogenität der Plattform an sich zu bewahren. Wie im Vorfeld in Kapitel 9 bereits erwähnt, soll diese erst durch Plattformfunktionalitäten hergestellt werden. Eine Direktintegration in den Quelltext ist somit kontraproduktiv und kann ausgeschlossen werden.

Zuletzt bleibt die Verwendung von Renew-Plugins. Diese weisen den Vorteil auf, dass sie als optionale Komponenten in den Simulator hineingeladen werden können. Darüber hinaus erweitern sie – bezogen auf Simulationen – die Plattform auf der Ebene der Plattform selbst und nicht etwa durch untergeordnete Entitäten. In ihrer klassischen Form (ab Renew 2.0) können Plugins zur Laufzeit zumindest nachgeladen werden.

Als bestgeeigneter Kandidat sollten Plugins daher noch einmal im Kontext der restlichen Mushu-Architektur betrachtet werden. Plattformfunktionalitäten müssen portabel sein, damit sie auf verschiedenen Plattformen eingesetzt werden können. Plugins erfüllen diese Vorgabe durch ihre Kapselung. Sie können als Artefaktarchive verschoben und an physikalisch andere Orte kopiert werden. Diese Aspekte zusammen mit dem dynamischen Laden von Plugins sind die Hauptanforderungen durch die Architektur. Würde also nicht auf das bestehende Renew Plugin-System zurückgegriffen werden, würde zwangsläufig eine Art konkurrieren-

des Plugin-System eingeführt werden, dessen Grenznutzen im besten Fall fraglich und somit nicht wünschenswert wäre.

Auf der contra Seite kann angeführt werden, dass nicht zwangsläufig alle denkbaren Funktionalitäten durch Plugins abgedeckt werden können. Dazu zählen beispielsweise bestimmte Eingriffe in den Ladeprozess von Plugins, wie etwa die Prüfung von Signaturen anderer Plugins, wie es in der Arbeit (Jürgensen, 2021) betrachtet wurde. Fälle wie diese sind jedoch im Allgemeinen selten. Darüber hinaus müssen sich Funktionalitäten der generellen Struktur von Renew-Plugins fügen und können nicht vollständig frei entwickelt werden. Die Einschränkung ist dabei jedoch allerdings nicht sehr stark, da auch die Struktur eines Renew-Plugins viele Freiheiten lässt.

Zusammenfassend lässt sich festhalten, dass die Konstruktion von Plattformfunktionalitäten durch Renew-Plugins die vielversprechendste Variante der Umsetzung ist.

11.1.2. Erweiterung von Agentendefinitionen

Der nächste zu adressierende Punkt ist die Erweiterung von Agentendefinition. Im vergangenen Abschnitt wurde bereits erläutert, dass Netze und Netzinstanzen untergeordneten Elementen bezüglich eines Renew Simulators entsprechen. Dies entspricht wiederum der Darstellung von Agenten und Plattformen innerhalb der Mushu-Architektur.

Als weiteren Aspekt definieren die einschlägigen Literaturquellen wie (Cabac, 2010), (Reese, 2009) und (Duvigneau, 2002) sowie die Mushu-Architektur selbst Agenten (im Kontext von Renew) stets als Netze. Während die Modellierung als Plugin zwar theoretisch denkbar und möglich wäre, würde eine Entscheidung für Plugins gleichzeitig jedoch die Abkehr von vielen Erkenntnissen zur referenznetzenbasierten Simulation von Agenten bedeuten. Darüber hinaus ist der intendierte Verwendungszweck von Plugins nicht die Repräsentation von Agenten.

Aus diesem Grund soll die Modellierung von Agenten durch Netze auch im Kontext dieser Arbeit erhalten bleiben. Somit kann die Anforderung zur Erweiterung von Agentendefinitionen direkt auf die Bereitstellung neuer (statischer) Netze abgebildet werden. Für diese muss der Simulator eine entsprechende Uploadfunktionalität bereitstellen.

11.1.3. Aktivierung von Agenten

Es wurde bereits festgehalten, dass die naheliegendste Umsetzung für Agentendefinitionen der Einsatz von statischen Netzen ist. Der darauffolgende Schritt, Agenten zu aktivieren, entspricht der Instanziierung von statischen Netzen. Dabei sind die wesentlichen Anforderungen insofern gegeben, als das Instanziieren neuer Netzinstanzen von außen traditionell mit dem Start einer Simulation innerhalb von Renew einhergeht. Renew selbst ist als nebenläufiges System konstruiert, der simultane Ablauf mehrerer Simulationen innerhalb einer Renew Instanz ist jedoch nicht vorgesehen. Aus diesem Grund müssen die verschiedenen Umsetzungsmöglichkeiten von externer Instanziierung diskutiert werden.

11.1.4. Status-Monitor

Der Status-Monitor in der Form, wie ihn die Mushu-Architektur vorsieht, besteht in keiner Form in gegenwärtigen Implementationen in Renew (-Plugins). Die Implementation muss daher von Grund auf neu erfolgen. Die vorgegebenen Arten von Detailinformationen umfassen dabei Informationen zum Systemzustand sowie Informationen zu den verfügbaren Plattformfunktionalitäten und Agentendefinitionen.

Für die Schnittstelle zu Systeminformationen kann dabei voraussichtlich auf fertige Bibliotheken zurückgegriffen werden. Herausfordernder gestaltet sich die Konstruktion der internen Informationsabfrage. In Abschnitt 11.1.1 wurde bereits motiviert, dass Funktionalitäten durch Plugins abgebildet werden können. Ein naheliegender Ansatz wäre es, Plugins dazu zu verpflichten Informationen über ihren internen Zustand preiszugeben. Dies würde jedoch dem Grundgedanken der Autonomie in der Umsetzung von Renew widersprechen. Eine andere Möglichkeit wäre die Bereitstellung einer internen Schnittstelle, über welche Plattformfuntkionalitäten ihren internen Zustand veröffentlichen können. Dies würde jedoch wiederum ggf. zu Problemen in der Abhängigkeitsstruktur führen. Abschnitt 11.5 diskutiert die Details zu dieser Problematik.

11.1.5. Umzug von Agenten

Mit den bisher umgesetzten Konzepten zur Bereitstellung von neuen Netzdefinitionen und der Aktivierung von Agenten kann der Umzug eines Agenten vergleichsweise komfortabel durch die Kombination dieser beiden Ansätze bewerkstelligt werden. Aus diesem Grund genügt es, dass der Umzug von Agenten indirekt im Kontext dieser beiden Elemente betrachtet wird.

Zu einem vollwertigen Umzug zählt neben der Bereitstellung der Definition des Agenten und seiner Aktivierung ebenfalls der Transfer seiner Wissensbasis. Dieser Transfer kann jedoch über klassische Agentennachrichten erfolgen, welche der neuen Plattform zugestellt werden. Daher ist für diesen Aspekt ebenfalls keine separate Betrachtung notwendig.

11.1.6. Weitere Aspekte durch Cloud-Nativity

Wie in den ersten Teilen der Arbeit beschrieben, diente der Ansatz der Cloud-Nativity als maßgeblicher Einfluss bei der Konstruktion der Mushu-Plattform. Nicht alle Aspekte wurden direkt innerhalb der Architektur abgebildet, da sie teils sehr technisch motiviert waren. Einige davon besitzen jedoch sehr hilfreiche Eigenschaften für die Realisierung einer Mushu-Plattform. Aus diesem Grund sollen diese auch an dieser Stelle noch einmal aufgegriffen werden und mögliche Lösung für die Realisierung diskutiert werden.

Simulationsfeed

Einer der spannendsten Aspekte ist die Beobachtbarkeit der Simulation selbst. Dabei umfasst das erklärte Ziel, einem externen Beobachter Zugriff auf eine Darstellung aller Ereignisse der Simulation zu verschaffen. Aus Sicht der Mushu-Architektur handelt es sich hierbei jedoch um keinen konzeptuell neuen Aspekt, da ein Beobachter als Agent oder Plattform in einem weiteren Plattformmanagement interpretiert werden kann. Die Bereitstellung eines derartigen Simulationsfeeds ist auf abstrakter Ebene daher nichts weiter als die gebündelte Bereitstellung aller Ereignisse auf allen Plattformen des lokalen Plattformmanagements durch Nachrichten an ein externes beobachtendes Plattformmanagement, bzw. einem auf einer seiner Plattformen agierenden Agenten.

Obwohl die konzeptuelle Darstellung in wenigen Worten beschrieben werden kann, gestaltet sich die konkrete Realisierung deutlich schwieriger. Zunächst müssen Daten und Feuervorgänge vereinfacht werden, da ansonsten der gesamte Schaltaufwand auch hätte durch den externen Beobachter erfolgen können und die Vorteile der Verteilung hinfällig wären. Darüber hinaus können und werden gerade bei stark skalierenden Simulationen extrem viele Daten anfallen. Es müssen entsprechende Systeme bereitgestellt werden, die mit einer solchen Datenflut umgehen können. Zentralisierte Systeme sind dabei unter Last zwangsläufig ein Flaschenhals für die gesamte (globale) Berechnung.

Eine weitere Problematik besteht darin, dass Referenznetzsimulationen mit Renew im Normalfall nach Partial Order bzw. True-Concurrency-Semantik geschaltet werden. Dieser Semantik haftet an, dass es zu keinem Zeitpunkt einen eindeu-

tig definierten Zustand aller Markierungen und Schaltvorgänge geben muss. Der Einsatz von einer lokalen Interleaving- oder Step-Semantik bremst das System deutlich aus. Der Einsatz von globaler Interleaving- oder Step-Semantik würde die gesamte Verteilung hinfällig machen oder deren Wert abseits spezifischer heterogener und lokal-spezifischer Plattformfunktionalität zumindest massiv infrage stellen.

Eine entsprechende Konzeption eines Vorschlags für einen Simulationsfeed für externe Beobachter stellt Abschnitt 11.6 dar.

Abhärtung

Ein weiterer sinnvoller Aspekt betrifft die Abhärtung des Systems und der Kommunikation. In diesem Kontext ist insbesondere das Distribute Plugin erwähnenswert, da es die interne Kommunikation zwischen einzelnen Netzinstanzen darstellt.

In seiner gegenwärtigen Fassung basiert das Distribute Plugin auf Java RMI und damit einer zentralen Registry. Der Ausfall dieser Registry hat einen kaskadierenden Fehlschlag des Gesamtsystems zur Folge, welcher nach den Grundüberlegungen der Abhärtung vermieden werden muss. Darüber hinaus kann das Distribute Plugin nicht zwischen aktuellen und vergangenen Simulationen unterscheiden. Daraus entstehen potenziell Inkonsistenzen zwischen den Datenbeständen oder generelle Fehler.

Eine wichtige Vorarbeit für die Konstruktion eines abgehärteten Distributes stellt die Umsetzung zu komplexer Agentenkommunikation in Kapitel 5 sowie die diesbezügliche Umsetzung in Kapitel 12 dar. Da diese Überlegungen unerlässlich für die Konstruktion sind, wird die Thematik am Ende des entsprechenden Kapitels in Abschnitt 12.7 erneut aufgegriffen werden.

11.2. Uploadfunktionalität

In den vergangenen Abschnitten wurde bereits hergeleitet, dass Plattformfunktionalitäten durch Plugins und Agenten durch Netze bzw. aktivierte Agenten durch Netzinstanzen dargestellt werden sollen. Des Weiteren umfasst die Mushu-Architektur die Möglichkeit zur Erweiterung von sowohl Plattformfunktionalitäten als auch Agentendefinitionen. Aus diesem Grund sollte eine Umsetzung der Architektur ebenfalls eine entsprechende Möglichkeit der Erweiterung anbieten. Da viele Betrachtungen im Rahmen der Umsetzung bereits im Kontext der Webservices erfolgt ist, liegt es nahe, die Möglichkeit der Erweiterung als »Uploadfunktionalität« zu beschreiben.

Plugins werden im Kontext von Renew stets als Java Archive bereitgestellt. Beim Start einer Renew Plattform werden alle ansässigen Plugins entsprechend geladen und stehen mit ihrer Funktionalität allen Simulationen und ihren Bestandteilen zur Verfügung. Die Anwendung sollte zur Erweiterung daher eine einfache Schnittstelle bereithalten, welche entsprechende Archive entgegennimmt und zusätzlich zu den bereits existierenden Plugins ablegt.

Analog verhält es sich dabei mit den Definitionen von Netzen. Diese können von Renew als textbasierte Darstellung gesichert werden und umfassen dabei alle Informationen und Darstellungsdetails wie beispielsweise Koordinaten von einzelnen Objekten des Netzes. Als Vorstufe zur Ausführung von Netzen können diese durch Renew in ein sogenanntes Schattennetz kompiliert werden. Schattennetze beschränkt sich auf die wesentlichen Aspekte von Transitionen, Plätzen und ihren Verbindungen, beinhalten keine grafischen Informationen und können in einem Binärformat gesichert werden. Schattennetze oder Systeme aus Schattennetzen (engl.: Shadow Net Systems) sind daher prädestiniert für den hier gewünschten Einsatzzweck.

Zusammengefasst wird also eine Schnittstelle benötigt, welche sowohl Java Archive als auch Schattennetzsysteme akzeptiert und in der lokalen Umgebung der Plattform ablegen kann.

11.3. Laden von Plugins

Das Laden von Plugins ist eine bereits vorhandene Funktionalität des Simulators Renew, auf die zurückgegriffen werden kann. Neu ist dabei allerdings die Fähigkeit, das Laden von Plugins aus einer externen Quelle anzustoßen. Aus diesem Grund muss eine entsprechende Verbindung zwischen externer Schnittstelle und dieser Funktionalität geschaffen werden. Dabei bietet es sich an, einen weiteren Endpunkt in den Uploadcontroller einzufügen.

Beim Laden von Plattformfunktionalitäten und auch Agentendefinitionen muss die Vertrauenswürdigkeit des hochladenden Kommunikationspartners angenommen werden. Dies liegt darin begründet, dass die so beschriebene Bereitstellung von Java Archiven im Kern der Upload von Turing-mächtigem Code ist. Ein etwaiger Angreifer hätte somit leichtes Spiel, einen Simulator zur Ausführung von Malware zweckzuentfremden. Der Aspekt der Sicherheit in diesem Kontext unterscheidet sich nicht grundlegend von anderen ähnlichen Szenarien, bei denen dynamisch Code zur Verfügung gestellt wird. Dabei handelt es sich um ein zentrales Forschungsfeld im Kontext der IT-Sicherheit und es existieren unzählige Beiträge zu dem Thema. Da IT-Sicherheit nicht der Kernaspekt dieser Arbeit ist, sei für weitere Überlegungen diesbezüglich auf entsprechende Literatur ver-

wiesen. Die Arbeit (Jürgensen, 2021) beispielsweise umfasst den Einsatz von Signaturen, um die Sicherheitsproblematik abzumildern.

Während das Laden von Plugins von der technischen Realisierung aus betrachtet keine große Herausforderung darstellt, ist das Entladen von Plugins deutlich komplexer. Dies kann unter Umständen ein wünschenswertes Feature darstellen, um beispielsweise Plugins zur Laufzeit durch eine neuere Version zu ersetzen. Aber auch andere Szenarien sind denkbar, bei denen einzelne Plugins ein bestimmtes Muster implementieren und je nach aktueller Aufgabe der Plattform eine andere Ausprägung aufweisen können. Muss diese Ausprägung geändert werden, kann es notwendig sein, das entsprechende Plugin wieder zu entladen.

Das Entladen von Plugins ist insoweit problematisch, als zugehöriger Programmcode in verschiedenen Teilen der Anwendung in Verwendung sein kann. Eine gezielte Entkopplung aus der laufenden Anwendung ist erforderlich, um Plugins zu entfernen.

Eine Entwicklung in diese Richtung wurde mithilfe des Java Modulsystems in der Arbeit (Janneck, 2021) vorgestellt. Dabei dienen Module der starken Kapselung einzelner Bereiche von Funktionalitäten, wobei die Kapselung über die bereits existierende Kapselung durch das Plugin-System hinausgeht. Module sind ein Konzept, welches vor allem zur Kompilierzeit relevant ist, aber zur Laufzeit für das dynamische Laden von Code nicht mehr erzwungen wird. Zu diesem Zweck existiert bei Java das Modullayer-System, welches das Konzept der einzelnen Module auf dynamisch zur Laufzeit geladenen Code ausdehnt. Das modulare Renew, welches in Renew 4.0 umgesetzt ist, implementiert sowohl das Modul- als auch das Modullayer-System. In diesem Kontext wäre es denkbar somit auch das Entladen von Plugins zu unterstützen.

11.4. Laden von Netzen

Das Laden von Netzen erscheint auf den ersten Blick nicht sehr unterschiedlich zum Laden von Plugins. Im Detail ist das Laden von Netzen jedoch vergleichbar mit der Instanziierung von Objekten, während das Laden von Plugins mit der Erweiterung statischer Funktionalität vergleichbar ist.

Referenznetzsimulationen beginnen im Normalfall mit einer einzelnen Netzinstanz, welche dann während ihrer Simulation weitere Netzinstanzen erzeugen kann. Die Instanziierung eines Netzes ohne eine weitere Netzinstanz als Ausgangspunkt kann somit auf verschiedene Weisen umgesetzt werden. Da diese sehr unterschiedliche Implikationen aufweisen, sollten die verschiedenen Möglichkeiten separat diskutiert werden.

Der intuitive Ansatz ist die Erweiterung der Simulation um eine weitere Netzinstanz analog zur Instantiierung durch eine bestehende Netzinstanz. In vielen Fällen ist dies unproblematisch, die Schwierigkeit entsteht jedoch durch einen Bruch in den betrachteten Ebenen. Während die Instantiierung durch eine Netzinstanz aus der Anwendung (Simulation) selbst erfolgt, ist das Instantiieren ohne ausgehende Netzinstanz aus Sicht der Mushu-Architektur ein Eingriff der Plattform in die Ebene der Agenten. Aus Mushu-Sicht ist dies gewollt, im Modell abgebildet und unproblematisch, da Agenten als autonome Entitäten konzipiert sind. Bei der Verwendung statischer Daten durch Netzinstanzen kann dies im Allgemeinen jedoch durchaus zu Problemen führen, da die Plattform selbst kein anwendungsspezifisches Wissen besitzt und besitzen soll.

Eine weitere Möglichkeit der Interpretation beinhaltet, dass die Instantiierung eines neuen Netzes gleich dem Start einer neuen, separaten Simulation gleicht. In diesem Kontext würden aber mehrere Simulationen nebenläufig auf einem Simulator bzw. einer Plattform ausgeführt werden. Konzeptuell ist dies problemlos möglich, technisch entstehen jedoch einige Hürden, die im Folgenden erörtert werden.

11.4.1. Nebenläufige Referenznetzsimulationen

Petrinetze und damit insbesondere auch Referenznetze sind als nebenläufige Strukturen konzipiert. Daher macht die Ausführung nebenläufiger Simulationen von Referenznetzen auf konzeptueller Ebene wenig Probleme. Tatsächlich gestaltet sich die technische Umsetzung jedoch etwas komplizierter. Dieses Problem wird sich erneut im Kontext von Kapitel 12 ergeben, sodass die hier geschilderten Überlegungen auch dort wieder aufgegriffen werden.

Eine grundlegendere Frage umfasst den Grad der Loslösung von gleichartigen Prozessen und Prozessstrukturen der beiden Referenznetzsimulationen. Konkret stehen die folgenden verschiedenen Ansätze zur Verfügung: die Umsetzung von nebenläufigen Simulationsprozessen, die Umsetzung von nebenläufigen Simulatoren in einem Simulationsprozess sowie die nebenläufige Simulation innerhalb eines Simulators.

Simulationen in einem Simulator Das generelle Ausführen mehrfacher nebenläufiger Simulationen innerhalb eines einzigen Simulators dürfte auf den ersten Blick problemlos möglich sein. Wie bereits einführend erläutert, sind Petrinetze nebenläufige Strukturen und zugehörige Simulatoren sollten daher ebenfalls nebenläufig aufgebaut sein. Wird eine weitere Netzinstanz innerhalb einer Referenznetzsimulation erzeugt, so kommt dies dem Erzeugen der initialen Netzinstanz in einer neu gestarteten Simulation gleich. Alle Konstrukte wie Netzinstanzen oder Objekte, die durch die Simulation der

neuen Netzinstanzen erzeugt werden, können keine Referenzen auf die Objekte der bereits innerhalb des Simulators bestehenden Simulation besitzen. Ebenso können Objekte der alten Simulation keine Referenzen auf die der neuen Simulation besitzen. Dadurch entsteht innerhalb des Simulators ein abgetrenntes Ökosystem aus Referenzen.

An dieser Stelle muss jedoch deutlich darauf hingewiesen werden, dass diese ideale Vorstellung nur dann haltbar ist, wenn weder in den Netzen noch in den dahinterliegenden Objekten auf statische Strukturen zugegriffen wird, welche nur einmal pro Simulator vorgehalten werden. Dabei ist explizit nicht eine Beschränkung auf beispielsweise das `static` Schlüsselwort gemeint, welches in vielen Sprachen eingesetzt wird, um statische Konstrukte zu beschreiben, sondern auch entsprechende Entwicklungsmuster wie beispielsweise Singletons. Auch der Zugriff auf Datenbanken oder auf Web- oder sonstige Netzwerkressourcen kann unter dem Aspekt der statischen Inhalte betrachtet werden. Durch statische Inhalte können sich Referenzen der beiden getrennten Ökosysteme vermischen und es kann ohne anwendungsseitige Vorbereitung zu unvorhersehbaren Seiteneffekten kommen. Dies betrifft insbesondere die Simulation von zwei Referenznetzen, die ohne Koexistenz innerhalb eines Simulators jeweils fehlerfrei simuliert werden konnten.

In Simulationen, in denen die Abhängigkeit von statischen Strukturen nicht direkt gegeben ist oder der Fall der externen Instantiierung anwendungsseitig vorgesehen ist, ist dieser Ansatz daher eine valide Möglichkeit, effizient mehrere Simulationen zu integrieren, ohne nennenswerten Mehraufwand zu betreiben. Die Anwendung im Agentenkontext und Mushu ist solch ein Fall.

Simulatoren in einem Prozess Ein weiterer Ansatzpunkt ist die Nutzung mehrerer Simulatoren in einem Simulationsprozess. Bei diesem Ansatz würde eine eigene Simulatorstruktur für jede Simulation erzeugt werden, wobei beide die gleiche Umgebung im Prozess nutzen könnten. Ein Vorteil entsteht direkt durch die gemeinsame Nutzung von bereitgestellten Systemressourcen und der vereinfachten Kommunikationen zwischen den Simulationen, sofern diese beabsichtigt sind. Die Methode hat damit Analogien zur Virtualisierung auf der Betriebssystemebene, beschränkt sich dabei jedoch auf die Interna der Anwendung.

Die Argumentation aus dem vorherigen Ansatz bleibt jedoch bestehen, dass statische Strukturen potenziell für Probleme sorgen können. Eine weitere Problematik entsteht dadurch, dass der Simulationsprozess von sich aus darauf ausgelegt sein muss, mehrfache Simulatoren zulassen zu können. Insbesondere muss der Simulationsprozess daher so konzipiert sein, dass etwaige Threadpools, Daten und Netzwerkzugriffe und generell alle Klassen für die Repräsentation eines Simulators so konzipiert sind, dass davon nebenläufig

mehrere Objekte existieren können. Als weiterer Nachteil ist zu nennen, dass beide Simulationen immer noch eng aneinandergekoppelt sind und dass der Ausfall des Simulationsprozesses den Ausfall beider Simulationen mit sich ziehen würde.

Wird darüber hinaus der Blickwinkel der Erzeugung einer weiteren Netzinstanz verlassen und Prozess und Simulator(en) als Einheit betrachtet, stellt sich die Frage nach der Sinnhaftigkeit des Ansatzes, da ein einzelner Simulator bereits ein beliebig[1] nebenläufiges System ist.

Es fällt schwer, einen geeigneten Anwendungsfall zu formulieren, bei dem es sinnvoll sein könnte, mehrere Simulatoren innerhalb eines Prozesses zu präferieren, da viele Nachteile nur einigen Vorteilen gegenüber stehen.

Prozesse auf einer physikalischen Recheneinheit Der letzte Ansatz setzt fokussiert auf Entkopplung aller Strukturen und schlägt den Einsatz mehrerer Prozesse vor. Dabei erhält jede Simulation ihren eigenen Simulationsprozess. Durch die Entkopplung entfällt die zuvor beschriebene Problematik mit statischen Strukturen, da diese für jeden einzelnen Prozess neu erstellt werden. Zu den Nachteilen zählt ein erhöhter Ressourcenbedarf und etwaige erschwerte Kommunikation zwischen Simulationen, sofern diese gewünscht ist.

Ein gewaltiger Vorteil neben der Unabhängigkeit von statischen Inhalten bei dieser Methode besteht aber darin, dass diese Prozesse potenziell auf verschiedenen physikalischen Maschinen ausgeführt werden können, aber nicht müssen. Eine Herausforderung bei diesem Ansatz ist die dynamische Erzeugung eines neuen Simulationsprozesses, sobald dieser benötigt wird.

In Kapitel 10 wurde bereits ausführlich die Gestaltung der Simulationsumgebung für eine skalierende Simulation dargelegt. Zusammenfassend ergab sich dort die Containerisierung als eine der vielversprechendsten Technologien für die Ausführung von skalierenden verteilten Referenznetzsimulationen. Unter dem Aspekt der Erweiterung der aktuell betrachteten Simulation, kann die Technologie auch problemlos eingesetzt werden, um eine weitere unabhängige Simulation zu starten.

Gepaart mit dem Einsatz eines entsprechenden Containermanagers kann somit eine noch stärkere Entkopplung erreicht werden. Dies ergibt sich daraus, dass durch den Containermanager der neue Simulationsprozess ohne nennenswertes Zutun auf einem anderen physikalischen Rechner innerhalb des Gesamtsystems hochgefahren werden kann.

Ein Nachteil würde insofern auftreten, als die Umsetzung mit mehreren Prozessen die direkte Zuordnung zwischen Prozess und Mushu-Plattform

[1] Simulatorseitig beliebig, jedoch praktisch beschränkt durch die Anwendung

zerstört. Eine Mushu-Plattform wäre in der hier vorgeschlagenen Umsetzung somit ein logisches Konstrukt wie das Mushu-Plattformmanagement. Daraus ergeben sich einige weitere Probleme, wie etwa die plattforminterne Kommunikation, welche somit nicht mehr uniform gelöst werden kann, da im Zweifel Interprozesskommunikation zum Einsatz kommen muss. Dabei muss es sich nicht um einen Nachteil handeln, wenn eine effiziente Form der Interprozesskommunikation zum Einsatz kommt. Erste Untersuchungen in diese Richtung bietet die Arbeit (Feldmann, 2019).

Unabhängig von der konkreten Umsetzung muss dann jedoch zusätzlich die Barriere der Containerisierung überwunden werden. Während dies prinzipiell möglich sein sollte, führt es jedoch einen zusätzlichen Overhead an Komplexität ein.

Eine andere Form der Interpretation ist es, die Trennung der Simulatoren auf konzeptueller Ebene dem Erzeugen einer neuen, separaten Plattform gleichzusetzen. Diese Interpretation ist für die Umsetzung der Mushu-Architektur jedoch problematisch, da das Aktivieren eines Agenten die Konstruktion einer neuen Plattform erzwingen würde. Dieses Verhalten ist weder gewünscht, noch hilfreich im Kontext von Mushu.

Dennoch sollte dieser Ansatz nicht vorschnell verworfen werden, da durchaus Einsatzmöglichkeiten existieren. Weitere Ausführungen hierzu folgen in Kapitel 12.

11.4.2. Laden von Netzen im Kontext von Mushu

Abschließend kann das Laden von Netzen erneut im Kontext der Mushu-Architektur erörtert werden. Die Architektur der Anwendungsdomäne ist durch die Vorgabe der Mushu-Architektur insoweit eingeschränkt, als Netzinstanzen bzw. aktive Agenten autonome Entitäten darstellen, welche sich direkt keine statischen Strukturen teilen sollten. Vorhandene statische Strukturen der Plattform können und müssen auf geordnete Weise angesprochen werden.

Dies erfolgt im Allgemeinen durch Nachrichten, welche sowohl zwischen Plattform und Agent als auch zwischen zwei Agenten ausgetauscht werden können. Die verzahnte Abarbeitung von Nachrichten verschiedener Quellen muss ebenfalls anwendungsseitig unterstützt werden, da bei der Bearbeitung kein grundlegender Unterschied zwischen der Herkunft der Nachricht gemacht werden soll. Somit werden beispielsweise Nachrichten von Agenten auf anderen Plattform gleichermaßen behandelt wie Nachrichten von lokalen Agenten.

Ein weiterer Aspekt ist die Bereitstellung von Plattformfunktionalitäten durch die Plattform. Diese stammen aus Bibliotheken, spezialisierten Treibern oder sonsti-

gen anwendungsspezifischen Implementationen. In jedem Fall muss die Auswahl oder Implementation mit Bedacht auf die intendierte Anwendung erfolgen, sodass Plattformfunktionalitäten im Rahmen der Mushu-Architektur auf natürliche Weise robust gegenüber nebenläufigem Zugriff sein müssen. Eine Besonderheit weisen hierbei zustandslose Plattformfunktionalitäten auf, da diese durch das Fehlen eines internen Zustands stets ein transparentes Verhalten gegenüber nebenläufigem Zugriff erlauben, unabhängig von der Anwendungsarchitektur.

Zusammengefasst kann somit festgehalten werden, dass das Erzeugen einer weiteren Netzinstanz innerhalb einer laufenden Simulation im Kontext von Mushu und der Aktivierung von Agenten unproblematisch ist. Die neu aktivierten Agenten müssen sich jedoch gezielt bei der Plattform registrieren. Die Registrierung für Agenten auf der Plattform muss darauf ausgelegt sein, weitere Agenten zu integrieren.

11.5. Statusmonitor

Die Umsetzung der Statusmonitor-Funktionalität ermöglicht es einem externen Beobachter, Informationen darüber zu erhalten, ob sich die Plattform in einem ordnungsgemäßen und funktionsfähigen (»gesundem«) Zustand befindet. Sowohl durch das Referenzparadigma der Cloud-Nativity als auch durch die Sicht der Plattform als Agent sollte die Plattform selbst in der Lage sein, Nachrichten über diese Eigenschaften zu versenden.

Vonseiten des Inhalts dieser Zustandsinformationen kann zwischen Informationen bezüglich der Umgebung der Plattform und Informationen bezüglich der Interna der Plattform unterschieden werden. Für externe Informationen sollten generelle Schnittstellen und Bibliotheken zum Einsatz kommen, da das der Plattform zugrunde liegende System einerseits nicht innerhalb der Mushu-Architektur adressiert ist und daher ohne Einschränkungen abgefragt werden kann und andererseits, da diese Anforderung keine Neuerung im Rahmen der Architektur darstellt und daher bereits vielfach verfügbar ist.

Die Konstruktion der Abfrage der inneren Daten gestaltet sich jedoch als weniger geradlinig. Insgesamt sollte das autonome Grundprinzip der Agententechnik und damit der Basis der Mushu-Architektur nicht verletzt werden. Können Agenten zu jeder Zeit vollständig überwacht und ihre Interna ausgelesen werden, so ist es für sie schwierig, autonom zu agieren. Ebenso wäre dies ein Bruch der Prämisse eines Multiagentensystems.

11.5.1. Externe Daten

Die Bereitstellung von externen Daten ist insbesondere für die proaktive Skalierungsmöglichkeit durch das Plattformmanagement von Interesse. Dabei entscheidet das Plattformmanagement auf Basis von abstrakten, externen Faktoren bezüglich seiner verwalteten Plattformen, inwieweit das Hinzuschalten oder Entfernen einer Plattform angebracht ist.

Dabei ist insbesondere die technische Umgebung der Plattform von Interesse. Allgemeingültige externe Faktoren umfassen beispielsweise die Auslastung des Prozessors sowie die Auslastung des Hauptspeichers. Für eine Entscheidung durch das Plattformmanagement sind dabei aber auch insbesondere Durchschnittswerte der jüngsten Vergangenheit von Interesse. Dadurch können punktuelle Verzerrungen, die beispielsweise durch den Zugriff selbst erfolgen, vermindert werden.

11.5.2. Interne Daten

Bei der Bereitstellung von internen Daten gilt es zunächst zu identifizieren, welche Daten extern eine gewinnbringende Sicht auf den Zustand des Simulators ermöglichen. Wie bei allen Plugin-basierten Systemen ist diese Frage nicht direkt und leicht zu beantworten, da die Gesamtheit aller Plattformen einen heterogenen Charakter aufweist.

Im Falle eines Petrinetzsimulators liegt es nahe, Informationen aus der Simulationskontrolle selbst zu extrahieren, wie beispielsweise Anzahl der aktivierten Transitionen, Feuervorgänge der letzten betrachteten Zeitperiode, Anzahl an Marken auf Plätzen und ähnliche Daten. Erweiterte Analyseaufgaben könnten beispielsweise die Berechnung von Invarianten in dem (dynamischen) System aus Netzinstanzen umfassen.

Im Spezialfall von Renew ist der gesamte Simulationsprozess jedoch selbst wiederum auf Basis eines Plugins implementiert. Andere, nicht obligatorische Plugins könnten in einer speziellen Anwendung jedoch ebenso Informationen beitragen. Aus diesem Grund sollte die Implementation abstrakter erfolgen und eher als allgemeine Schnittstelle innerhalb der Anwendung für die Bereitstellung von Informationen konstruiert werden, welche dann wiederum beispielsweise durch das Simulator-Plugin verwendet werden kann. Diese Konstruktion kann auf verschiedenen Arten und Weisen erfolgen. Die hier betrachteten Varianten umfassen die (Zwangs-)integration in jedes Plugin, die Bereitstellung an der zentralen externen (Web-)schnittstelle des Simulators, die Präsentation in den nativen Code der Basis sowie die Konstruktion eines separaten Systems auf Plugin-Basis.

Pluginbasiert Die nahe liegende Variante der Implementation umfasst die Integration in jedes Plugin. Dabei kann die generelle Struktur des Plugins auf

einfache Art und Weise erweitert werden, sodass jedes Plugin eine Methode implementieren muss, mit der aktuelle Informationen ausgelesen werden können.

Während dieser Ansatz auf den ersten Blick intuitiv und logisch erscheint, besitzt er jedoch einige Schwächen. Die größte Schwierigkeit besteht darin, dass mit der Änderung Plugins zu einer grundlegenden Form von Auskunft gezwungen werden und die Möglichkeit der Informationsbereitstellung nicht freiwillig ist. Darüber hinaus würde die Möglichkeit, Informationen bereitzustellen, eins zu eins auf Plugins abgebildet. Dabei ist jedoch problemlos vorstellbar, dass einzelne Plugins mehrere Teilbereiche aufweisen, welche sehr unterschiedliche Information bereitstellen können und andere Plugins hingegen gar keine. Darüber hinaus wäre speziell im Kontext von Renew eine Nachimplementation aller bestehenden und optionalen Plugins notwendig, damit diese kompatibel bleiben.

Auch im Kontext der Mushu-Architektur ist dieser Ansatz problematisch, da wie zuvor beschrieben, Plattformfunktionalitäten durch Plugins abgebildet werden sollten. Diese wiederum kapseln Funktionen und weisen einige Vorteile auf, sofern sie zustandslos implementiert sind, wenngleich dies keine feste Voraussetzung ist. Zustandslose Komponenten können logischerweise keine Auskunft über Zustände tätigen. Dennoch ist die Existenz von zustandsbehafteten Funktionalitäten, wie beispielsweise dem Simulationsalgorithmus selbst, durchaus anzunehmen.

Integriert in die Schnittstelle Eine weitere Möglichkeit besteht darin, die Konstruktion von Statusmeldungen direkt in die Schnittstelle nach außen zu integrieren. In diesem Fall könnte die Funktionalität unmittelbar zusammen mit der Schnittstelle bereitgestellt werden. Es müsste keine weitere Komponente für diesen Zweck geladen werden.

Dadurch ergibt sich allerdings der Effekt, dass die Schnittstelle ihrerseits wiederum intern auf alle anderen Komponenten zugreifen können muss. Dadurch wird innerhalb der Plattform eine starke Zentralisierung erwirkt und die föderale Struktur des Plugin-Systems geht an dieser Stelle verloren. Darüber hinaus wäre die Struktur der Abhängigkeiten schwierig zu realisieren, da die Schnittstelle Abhängigkeiten in viele Komponenten der Anwendung besitzt und mitunter Komponenten ebenfalls auf die Schnittstelle zugreifen müssen. Dies führt zu zyklischen Abhängigkeiten, welche viele Probleme mit sich bringen und selten gewünscht sind. Insbesondere im Kontext von Renew 4.0 werden zyklisch Abhängigkeiten bewusst und gezielt nicht unterstützt.

Eine andere Problematik liegt darin, dass die Schnittstelle auf private Informationen anderer Komponenten zugreifen müsste. Dieser Zusammenhang

sollte allerdings wie auch zuvor beschrieben vermieden werden. Ferner wäre die Implementation von Zustandsinformationen lediglich der Schnittstelle vorbehalten und es würde keine allgemeine intern verwendbare Möglichkeit geschaffen werden, Statusinformationen der Komponenten abzufragen, an die später weitere Arbeiten anschließen könnten.

Nativ Die native Implementation würde eine Integration in die grundlegende Codebasis mit sich ziehen. Die Möglichkeit, Informationen abzufragen, würde somit nicht in Form von hinzuschaltbarer Funktionalität erfolgen, sondern grundlegend für alle Elemente der Plattform zur Verfügung stehen. Auf den ersten Blick klingt dies nach einer erstrebenswerten Lösung. Abhängigkeitsprobleme würden verhindert werden, da ohnehin jedes Plugin von den Kernkomponenten abhängt. Der einfache Durchgriff auf Interna von Komponenten wäre jedoch nicht mehr leicht zu realisieren, da die grundlegende Codebasis kein Wissen über die möglicherweise geladenen Plugins besitzen soll.

Darüber hinaus ist fraglich, warum eine solche Funktionalität auf der untersten Ebene angesiedelt sein sollte. An dieser Stelle sollte nur solch eine Funktionalität bereitstehen, bei der zwangsläufige Gründe dafür bestehen. Ein Beispiel für solche Gründe kann in Sicherheitsbetrachtungen liegen, falls beispielsweise nur signierte Plugins geladen werden sollen. Diese Funktionalität müsste bereits dann bereitstehen, wenn noch kein einziges Plugin geladen ist. Nach bestem Wissen ist diese Gegebenheit für ein Statusmonitor jedoch nicht erfüllt. Darüber hinaus bleiben die problematischen Zentralisierungsgedanken aus dem vorhergegangenen Abschnitt zusätzlich bestehen.

Separates System mit Weiterleitung Die letzte betrachtete Variante zur Umsetzung besteht darin, ein separates System zum Sammeln der Statusinformationen bereitzustellen. Dabei sollen die Informationen nicht gecacht werden, sondern lediglich eine dynamische Weiterleitung zwischen Informationsproduzenten und Informationskonsumenten bereitgestellt werden. Den Produzenten entsprechen in diesem Fall die Komponenten, welche Informationen bereitstellen können. Im Kontext der hier besprochenen Implementation entspricht der Konsument der Schnittstelle, welche Anfragen nach außen beantworten kann. Geht die externe Anfrage nach Informationen beim Informationskonsumenten ein, fragt dieser alle registrierten Informationsproduzenten an. Der Ansatz hat damit gewisse Ähnlichkeiten mit einem klassischen Publish-Subscribe Ansatz, arbeitet jedoch mit Polling.

Dieser Ansatz ist aufwendiger als die anderen Ansätze, bietet jedoch einige Vorteile. Somit weisen sowohl Informationskonsumenten als auch Informationsproduzenten nur Abhängigkeiten zum neuen System auf. Auf diese Weise entstehen keine zyklischen Abhängigkeiten im System. Zusätzlich muss die externe Schnittstelle nicht geladen werden, damit Plugins, welche potenziell

Auskünfte erteilen wollen, geladen werden können. Dazu genügt das Laden des neu zu konstruieren Systems. Da die Schnittstelle für Informationskonsumenten ebenfalls allgemein gehalten ist, ist eine beliebige Aufbereitung der Informationen durch andere Komponenten der Anwendung problemlos umsetzbar. Da das System wie oben beschrieben auf der Basis von Polling durch die Informationskonsumenten implementiert sein soll, entsteht keinerlei Mehraufwand in den Komponenten, falls keine Informationskonsumenten registriert sind.

Während das neue einzelne System immer noch wie ein zentralisierter Ansatz wirkt, bietet das System jedoch nur eine Registrierung für die Komponenten und eine Weiterleitung zwischen den Komponenten. Die eigentliche Bereitstellung der Informationen geschieht innerhalb der Komponenten unter deren Maßgaben. Letzten Endes entsteht dadurch auch der Effekt, dass keine Eins-zu-eins-Beziehung zwischen Plugin und Informationsbereitstellung besteht. Dadurch entsteht die Flexibilität, dass Plugins wahlfrei keinen, einen oder mehrere Sätze an verschiedenen Informationen bereitstellen können.

Zusammengefasst fällt es schwer, eine andere Methode als das separate System mit Weiterleitung zu bevorzugen. Der Ansatz löst alle wesentlichen Probleme von zyklischen Abhängigkeiten, Zentralisierung und unfreiwilliger Teilnahme. Der Anteil des Statusmonitors für interne Metriken sollte daher nach diesem Ansatz erfolgen.

11.6. Simulationsfeed

Der letzte im Kontext der Beobachtbarkeit beschriebene Punkt ist die Konstruktion eines Simulationsfeeds. Wie zuvor eingeleitet wurde, ist die Konstruktion eines solchen Simulationsfeeds nicht trivial. Die zentralen Probleme dabei liegen in der Handhabung der True-Concurrency-Semantik bei der Simulation des (globalen) Referenznetzes sowie die verteilte Berechnung und damit potenzielle Ausnutzung großer Rechenleistung und die verbunden damit entstehende Menge an Daten.

Dennoch ist die Bereitstellung eines Simulationsfeeds für weitere Anwendung äußerst hilfreich. Auf dieser Basis sind zum Beispiel grafische Oberflächen, Erweiterung des Plattformmanagements oder Aussagen zur Gesamtperformance des Systems möglich. Im konzeptuellen Teil der Arbeit wurde außerdem motiviert, dass ein Simulationsfeed zur Konstruktion holonischer Multiagentensysteme verwendet werden kann.

In Renew sind bereits einzelne Plattformen (Simulatoren) durch Renew Remote beobachtbar. Dafür ist lokal eine Java- und Renew-Installation erforderlich.

Das Plugin basiert auf Java RMI. Die Bereitstellung eines Simulationsfeeds entspricht somit in gewissem Maße der Erweiterung von Renew Remote auf ganze Plattformmanagements.

Die dabei entstehenden Herausforderungen und entsprechende Lösungsvorschläge werden im Rahmen dieses Abschnitts diskutiert.

11.6.1. Vorüberlegungen, Anwendungsszenarien und Gründe

Bei der Konstruktion ist die vermutlich erste entstehende Frage, welche Beobachtungen überhaupt mitgeteilt werden sollen und wie dies umgesetzt werden kann. Die Komplexität der Aufgabe unterscheidet sich beispielsweise wesentlich je nachdem, ob nur das Ergebnis der Simulation oder auch der Weg zum Ergebnis dargestellt werden soll.

Sofern nur das Ergebnis von Relevanz ist, können Daten beliebig später und unter Umständen entsprechend aggregiert bereitgestellt werden. Beispiele dafür sind der Map-Reduce Algorithmus oder auch Frameworks wie Hadoop[2] und Apache Spark[3], bei denen Berechnung verteilt ausgeführt werden und Ergebnisse zum Abschluss konsolidiert bereitgestellt werden. Damit ein Ergebnis so entsprechend bereitgestellt werden kann, muss die zugrunde liegende Aufgabe auch auf die Berechnung eines entsprechenden Ergebnisses ausgelegt und nicht in erster Linie kontinuierlicher Natur sein. Beispiele für eine Aufgabe kontinuierlicher Natur sind kooperative Onlinespiele, bei denen jeder Teilnehmer über das aktuelle Voranschreiten der Simulation informiert werden muss. Ein anderes Anwendungsszenario ist beispielsweise die verteilte Erkennung von Objekten oder Situationen durch Roboter, die jeweils eigene Sensoren aufweisen. Auch kompetitive Anwendungen sind denkbar, wie die Simulationen von Schlachtfeldern.

Wird nun der Kontext von Referenznetz(-simulationen) betrachtet, ist insbesondere für die Anwendungsfälle der grafischen Darstellung und der Reaktion auf Auslastungen die kontinuierliche Variante naheliegender. Aber auch darin können weitere Aufteilung erfolgen. Somit ist es denkbar, dass durch den Modellierer des Systems einzelne Teilergebnisse zu definierten Situationen mitgeteilt werden. Da hierbei der Modellierer aktiv Benachrichtigungen in das Modell integriert, soll diese Variante fortan als *aktiver Simulationsfeed* bezeichnet werden. Im Gegensatz dazu könnte eine entsprechen Implementation auch passiv alle Ereignisse aufnehmen und diese bei Bedarf widerspiegeln ohne Eingriff ins Modell selbst. Analog soll diese Art der Umsetzung als *passiver Simulationsfeed* bezeichnet werden.

[2] https://hadoop.apache.org/ - Zuletzt abgerufen am 17.12.2021
[3] https://spark.apache.org/ - Zuletzt abgerufen am 17.12.2021

Beide Varianten können eine sinnvolle Ergänzung aus Sicht der Plattform für das Gesamtsystem darstellen. Bei der Konstruktion kann auf entsprechende Literatur aus den oben referenzierten Anwendungsgebieten zurückgegriffen werden. Dennoch enthält die Abbildung auf Referenznetze sehr viele Variablen und viele offene Fragen. Die einzelnen Bestandteile des Simulationsfeeds werden hier in allgemeiner Form diskutiert. Da sich die Auswahl der einzelnen Alternativen je Aspekt jedoch maßgeblich anhand eines möglichen Anwendungsgebiet unterscheidet, entfällt eine konkrete Auswahl einzelner Alternativen für den Kontext dieser Arbeit.

Bei der genauen Betrachtung eines solchen Simulationsfeeds soll die Sicht auf den Lebenszyklus eines einzelnen Eintrags innerhalb des Feeds eingenommen werden. Einträge werden lokal in der Simulation erzeugt. Dafür muss zunächst geklärt werden, in welchen Situationen ein solcher Eintrag erzeugt wird. Darüber hinaus müssen beinhaltete Daten definiert werden und eine Form der Repräsentation der Daten bekannt sein bzw. gewählt werden. Dabei sollte aus den Daten auch hervorgehen, wie sich der Eintrag ablaufseitig zu anderen Einträgen positioniert. Sobald ein Eintrag erzeugt wurde, muss er an einer geeigneten Stelle gespeichert werden. Wenn Interesse an dem Eintrag besteht, muss er an einen Konsumenten des Simulationsfeeds ausgeliefert werden. Wird ein Eintrag nicht mehr benötigt, so kann er gelöscht werden. Dies sollte nach definierten Kriterien erfolgen.

All diese Fragen werden in den folgenden Abschnitten erörtert. Zusammengefasst orientiert sich die Diskussion der Fragestellung, ob und wie ein solcher Feed realisierbar ist, an den folgenden wesentlichen Fragestellungen:

- »Welche Anwendungsszenarien existieren für Simulationsfeeds?« (Dies wurde bereits in diesem Abschnitt adressiert)
- »Wo werden die Daten vorgehalten?«
- »Was beinhalten die Daten?«
- »Wie ist die Repräsentation der Daten?«
- »Wie lange werden Daten vorgehalten?«

All diese Fragestellungen unterscheiden sich nicht wesentlich je nachdem, ob ein aktiver oder passiver Simulationsfeed betrachtet wird. Passive Simulationsfeeds im Speziellen benötigen jedoch noch die Klärung einer weiteren Fragestellung:

- »Wann sollen Aufzeichnungen erfolgen?«

Das permanente Aufzeichnen aller Events kann bewirken, dass die Aufzeichnung aufwändiger als die eigentliche Berechnung wird. Insbesondere im Kontext der True-Concurrency-Semantik ist diese Frage nicht leicht zu beantworten.

11.6.2. Ort der Datenspeicherung

Ein Aspekt der Konstruktion eines Simulationsfeeds umfasst die Verortung der Bereitstellung der Daten. Der beabsichtigte Zweck des Simulationsfeeds liegt darin, einer externen Anwendung, wie beispielsweise einer grafischen Oberfläche, die Möglichkeit zu verschaffen, auf die Gesamtheit der verteilten Simulation zugreifen zu können. Folglich muss für dieses Unterfangen die Gesamtmenge der (persistierten) Simulationsevents integriert und bereitgestellt werden. An dieser Stelle gibt es mehrere Ansätze, welche im Folgenden gegeneinander abgewägt werden.

Lokales Vorhalten der Simulationsevents Bei diesem Ansatz wird durch jede Plattform selbst die Menge ihrer persistierten Simulationsevents in einem lokalen Speicher gespeichert. Durch ein Verzeichnis im Plattformmanagement kann eine externe Anwendung Informationen über die an der Simulation beteiligt Plattformen einholen. Sobald für die externe Anwendung geklärt ist, welche Plattformen an der verteilten Simulation beteiligt sind, kann diese die Simulationsevents bei den entsprechenden Plattformen abfragen. Dabei ist es dann entweder notwendig, dass alle Plattformen die Simulationsevents im selben Format speichern oder aber die externe Anwendung über Informationen zu jeder einzelnen Plattform verfügt, um das dort eingesetzte Format in ein globales Schema zu integrieren.

Als Vorteil dieser Methode ist in erster Linie die hervorragende Skalierbarkeit zu nennen. Diese ergibt sich daraus, dass keine zentrale belastete Komponente existiert, die bei der Hochskalierung des Systems überlastet werden könnte. Besonderes Augenmerk muss hierbei auf das Verzeichnis gelegt werden, da dieses zwar zentral ist, aber im Allgemeinen nur zu Beginn und einmalig bei der Anfrage der Simulationsevents involviert werden muss.

Ein weiterer Vorteil besteht darin, dass bei dieser Methode neben dem Verzeichnis keine zusätzliche Infrastruktur benötigt wird, um die Nachrichten zur Verfügung zu stellen.

Ein deutlicher Nachteil dieses Verfahrens ist jedoch darin zu sehen, dass unter Umständen ein erheblicher Integrationsaufwand aufseiten der externen Anwendung geschehen muss. Da dieser Integrationsaufwand bei jeder Anwendung anfällt, welche die Daten konsumiert, muss auch in jeder einzelnen externen Anwendung diese Anforderung aufs Neue umgesetzt werden. Somit entsteht unnötige Coderedundanz, sofern nicht für derartige Anwendungen eine Bibliothek konzipiert wird.

Verwenden eines zentralen Nachrichtenbrokers Entgegen dem Ansatz, dass jede Plattform ausschließlich ihren eigenen lokalen Simulationsfeed zur Verfügung stellt, ist es ebenfalls eine Option, auf einen zentralisierten Nach-

richtenbroker zurückzugreifen. Das Plattformmanagement muss einen solchen Nachrichtenbroker ohnehin bereits für komplexe Agentenkommunikation zur Verfügung stellen, wie es in Kapitel 5 beschrieben wurde.

Durch diese Lösung entsteht eine zentrale Anlaufstelle für alle externen Anwendungen, welche sich für den Ablauf der Simulation interessieren. Eine Art Verzeichnis bleibt für die Plattformen der Simulation weiterhin notwendig, jedoch sind externe Anwendungen nun nicht mehr darauf angewiesen, auf das interne Verzeichnis zugreifen zu können. Dies bietet insbesondere auch aus sicherheitstechnischen Blickpunkten Vorteile, da somit keiner externen Anwendung Zugriff auf die innere Struktur des Plattformmanagements gewährt werden muss.

Eins der größten Probleme besteht bei dieser Lösung jedoch darin, dass jedes einzelne Simulationsevent an den zentralen Nachrichtenbroker übertragen werden muss. Bei der lokalen Variante mussten diese lediglich in den ungleich schnelleren Hauptspeicher oder lokalen persistenten Speicher übertragen werden. Zusätzlich entsteht durch den zentralisierten Nachrichtenbroker ein potentieller Flaschenhals, da sowohl alle Simulationskomponenten ständig als auch alle externen Anwendungen auf diesen zugreifen.

Kombinierte Lösungen Häufig liegt die Lösung nicht in einem der Extremfälle, daher ist es naheliegend, auch bei der Form der Bereitstellung eine hybride Variante zu betrachten. Ein Hauptproblem bei der Verwendung eines Nachrichtenbrokers liegt wie zuvor beschrieben darin, dass Simulationsevents stets an diesen übertragen werden müssen. Da eine Netzwerkkommunikation immer deutlich langsamer als eine Kommunikation mit dem Hauptspeicher ausfällt, sollte diese Form der Kommunikation nach Möglichkeit gering gehalten werden.

Die Überlegung, zunächst eine gewisse Menge an lokalen Simulationsdaten zu sammeln und diese dann in ihrer Gesamtheit an den Nachrichtenbroker zu übertragen, liegt nahe. Dabei ist die Frage, zu welchen Situationen eine Übertragung am günstigsten ist. Sinnvolle Gelegenheiten könnten durch Zeitpunkte, Anzahl an Simulationsevents oder aus formaler Sichtweise an das Feuern synchroner Kanäle gekoppelt werden. Das Feuern *verteilter* synchroner Kanäle bildet ohnehin in der Regel eine komplexe Kommunikation ab und hat daher gute Chancen, ohnehin den Nachrichtenbroker zu involvieren.

11.6.3. Events und ihre Inhalte

Eine weitere zu diskutierende Frage umfasst, welche Art von Daten gesichert werden sollen. Dabei spielt nicht das Format eine zentrale Rolle, sondern vielmehr

die Komplexität und die gespeicherten Eigenschaften von Simulationsevents. Zunächst muss eine Entscheidung zwischen dem Speichern von aktiven oder passiven Elementen des Netzes erfolgen. Ein Speichern passiver Elemente umfasst die Aufnahme eines punktuellen Zustandes, bestehend aus einer Markierung des Netzsinstanzsystems. Eine Speicherung aktiver Inhalte hingegen würde einer Event-basierten Speicherung entsprechen und sich effektiv nach dem Event-Sourcing Entwicklungsmuster richten, wie es in Abschnitt 2.6.1 vorgestellt wurde. Darüber hinaus stellt sich die Frage, wie und ob Informationen beim Speichern ausgelassen werden können. Ebenso ist die Abgrenzung des Speicherns passiver Inhalte zum Statusmonitor fragwürdig, da beide ähnliche Konzepte realisieren würden. Aus den Kombinationen dieser Überlegungen entstehen nun vier verschiedene Ansätze. Die jeweiligen Vor- und Nachteile der vier Ansätze werden im Folgenden diskutiert.

Speichern von Vereinfachungen der Markierung des Netzsystems Die Grundidee bei diesem Ansatz besteht darin, lediglich einfache Version der Marken zu speichern und keine komplexen Objekte. Es handelt sich daher um die datensparsame Variante der Persistierung von Zuständen. Das Vereinfachen der Marken hat zur Folge, dass die Simulation des Netzes von außen betrachtet der Simulation einer Petrinetzklasse entspricht, welche niedriger angesiedelt ist als Referenznetze.

Durch die Vereinfachung der Marken reduziert sich auch deutlich der Speicheraufwand und der Analyseaufwand, welche Daten konkret gespeichert werden müssen für die jeweilige Marke.

Offensichtlich ist dabei jedoch auch, dass durch vereinfachte Marken die Simulation weniger aussagekräftig ist als in ihrem Originalzustand. Diese Form der Speicherung kann jedoch genutzt werden, um den Kontrollfluss im Netzinstanzsystem darzustellen und zumindest zum Teil Eigenschaften wie beispielsweise eine Verklemmung zu identifizieren.

Speichern von vollen Kopien der Markierung Dieser Ansatz ist die speicherintensive Variante der Zustandssicherung. Nach jedem Schalten einer Transition wird ein komplettes Abbild aller Marken in allen Netzinstanzen persistiert. Dieser Ansatz liefert offensichtlich den größten Nutzen, da sämtliche Daten verfügbar sind.

Ein großer Vorteil ergibt sich daraus, dass ein konsumierendes Interface aus jedem gesicherten Zustand das System sofort und ohne weitere Berechnung darstellen kann. Leider lastet dem Ansatz der genau so offensichtliche gewaltige Aufwand der Speicherung an. Darüber hinaus würden viele Daten redundant gespeichert, da die Annahme nahe liegt, dass in vielen Fällen ein Großteil der Marken durch das Feuern einer einzelnen Transition nicht verändert wird.

Ein weiteres Problem ergibt sich dadurch, dass beim Feuern einer Transition eine große Menge an Daten gespeichert werden muss. Da Transitionen im Allgemeinen nebenläufig feuern können, kann es dazu führen, dass das System nur durch das Abspeichern des aktuellen Zustands überlastet wird. Ein weiteres Problem ergibt sich beim Einsatz von True-Concurrency-Semantik, da hierbei das Netzinstanzsystem nach dem Feuern einzelner Transitionen nicht in einem definierten Zustand sein muss, da andere Transitionen nebenläufig und unvergleichbar in Bezug auf ihre Reihenfolge schalten können. Abhilfe kann hier der Einsatz von Checkpoints schaffen, welche jedoch aber auch das System ausbremsen. Checkpoints werden insbesondere im Kontext von True-Concurrency noch einmal detaillierter in Abschnitt 11.6.6 diskutiert.

Speichern von generierten und konsumierten Marken Eine Erweiterung des zuvor genannten Modells der vollen Speicherung entsteht dadurch lediglich die Differenz des durch eine Transitionsschaltung erzeugten Zustandes in Relation zum vorherigen Zustand zu sichern. Dabei entfällt die Notwendigkeit, das gesamte System im gegenwärtigen Zustand zu sichern. Dabei entsteht der Effekt, dass der Systemzustand nun nicht weiter durch eine einzelne Aufzeichnung des Zustands nachvollzogen werden kann. Da eine inkrementelle Speicherung vorliegt, muss stets ebenfalls mit gelegentlich gesicherten vollen Abbildern gearbeitet werden, von denen der aktuelle Zustand aus rekonstruiert werden kann. Folglich kann diese Form des Feeds nur sinnvoll eingesetzt werden, wenn sie zumindest grundlegend mit der vorhergegangenen Form kombiniert wird. Dieser Ansatz entspricht der Speicherung von aktiven Elementen mit vollem Datenbestand.

Eine weitere potenzielle Verbesserung kann durch die Unterscheidung vom Beginn eines Transitionsschaltens und dem Abschluss des Schaltvorgangs umgesetzt werden. Dies ist insbesondere bei lang schaltenden Transitionen sinnvoll, um diese Begebenheit zu analysieren oder beispielsweise in einer externen grafischen Oberfläche darzustellen. Der Speicherbedarf würde nur unwesentlich steigen, da auch beim Speichern des kombinierten Events sowohl konsumierte als auch produzierte Marken gesichert werden müssen. Beginne von Schaltungen sichern Informationen über die konsumierten Marken, Ende von Schaltungen die der produzierten Marken. Als Overhead würde also die erneute Sicherung der zugehörigen Zeitstempel bleiben.

Speichern von feuernden Transitionen allein Dieser Ansatz entspricht dem datensparsamen Speichern der Aktionen aktiver Elemente. Hierbei müsste ein Konsument der Daten im Zweifel die Feuervorgänge der Transitionen selbst ausführen, um Informationen über die Markierung des Netzes erlangen zu können. Da insbesondere bei Referenznetzen komplexe Anweisungen Teil der Transitionsanschriften sein können, ist dadurch die Ausprägung der Fol-

gemarkierung keineswegs trivial zu erschließen. Daher müsste der Feuervorgang lokal beim Konsumenten erneut berechnet werden. Somit ginge somit der Vorteil der entfernten und skalierbaren Ausführung gänzlich verloren.

Komplett hinfällig ist dieser Ansatz jedoch nicht, da dennoch aggregierte Informationen über das Verhalten des Systems gesammelt bzw. dargestellt werden können. Diese aggregierten Informationen können beispielsweise genutzt werden, um stark genutzt Transitionen zu identifizieren. Hilfreich ist diese Information beispielsweise beim Monitoring oder für Entscheidungen, an welchen Stellen eine Verteilung stattfinden soll und an welchen ein lokales Schalten günstiger wäre. Diese Informationen können das Plattformmanagement außerdem befähigen, gezielter Plattformen hinzuzuschalten oder abzubauen.

Zusammengefasst erscheint die Speicherung aktiver Elemente mit vollem Dateneinsatz am vielversprechendsten, da diese einen guten Mittelweg zwischen Datenmenge, Aufwand und Simulationseinschränkung beinhaltet. Dies ist zumindest für allgemeine Fälle zu empfehlen. Ist die Natur des Datenkonsumenten weiter spezifiziert, kann der Einsatz eines anderen Ansatzes lohnenswert sein, wie beispielsweise beim skizzierten Hinzuschalten von Plattformen durch das Plattformmanagement.

11.6.4. Datenrepräsentation

Alle diskutierten Ansätze erfordern, dass der Simulationsfeed in einer geeigneten Form repräsentiert und zum Abruf bereitgestellt wird. Als Empfehlung ging aus den vergangenen Abschnitten die Aufzeichnung von Transitionsfeuerevents hervor. Darüber hinaus benötigen alle Ansätze bis auf dem alleinigen Speichern der Transitionsschaltungen zusätzlich Datenobjekte zu jedem publizierten Event. Es existieren verschiedene Ansätze, die nachfolgend in Bezug auf ihre Benutzbarkeit, Kompatibilität zu konsumierenden Systemen und ihre Umsetzbarkeit untersucht werden.

Dabei wird zum einen eine abstrakte Sicht auf die Repräsentation eingenommen und diskutiert, welche Form der Strukturierung Anwendung finden sollte. Zusätzlich wird die niedrigere technische Sicht auf die Repräsentation diskutiert.

Abstrakte Sicht

Zunächst soll die generelle Datenstruktur, in der die einzelnen Events des Simulationsfeeds gespeichert werden, diskutiert werden. Im Normalfall würde hier eine einfache Aufreihung von Logeinträgen genügen. Da bei der Simulation jedoch

Schaltfolgen in Petrinetzen abgebildet werden, welche inhärente Nebenläufigkeit erlauben, ist eine Abbildung mit sequenziellen Logs mitunter schwierig. Dies gilt insbesondere unter Anwendung der True-Concurrency-Semantik, bei der einzelne Events bezogen auf ihre Abfolge mitunter nicht vergleichbar sind. Eine mögliche Lösung stellen Kausalnetze da, welche keine Verzweigung an Plätzen erlauben.

Einfache Logs Der Vorteil bei der Benutzung einfacher Logs liegt klar in der einfachen Konstruktionsweise. Für jedes Event genügt eine Zeichenkette mit einer geeigneten Speicherung eines dazugehörigen Datenobjektes. Diese Möglichkeit ist in Renew auch bereits durch das Logging Plugin gegeben.

Ein Nachteil dieser Methode besteht in der zwangsläufigen Sequentialisierung der Schaltfolge. Ohne weitere Informationen könnte beispielsweise eine grafische Oberfläche, welche auf dem Simulationsfeed aufbaut, keine entsprechende Nebenläufigkeit darstellen. Je nach Geschwindigkeit der Darstellung ist dieser Nachteil für den Betrachter jedoch unter Umständen verschmerzbar. Bei sehr großen Simulationen kann es auf diese Weise allerdings zu Problemen kommen.

Auch zunächst unklar ist, wie die Aggregation von mehreren nebenläufigen Logs von verschiedenen Plattformen geschehen sollen. Eine mögliche Lösung umfasst die Unterstützung mehrerer sequenzieller Logs, welche von Grund auf als nebenläufig angesehen werden. Besonderes Augenmerk muss hierbei jedoch dann auf die Schaltung verteilter synchrone Kanäle gelegt werden, da diese die separaten sequenziellen Logs miteinander verknüpfen. Genau auf dieser Basis wäre es aber auch denkbar, mithilfe verteilter synchroner Kanäle als Referenzpunkte ein entsprechendes sequenzielles globales Log zu erzeugen. Dieser könnte dann wie beschrieben ausgelesen werden.

Kausalnetze Eine gute Lösung für die bei einfachen Logs beschriebene Problematik umfasst der Einsatz von Kausalnetzen. Da Kausalnetze keine Verzweigung an Plätzen erlauben, können sie lediglich in einer Art und Weise geschaltet werden. Dennoch erlauben sie die Modellierung nebenläufiger und auch bezüglich der Abfolge unvergleichbarer Operationen.

Während ihre Konstruktion natürlich auch entlang eines sequenziellen Logs geschehen kann, spielen Sie Ihre Vorteile erst bei der tatsächlichen Ausnutzung von Nebenläufigkeit aus. Die zugehörige Konstruktion ist in diesem Fall deutlich aufwendiger und bedarf der Bewertung der Unabhängigkeit zweier oder mehrerer Schaltvorgänge im Netzinstanzsystem voneinander. Insbesondere unter der Verwendung von True-Concurrency-Semantik kann diese ständige Bewertung mitunter merkliche negative Effekte auf die Gesamtperformance des Systems haben und schwer zu realisieren sein. Durch synchrone Kanäle und die hierarchischen Strukturen von Referenznetzen sind Beziehungen zueinander nicht immer trivial zu erschließen.

> In der Arbeit (Krön, 2021) wurde die dynamische Erzeugung von Kausalnetzen aus schaltenden Netzen heraus thematisiert. Dabei wurde die Semantik jedoch auf die Interleaving-Semantik eingeschränkt, um die Aufgabe für den Rahmen der Arbeit bewältigbar zu halten. Dennoch gibt der generelle Ansatz einen guten Gedankenanstoß, wie ein derartiges Log realisierbar sein könnte.

Zusammengefasst bietet der Ansatz, ein Log aus Kausalnetzen zu generieren, eine wesentlich bessere Verwendbarkeit, da auch Nebenläufigkeit entsprechend abgebildet werden kann. Dennoch ist die Erzeugung noch nicht abschließend geklärt und bearbeitet und würde umfangreichere Nachforschungen erfordern. Daher sollte für den Moment noch ein einfaches Log eingesetzt werden, bis in zukünftigen Arbeiten die dynamische Erzeugung von Kausalnetzen weiter erforscht wurde.

Technische Sicht

Bei der Verwendung klassischer Logs sollten andere Formen der Zuordnung der Abfolge der Logs zueinander zum Einsatz kommen. Eine Variante wäre der Einsatz entsprechender Zeitstempel. Für dieses Vorgehen müssen allerdings zumindest grundlegend synchronisierte Uhren auf den Knoten vorliegen. Wie jedoch in einschlägiger Literatur nachzulesen ist, ist das Problem der synchronisierte Uhren auf verschiedenen Rechnerknoten keineswegs trivial. Für realweltliche Umgebungen existieren jedoch für unseren Anwendungsfall ausreichend präzise Lösung, wie der Berkeley Algorithmus (Gusella und Zatti, 1989), der Algorithmus von Cristian (Cristian, 1989) oder das »Network Time Protocol (NTP)« (Mills, 1985). Aktuellere Arbeiten behandeln beispielsweise Zeitsynchronisation im Kontext von möglichem Fehlverhalten einzelner beteiligter Knoten (Lenzen und Rybicki, 2019). Die grundlegenden Prinzipien unterscheiden sich jedoch nicht ausreichend für eine detailliertere Diskussion an dieser Stelle.

Ein anderer Ansatz wäre die Verwendung logischer bzw. vektorieller Zeitstempel nach Lamport (Lamport, 1978). Dabei wird lediglich die Anzahl der Events (Transitionsschaltungen) gezählt, dabei jedoch nach dem jeweiligen Prozessor, welcher das Event ausführt, unterschieden. Somit ergibt sich ein Vektor, aus welchem die partielle Ordnung bezüglich der Reihenfolge der Schaltvorgänge inklusive Nebenläufigkeit ableitbar ist. Da im Fall von Plattform und Plattformmanagement allerdings ein verteiltes (nebenläufiges) System vorliegt, welches wiederum lokal ebenfalls nebenläufig agiert, ist eine Gruppierung innerhalb des Vektors nach Plattformen sinnvoll. Zu diesem Zweck muss eine Plattform allerdings einen eindeutigen Identifikator aufweisen. Dies stellt im Allgemeinen kein Problem dar und wird in Abschnitt 12.7.3 noch einmal detaillierter aufgegriffen.

Unabhängig von der Sortierung der Events müssen die Daten in der einen oder anderen Form persistiert werden. Direkte Möglichkeiten umfassen die Speicherung als binär serialisierte Daten oder als textbasierte Daten. Die jeweiligen Vor- und Nachteile sollen im Rahmen der Anwendung kurz diskutiert werden:

Binär-serialisiert Beim binär-serialisierten Modell werden alle Objekte in einer Binärrepräsentation persistiert. Dadurch wird eine effiziente Speicherung möglich, welche durch eine Realisierung mit Zeichenketten nicht erreicht werden kann. Dazu ist es jedoch notwendig, dass das Binärformat von einem entsprechenden Parser wieder deserialisiert werden kann. Zur Realisierung des binär serialisierten Modells, stehen eine Vielzahl an Binärrepräsentation zur Verfügung. Eine Möglichkeit bietet die Serialisierung durch die Java Virtual Machine (JVM), da diese bereits in der Java Sprache integriert ist, auf der Renew aufbaut. Zu beachten ist, dass diese spezialisierten Parser bei Konsumenten des Simulationsfeeds vorliegen müssen und diese ebenfalls für Fehlersuche nicht leicht durch menschliche Betrachter ausgelesen werden können.

Serialisiert in Zeichenketten Ein weiterer Ansatz, um die Daten zu speichern, ist diese Serialisierung in Zeichenketten. Der Hauptvorteil bei der Serialisierung durch Zeichenketten besteht darin, dass diese durch Menschen lesbar sind. Durch diesen Umstand können auf einfache Art und Weise zusätzliche und neue Parser für eine Serialisierung in Zeichenketten entworfen werden. Wird auf ein standardisiertes Datenmodell zurückgegriffen, so genügt dabei die Abbildung des Datenmodells. Da eine mögliche Anwendung ein externes (ggf. webbasiertes) Simulationsinterface umfasst, ist dieser Ansatz reizvoll da er mit dem Kommunikationsprotokoll HTTP synergiert, bei dem häufig textbasierte Serialisierungen zum Einsatz kommen. Der Nachteil bei der textbasierten Darstellung ist klar der höhere Speicherplatzbedarf, welcher sich negativ auf Übertragungsperformance und Datenvorhaltung auswirkt.

Vereinfacht durch einzelne Zeichenketten Der letzte hier diskutierte Ansatz ist die Vereinfachung durch einzelne Zeichenketten. Dabei werden keine komplexen Objekte gespeichert sondern lediglich eine einfache Stringrepräsentation der Objekte. Dadurch gehen einige Information verloren und Objekte lassen sich nicht in ihrem vollen Detailreichtum darstellen, jedoch wird durch diesen Ansatz auch die Menge an Daten, die zu speichern sind, im Normalfall drastisch reduziert. Insgesamt ist dieser Ansatz aber kaum gewinnbringender als der Ansatz Events generell nur die feuernde Transition beinhalten zu lassen. Diese Variante ist gegenwärtig im Renew Logging plugin implementiert.

Auch bei der Wahl der technischen Repräsentation der einzelnen Events des Simulationsfeeds beruht die Auswahl stark auf dem intendierten Einsatzzweck. Soll

beispielsweise eine einfache Web-basierte grafische Oberfläche angebunden werden, bietet sich eine textbasierte Serialisierung an. Diese könnte jedoch bei größeren Simulationen basierend auf dem Datenaufkommen zu Ineffizienzen führen, sodass ein browserbasiertes Parsing binärer Daten den Aufwand gegebenenfalls rechtfertigt. Bei der internen Anwendung entlang dem Holonmodell kann auch auf interne Serialisierung und Deserialisierung zurückgegriffen werden, sodass die effizienteren Binärdaten verwendet werden können.

Der Einsatz von Realzeitstempeln und logischen Zeitstempeln unterscheidet sich vor allem in der Komplexität der Umsetzung. Zeitsynchronisation und Aufzeichnung von Zeitstempeln sind außerordentlich gut erforschte Bereiche und entsprechende Lösung stehen bereit. Beim Einsatz logischer Zeitstempel muss noch eine Unterscheidung nach nebenläufigem lokalen Prozess erfolgen und eine Gesamtintegration implementiert werden. Zusätzlich muss entsprechendes Wissen bei der Implementation eines Konsumenten vorhanden sein, um die logischen Zeitstempel korrekt zu parsen. Realzeitstempel gewähren einen deutlich intuitiven Zugang zur Implementation, können aber keine hundertprozentige Genauigkeit garantieren, wie es bei logischen Zeitstempeln möglich ist.

11.6.5. Datenbereinigung

Ein weiterer zentraler Punkt, den es zu beachten gilt, ist die Art und Weise wie Daten vorgehalten werden bzw. welche Mengen von Daten vorgehalten werden sollen. Da insbesondere bei großen Simulationen eine Vielzahl an Transitionsfeuervorgängen auftreten, entsteht dabei entsprechend auch eine große Menge an Simulationsevents, die persistiert werden müssen. Falls eine solche Simulation nun für längere Zeit läuft, kommt es früher oder später zwangsläufig dazu, dass die Menge an Daten den Speicher übersteigt. Folglich ist es notwendig verschiedene Strategien der Datenvorhaltung zu betrachten.

Einfaches Speichern ohne Löschung Der einfachste Umgang mit den Datenmengen ist das simple Speichern sämtlicher Daten, die bei der Protokollierung der Simulation anfallen. Die Vorteile sind dabei offensichtlich ein geringer Implementationsaufwand, sowie die vollständige Verfügbarkeit sämtlicher Daten der Simulation. Da jedoch jeder reale Speicher endlich ist, läuft dieses Modell unweigerlich früher oder später in ein Problem. In dem Moment, in dem der lokale Speicher zur Speicherung neuer Events nicht mehr ausreicht, würde das Programm entweder Daten löschen müssen oder mutmaßlich abstürzen. Da allerdings in modernen Rechnern sowohl die Hauptspeicher als auch die persistente Speicher sehr umfangreich ausgeführt sind, kann dieser Ansatz für einen ersten Prototypen durchaus in Betracht gezogen werden. Dabei sollte jedoch klar sein, dass bei einer größeren Simulation mit Problemen zu rechnen ist.

Löschen nach Realzeitstempel Eine weitere naheliegende Herangehensweise ist es die Simulationsevents nur für eine bestimmte Zeitspanne vorzuhalten. Dabei würden automatisiert sämtliche älteren Events gelöscht, um Platz für weitere Daten zu schaffen. Während dieser Ansatz bei Simulationen, die innerhalb einer gegebenen Zeitspanne mit einer ungefähr gleich bleibenden Rate Transitionsschaltungen ausführen, relativ verlässliche Ergebnisse erzielen kann, ist auch eine Simulation denkbar, bei der in sehr unterschiedlichem Umfang zu verschiedenen Zeitpunkten Transitionsschaltungen auftreten. Dies kann insbesondere bei Netzen, die externe Komponenten ansprechen, um Berechnung durchzuführen, auftreten. Als Vorteil ist jedoch zu nennen, dass der Implementationsaufwand lediglich das Abgleichen der aktuellen Zeit mit der gespeicherten Zeit des entsprechenden Events umfasst. Ein weiterer Nachteil besteht darin, dass bei entsprechend ungünstigem Aufkommen von Events trotz der Bereinigungsstrategie Speicher überlaufen können.

Löschen nach Anzahl Eine weitere einfache Methode um die Menge der persistierten Simulationsevents auf einem beherrschbaren Niveau zu erhalten, besteht darin nur eine feste Anzahl an Events vorzuhalten. Das dabei verwendete Modell entspricht am ehesten einer Queue, aus der erst dann Elemente entfernt werden, wenn das obere Limit erreicht wurde. Leider weist auch dieses Modell Probleme mit den Mengen der Transitionsschaltungen auf. Dies begründet sich darin, dass unter Umständen ein Netz mit wenig Transitionsschaltungen auch noch sehr alte Events vorhält, während ein Netz welches höhere Raten an Transitionsschaltungen aufweist, nur eine zeitlich gesehen sehr kurze Historie aufweist. Dies kann insbesondere dann problematisch werden, wenn viele Simulationsevents von verschiedenen Knoten aggregiert dargestellt werden sollen. Wenn nun auf diesen Knoten jeweils deutlich unterschiedliche Zeitspannen durch die Vorhaltung von persistierten Simulationsevents abgedeckt sind, kann eine Darstellung der globalen Gesamtsimulation deutlich erschwert sein. Auch bei diesem Modell ist die Implementation sehr einfach, da nur auf die Anzahl der Einträge in der verwendeten Datenstruktur Rücksicht genommen werden muss.

Hybride Lösungen Da alle vorgestellten einfachen Ansätze im allgemeinen Fall nicht unmittelbar zu wünschenswerten Ergebnissen führen, liegt die Überlegung nahe, ein hybrides Modell zu schaffen, welches die Vorteile der jeweiligen Ansätze vereint. Somit könnte beispielsweise die Rate der Transitionsschaltungen aus der jüngeren Vergangenheit über einen lokalen Simulationsfeed abgelesen werden und mittels eines geeigneten Protokolls zwischen den beteiligten Rechnerknoten eine sinnvolle Menge der vorzuhaltenden Simulationsdaten ausgehandelt werden. Dabei können verschiedene Faktoren Beachtung finden, wie beispielsweise die eben beschriebene Rate der Tran-

sitionsschaltungen, dem verfügbaren Hauptspeicher und der gewünschten Länge der Rekonstruierbarkeit der Simulation.

Alle hier diskutierten Lösungen beinhalten anwendungsspezifische Vor- und Nachteile. Die umfangreichste, aber vermutlich beste Realisierung wäre die Implementation aller Lösungen mit dem Angebot einer Konfiguration je nach Anwendungsfall. Kleine Anwendungsfälle können die Löschung während des Lebenszyklus der Simulation auslassen. Systeme, bei den harte Anforderungen an den rekonstruierbaren Zeitraum der Simulation bestehen, setzen auf die Löschung nach Realzeitstempel. Für viele Anwendungen wird die Löschung nach Anzahl eine gute Methode darstellen, da sie garantieren kann, dass Speicher nicht überlaufen. Für komplexe Anwendungen mit ebenso komplexen Anforderungen sind hybride und einstellbare Lösungen denkbar, welche für die jeweiligen Teillösungen den optimalen Ansatz wählen.

11.6.6. Passiver Feed: Auswahl der aufzuzeichnenden Events

Der letzte zu betrachtende Aspekt bezieht sich ausschließlich auf den passiven Simulationsfeed. Sofern keine Informationen seitens des Modellierers bezüglich der Zeitpunkte, wann und welche Events aufgezeichnet werden sollen bereitstehen, müssen diese nach anderen Gesichtspunkten bestimmt werden. Neben der reinen Bestimmung der Situation, zu den Events aufgezeichnet werden sollen, ist es lohnenswert zu betrachten, wann diese Events (sofern vorgesehen) an eine Sammelstelle wie einen Nachrichtenbroker versandt werden sollen. Darüber hinaus ist die Komplexität der Aufzeichnung deutlich abhängig von der Wahl der Semantik, nach der das Netzmodell simuliert wird.

Bei der Frage nach den aufzuzeichnenden Situationen lautet die intuitive Lösung, dass alle Transitionsfeuervorgänge aufgezeichnet werden müssen, damit ein umfassendes Bild der Simulation hergestellt werden kann. Tatsächlich muss die Aufzeichnung allerdings nur so genau sein, dass sie die Bedürfnisse der Konsumenten des Simulationsfeeds zufriedenstellen können. Im Kontext des bereits genannten Beispiels der externen graphischen Oberfläche werden alle Feuervorgänge benötigt, da jeder Einzelne im Zweifel dem Nutzer angezeigt werden soll. Bei der Variante der Auskunft eines Agenten über die Arbeit seiner Subagenten sind gegebenenfalls Meilensteine in der Berechnung bzw. Teillösungen ausreichend. Die Intensität der Protokollierung sollte daher für den jeweiligen Anwendungsfall konfigurierbar sein.

Insbesondere dann, wenn nicht jeder einzelne Vorgang benötigt wird, sind weitere Vereinfachungen denkbar. Existieren beispielsweise Netzstrukturen innerhalb des Netzes, welche bei gleicher Startmarkierung stets die gleichen Seiteneffekte erzeugen und die jeweils identische Endmarkierung aufweisen, kann auch bei lokal

nichtdeterministischem Schaltverhalten auf die genaue Schaltfolge verzichtet werden. Ferner könnte genauso mit entsprechenden Invarianten verfahren werden, bei denen beispielsweise solange der Teil des Netzes aktiv ist, lediglich einmalig die Daten der Invariante übermittelt werden. Weitere Gedankenanstöße in diese Richtung können aus der Literatur zu Model Checking-Verfahren entnommen werden, wie beispielsweise zu dem Tool LoLA (Wolf, 2018). Vereinfachungen von Netzstrukturen werden dort u.A. eingesetzt, um effizienter Erreichbarkeitsgraphen zu berechnen.

Beim Einsatz externer Speichermedien wie beispielsweise einem Nachrichtenbroker kann eine Zwischenspeicherung eines lokalen Simulationsfeeds sinnvoll sein, damit das Netzwerk nicht mit einzelnen Simulationsnachrichten überlastet wird. Damit jedoch regelmäßig Updates herausgeschrieben werden, kann es sinnvoll sein, nach entweder einer gewissen Anzahl an Einträgen oder nach einer festen Zeitspanne Updates zu senden. In jedem Fall sollte aber ein Update geschickt werden, sobald ein verteilter synchroner Kanal gefeuert wird. An dieser Stelle verlässt die Aufzeichnung die Domäne der lokalen Plattform und für alle Betrachtungen innerhalb dieser Arbeit wird das Feuern eines verteilten synchronen Kanals als finales Ereignis aufgefasst. Die Rücknahme eines solchen Feuerns ist zwar theoretisch denkbar, aber äußerst kompliziert. Diese Art »verteiltes Undo« ist als Gegenstand einer vollwertigen eigenen Arbeit denkbar.

Checkpoints

Unabhängig von der Identifikation der Situation, in denen ein passiver Simulationsfeed Daten persistierten sollte, lohnt ein Blick auf die verwendete Semantik der Simulation. Auch, wenn nach der Grundidee des Event-Sourcings verfahren wird, muss hin und wieder ein Zwischenzustand gesichert werden, von dem aus der aktuelle Verlauf berechnet werden kann. Um einen solchen Zwischenzustand zu sichern, ist es notwendig, einen Zeitpunkt oder ein Verfahren zu definieren, wann oder wie dieser aufgezeichnet werden kann. Werden Interleaving- oder Step-Semantik eingesetzt, so ist die Bestimmung, wann aufgezeichnet werden kann, trivial. Eine solche Situation ergibt sich stets, nachdem die aktuelle Transition (Interleaving) oder aktuelle Transitionsgruppe (Step) gefeuert wurde. Entsprechende Überlegungen und Implementationen finden sich beispielsweise in der Arbeit (Jacob, 2001).

Beim Einsatz von True-Concurrency-Semantik gestaltet sich dieses Problem als komplizierter. Wie bereits beschrieben, ergibt sich hierbei zu keiner Situation zwangsläufig ein Zustand der Markierung mit keinen feuernden Transitionen. Die Problematik ist ähnlich gelagert wie bei der Verarbeitung der Persistenz nebenläufiger Transaktionen in Datenbanksystemen. Die üblichen Lösungsvorschläge

liegen hierbei im Einsatz von Checkpoints. Ein Checkpoint entspricht hierbei einem P-Schnitt durch den Verlauf der Simulation dargestellt als Kausalnetz.

Der Einsatz von Checkpoints erfordert stets ein Abwägen zwischen der Stärke des Ausbremsens des Systems und der Einfachheit der Wiederherstellung eines bestimmten Zustandes. Die Extremvariante würde einen kompletten Stopp aller Transaktionen bzw. im Fall des Simulationsfeeds einen kompletten Stopp aller Feuervorgänge bedeuten. Nach einer Zeitspanne, die dem noch am längsten weiterlaufenden aktuell gestarteten Feuervorgang entspricht, kommt die Simulation zum erliegen und ein eindeutiger Zustand kann gesichert werden. Offensichtlich führt dieses Verhalten jedoch auch zu den größten Einschränkungen der Simulation. Diese Variante ist so auch bereits in Renew implementiert in Form des Speicherns des Simulationszustandes.

Eine elegantere Lösung könnte darin liegen, die Eigenschaft von Petrinetzen auszunutzen, dass nur aktivierte Transitionen auch feuern können. Ein Simulator muss zwangsläufig zunächst die Aktiviertheit einer Transition prüfen bzw. feststellen, bevor diese wirklich gefeuert werden kann. Folglich besteht vor jeder schreibenden Operation ebenfalls eine lesende Operation.

Die antizipierte Umsetzung beinhaltet die Einführung eines Checkpoint Managers und einer Feuerliste der aktuell feuernden Transitionen. Die Grundidee beruht darauf, dass beim Auftrag einen Checkpoint zu erzeugen kurzzeitig keine *neuen* Feuervorgänge starten können. Alle nicht in aktive Feuervorgänge involvierten Marken werden aufgezeichnet und danach die Feuervorgänge wieder freigegeben. Die Feuervorgänge, welche zu Beginn der Aufzeichnung aktiv waren, lassen ihre *produzierten* Marken nach Abschluss ebenfalls aufzeichnen. Nachdem der letzte aktive Feuervorgang beendet wurde, ist die Aufzeichnung des Checkpoints abgeschlossen. Der aufgezeichnete Zustand wurde vom System niemals tatsächlich eingenommen. Er wäre aber erreicht worden, wenn zur gleichen Situation die Simulation gestoppt worden wäre. Aus dem Zustand sind daher Transitionsfolgen möglich, wie sie im realen System möglich wären nach einem vollständigen Stopp.

Ein Beispiel für einen detaillierten Ablauf für ein verschachteltes Erzeugen eines Checkpoints in True-Concurrency-Semantik ist in Abbildung 11.1 aufgeführt.

Der Übersichtlichkeit halber ist das Ablegen und Entnehmen von Marken aus den Plätzen nicht mit entsprechenden Pfeilen zu den Plätzen im Diagramm dargestellt. Des Weiteren sind jederzeit Bindungssuchen möglich und ggf. auch notwendig. Auch dieses Detail wurde ausgelassen, um das Diagramm nicht zu überfrachten.

Um die zum Zeitpunkt der Aufzeichnung feuernden Transitionen auslesen zu können, müssen alle Transitionen vor einem Feuervorgang eine Anmeldung mit den Marken, die konsumiert werden sollen, in einer entsprechenden Liste durchführen.

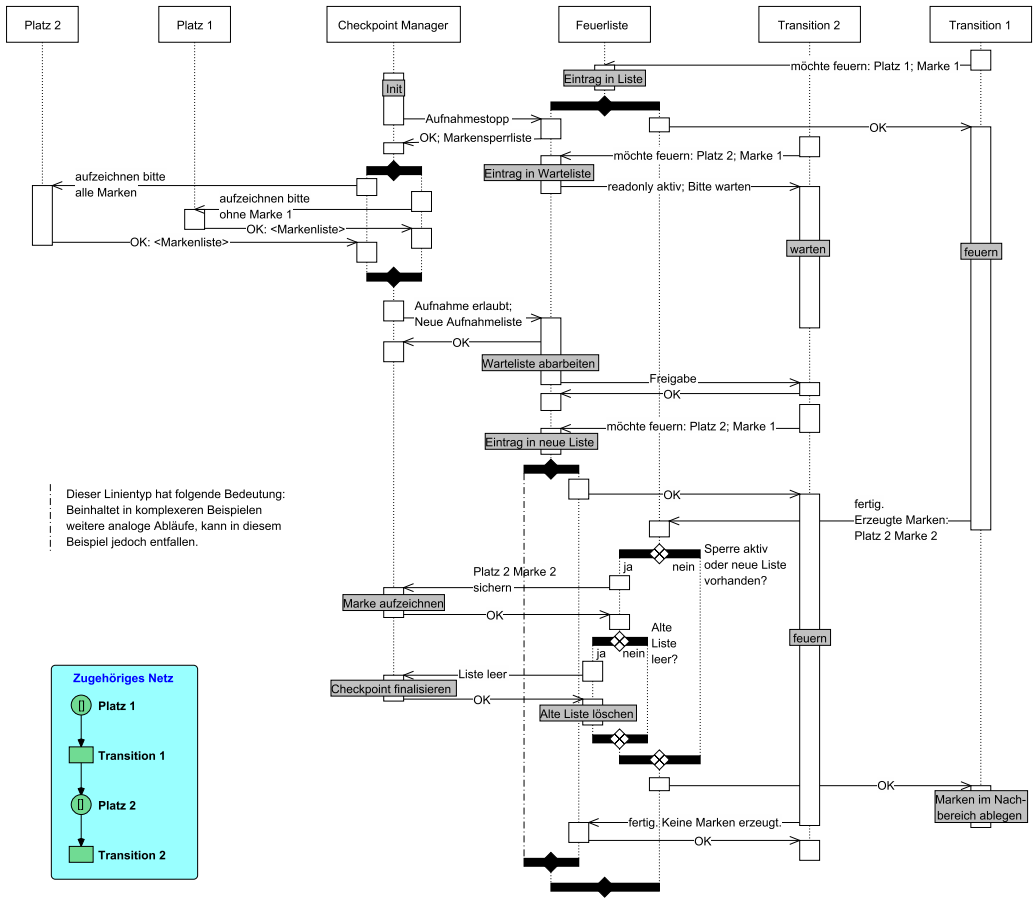

Abbildung 11.1.: True-Concurrency Checkpoints

Sobald der Checkpoint Manager die Erzeugung eines Checkpoints einleitet, wird die Liste mit einer entsprechenden Sperre versehen.

Versucht nun eine Transition zu feuern, wie im Beispiel Transition 1, so wird diese gebeten zu warten. Nach der Sperrung weist der Checkpoint Manager alle Plätze an, alle ihre Marken aufzuzeichnen, ohne jedoch die Marken anzufordern, welche aktuell auf der Sperrliste durch die Feuerliste stehen. Nachdem alle aufgezeichneten Marken eingetroffen sind, gibt der Checkpoint Manager die Feuerliste wieder frei. Die Feuerliste informiert alle wartenden Transitionen, dass diese nun feuern können.

Transition 1 startet nun den Feuervorgang und wird in einer neuen Liste eingetragen. Nachdem Transition 2 den Feuervorgang beendet hat, meldet Transition 2 der Feuerliste die erzeugten Marken. Die Feuerliste prüft, ob aktuell eine Sperre aktiv ist oder bereits eine neue Liste vorhanden ist und kann so herleiten, ob aktuell ein Checkpoint aufgezeichnet wird. Falls ja, werden die erzeugten Marken zur Checkpointerstellung hinzugefügt und falls die Liste der noch feuernden Transitionen jetzt leer ist, wird der Checkpoint finalisiert und die alte (leere) Liste gelöscht. Erst danach legt Transition 2 die erzeugten Marken in ihrem Nachbereich ab.

11.7. Zusammenfassung - Umsetzung der Plattform

In diesem Kapitel wurde eine mögliche Realisierung der Mushu-Plattform im Kontext des Referenznetzsimulators Renew beschrieben. Dabei wurde argumentiert, dass Plattformfunktionalitäten als Plugins und Agenten als Netze dargestellt werden können. Eine entsprechende Uploadfunktion und eine Ladefunktion bzw. Instanziierung müssen dabei noch umgesetzt werden. Die Statusmonitoraspekte der Plattform sollten durch ein eigenes Plugin umgesetzt werden, an dem bestimmte Informationsquellen angemeldet werden können. Informationsquellen sind dabei nicht auf einzelne Plugins beschränkt, sondern Plugins können keine bis beliebig viele Informationsquellen anmelden. Aus dem Kontext der Cloud-Nativity entsteht die Idee der Konstruktion eines Simulationsfeeds, welcher detailliert im Kapitel beschrieben wurde. Dabei wurde insbesondere auch ein (experimentelles) Verfahren für die Erzeugung von Checkpoints unter True-Concurrency-Semantik beschrieben.

12. Realisierungskonzept zur plattformübergreifenden Agentenkommunikation

In Abschnitt 5.3.3 wurde beschrieben, dass für komplexe Agentenkommunikationen mit Zustellungszusicherung eine abgewandelte bzw. erweiterte Form des Saga-Patterns zum Einsatz kommen kann. In vergangenen Kapiteln wurden Realisierungsmöglichkeiten der Konzepte Plattform und Plattformmanagement der Mushu-Architektur diskutiert. In ihrem Kontext kann nun die Erweiterung des Saga-Patterns auf nebenläufige Berechnungen vollzogen werden. Dabei kann die Erweiterung nur schwerlich isoliert betrachtet werden, da viele Komponenten im umliegenden System (Plattformmanagement) eingesetzt werden müssen. Aus diesem Grund betten sich die Ausführungen dieses Kapitels in die bereits betrachteten Realisierungsvorschläge ein.

Sagas werden in aktuellen Arbeiten meist im Kontext von Web- und Microservices betrachtet. Aus diesem Grund sollte das Konzept der nebenläufigen Sagas nach Möglichkeit allgemein und losgelöst von dem hiesigen Einsatzkontext der Agentenkommunikation betrachtet werden. Eine allgemeine Betrachtung ermöglicht auch allgemeinen Micorserviceanwendungen auf die Vorteile nebenläufiger Sagas zurückzugreifen.

Die Abschnitte 12.2 bis 12.6 diskutieren den allgemeinen Aufbau nebenläufiger und petrinetzbasierter Sagas unabhängig vom Einsatzzweck. Abschnitt 12.7 diskutiert den Einsatz speziell im Kontext der verteilten Referenznetzsimulation mit Renew und dem Distribute Plugin. Abschließend gibt Abschnitt 12.8 eine Übersicht über die Realisierungsvorschläge der bisher diskutierten Bestandteile der Mushu-Architektur aus Sicht des Plattformmanagements.

Alle im Folgenden ausgeführten Überlegungen bilden einen wichtigen Beitrag im Rahmen der Neuerungen durch diese Arbeit. Sie wurden im Rahmen des Beitrags (Röwekamp, Buchholz und Moldt, 2021) auf dem PNSE 2021 Workshop veröffentlicht und mit dem dortigen Fachpublikum diskutiert. Ausnahmen hiervon sind als solche gekennzeichnet unter Referenzierung des entsprechenden Autors und den zugehörigen Arbeiten.

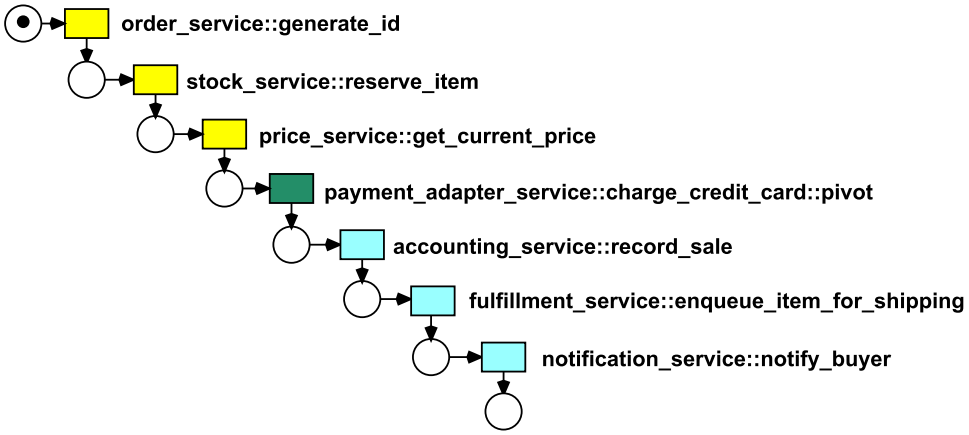

Abbildung 12.1.: Ein Beispiel für eine lineare Saga (ohne explizite Abbildung der kompensierenden Transitionen).

12.1. Vorüberlegungen

Als Einstieg soll ein anschauliches Beispiel dienen. Sagas werden üblicherweise mit einer Zustandsmaschine beschrieben, welche als endlicher Automat oder als einfaches Petrinetz modelliert werden kann. In Abbildung 12.1 ist zu diesem Zwecke eine klassische (nicht nebenläufige) Saga in Form eines sehr einfachen Petrinetzes dargestellt. Jede Transition entspricht hier der Ausführung einer jeweils lokalen Transaktion in einem Service. Zusätzliche Beschreibungen an den Transitionen geben Aufschluss auf den zugehörigen Service und der dort auszuführenden lokalen Transaktion (bzw. angesprochener Schnittstelle). Die Notation mit doppeltem Doppelpunkt (»::«) zwischen Servicebeschreibung und Schnittstelle ist dabei *nicht* unbedingt Saga-typisch, sie wird jedoch im weiteren Verlauf der Arbeit definiert und eingesetzt. Aus Gründen der Einheitlichkeit ist sie hier aber bereits in Verwendung. Sagas unterstützen auch kompensierende Operationen, welche der Übersichtlichkeit halber nicht explizit in der Abbildung dargestellt sind.

Das Szenario beschriebt einen Kunden, der ein Buch bestellt und mit einer Kreditkarte bezahlt. Wenn der Kunde in der Schnittstelle auf die »Jetzt bestellen«-Schaltfläche klickt, wird zunächst eine Order-ID erzeugt und vom Bestellservice gespeichert (erste Transaktion). Das Buch muss dann im System zur Verwaltung der verfügbaren Bestände mit der Order-ID reserviert werden (zweite Transaktion) und der aktuelle Preis muss für die Zahlungsabwicklung abgefragt werden (dritte Transaktion). Danach muss die Kreditkarte mit dem Betrag belastet werden, der zur Bezahlung des Buches und des Versands erforderlich ist. Dabei wird ein externer Zahlungsdienstleister involviert (vierte Transaktion). Nach erfolgreicher Zahlung wird die Bestellung an den Service des Buchhaltungssystems zur

Archivierung (fünfte Transaktion) und an einen Fulfillment-Service zur Beauftragung des Versandes (sechste Transaktion) übergeben. Abschließend erhält der Benutzer eine Quittung per E-Mail durch den Benachrichtigungsdienst (siebte Transaktion).

Dabei ist zu beachten, dass die Transaktionen eins, zwei und drei alle umkehrbar sind, da eine Order-ID wieder gelöscht und eine Buchreservierung storniert werden kann, um das Buch wieder verfügbar zu machen. Die Stornierung einer Kreditkartenabbuchung durch den externen Zahlungsanbieter ist weniger einfach, daher bietet es sich an diese als Pivot-Transaktion zu betrachten. Die Abbuchung entscheidet darüber, ob die Saga in ihrer Gesamtheit scheitert oder nicht. Die Transaktionen fünf bis sieben sind alle wiederholbar und haben keine nicht-technischen Gründe fehlzuschlagen. Bei technischen Fehlern, wie beispielsweise die Unerreichbarkeit des betreffenden Services, wird die Initiierung der dortigen Transaktion solange wiederholt, bis sie erfolgreich ist.

Durch den Einsatz einer Zustandsmaschine wird etwaig vorhandene Nebenläufigkeit, welche innerhalb der Folge von lokalen Transaktionen existieren kann, nicht genutzt. Mögliche Nebenläufigkeit richtet sich in erster Linie nach dem Bedarf an Daten aus vorherigen Schritten und kann in der beispielhaften Saga einfach beobachtet werden. So sind Preisabfrage und Reservierung vermutlich nicht abhängig voneinander (sofern keine kundenspezifischen Preise generiert werden) und können nebenläufig ablaufen. Ebenso ist der Eingang des Versandauftrags und die Benachrichtigung des Kunden unabhängig voneinander. Der Eintrag in das Buchhaltungssystem ist – je nach Maßgabe des Unternehmens – ebenso nicht abhängig von Versand und Benachrichtigung.

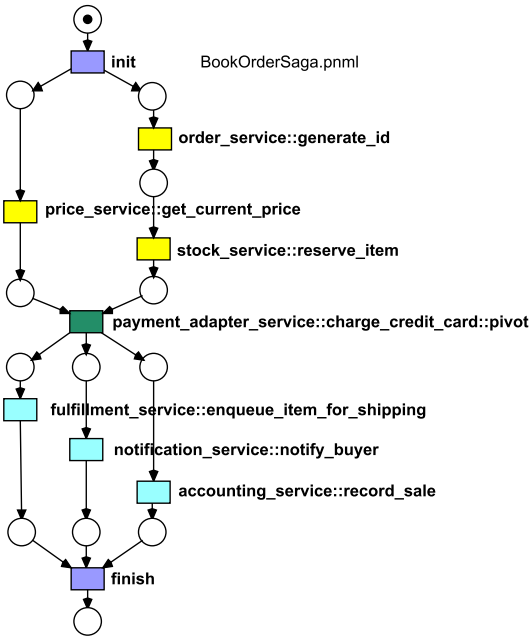

Abbildung 12.2.: Ein Beispiel für eine nebenläufige Petrinetz-basierte Saga

Ein Beispiel für die Vision einer petrinetzbasierten nebenläufigen Saga ist in Abbildung 12.2 zu finden. Es handelt sich dabei um eine Abwandlung des Beispiels aus Abbildung 12.1.

Um den Ansatz konzeptuell möglichst allgemein zu halten, bietet es sich an auf die Betrachtung einer konkreten technischen Implementation zu verzichten. Es existieren verschiedene technische Implementationen und Frameworks für Sagas, auf die im Kapitel 14 zu Prototypen genauer eingegangen wird. An dieser Stelle soll stattdessen ein generalisiertes Interface zwischen Petrinetz-Ausführungsumgebung und Framework definiert werden, sodass die Implementation bzw. das Framework auf Saga-Seite später beliebig austauschbar wird.

Darüber hinaus gilt es zu definieren, in welcher physikalischen Form bzw. durch welches Dateiformat die Saga durch ein Petrinetz beschrieben werden kann. Hierfür gibt es je nach Wahl des Petrinetz-Formalismus eine Vielzahl an möglichen Optionen. Im Allgemeinen sollte bei der Umsetzung darauf geachtet werden, dass ein allgemein verbreitetes und verständliches und nach Möglichkeit standardisiertes Datenformat eingesetzt wird oder zumindest als Quelle unterstützt wird.

Auch die Weitergabe von Daten zwischen den einzelnen jeweils lokalen Transaktionsausführungen innerhalb einer Saga muss erneut untersucht werden. Dies liegt darin begründet, dass zuvor im konventionellen Ansatz auf Basis von sequentiellen Zustandsmaschinen die Daten aus dem zuvor liegenden Schritt eindeutig definiert waren. Durch Ausnutzen von Nebenläufigkeit entsteht nun aber der Fall, dass ein Schritt oder mehrere Schritte von mehreren davor liegenden gleichberechtigten Datensätzen abhängen können. Dem Entwickler des Systems sollte somit eine Möglichkeit gegeben werden Listen bzw. mehrfache Datensätze zu aggregieren, falls die Weitergabe einer einfachen Liste aller vorheriger Ergebnisse nicht genügen sollte.

Insgesamt sollte im Kontext der Skalierbarkeit des Systems darauf geachtet werden, dass durch die Einführung des Saga-Patterns keine weitere Zentralisierung erfolgt. Dies kann beispielsweise durch die Einführung eines global eindeutigen Koordinators erfolgen und hätte zur Folge, dass sich dieser zu einem Flaschenhals und einem Single Point of Failure entwickelt. Da eine der üblichen Umsetzungen des Saga-Patterns auf Basis von Orchestration durch einen Orchestrator erfolgt, ist diese Anforderung nicht trivial umzusetzen. Jedoch auch abseits von einem global eindeutigen Koordinator soll die Einführung des Saga-Patterns das Gesamtsystem nicht unverhältnismäßig ausbremsen.

Auf dieser Basis diskutiert Abschnitt 12.2 die Grundlagen der Petrinetz-basierten Sagas und insbesondere die Auswahl des Basisformalismus der Petrinetzdarstellung, wie mit Fehlschlägen einzelner Transaktionen umgegangen werden kann und sollte sowie wie Daten weitergegeben werden können. Basierend auf den Abwägungen wird in Abschnitt 12.3 eine Entscheidung für eine Definition von Petrinetz-Sagas gefällt. Im Rahmen dieser beiden Abschnitte wird sich ergeben, dass zur Ausführung einer Petrinetz-Saga eine Transformation zwischen zwei Formalismen notwendig ist. Diese Transformation wird in Abschnitt 12.4 beschrieben. Abschlie-

ßend wird das Protokoll zum vorgeschlagenen detaillierten Ablauf der Ausführung einer Petrinetz-Saga in Abschnitt 12.5 dargestellt.

12.2. Grundlegende Konstruktion

Dieser Abschnitt diskutiert die grundlegenden Überlegungen auf dem Weg zur Definition einer Petrinetz-Saga. Da in den folgenden Abschnitten häufig die Begriffe »Transition« und »Transaktion« verwendet werden, sei an dieser Stelle noch einmal auf deren Unterschied hingewiesen. Eine Transition ist ein Element eines Petrinetzes und kann durch Feuern Marken von Plätzen ihres Vorbereichs in Plätze ihres Nachbereichs verschieben. Eine Transaktion ist eine Folge von Operationen, die meist im Kontext von Datenbanken bei Persistenz- und Konsistenzfragen anzutreffen ist, aber nicht unbedingt beschränkt auf die reinen Operationen auf der Datenbank sein muss. Transaktionen genügen in der Regel den ACID Kriterien (vgl. die Ausführungen im Kontext von Abschnitt 2.5.2). Im Kontext von Petrinetz-Sagas wird durch das Feuern einer Transition eine Transaktion im Hintergrund ausgeführt.

12.2.1. Basisformalismus

Die erste notwendige Überlegung umfasst die Wahl des zu Grunde liegenden Formalismus. Die Wahl des Formalismus hat weitreichende Konsequenzen bei allen weiteren Aspekten der Konstruktion von nebenläufigen Sagas. Er bestimmt maßgeblich die zu verwendende Syntax und die Ausdrucksmächtigkeit und Kompaktheit des die Saga beschreibenden Modells.

Der Grundgedanke bei der Konzeption von nebenläufigen Sagas impliziert nicht unmittelbar, dass diese ausschließlich im Kontext der komplexen Agentenkommunikation bzw. von verteilten synchronen Kanälen zum Einsatz kommen müssen. Vielmehr kann die Neuerung auch für sich alleine stehend in gänzlich andere Projekte eingebunden werden. Bezogen auf die Anforderungen an die Umsetzung einer Petrinetz-Saga ist die Abbildung der Saga Semantik als zentral zu sehen. Dennoch sollte ebenfalls die Überlegung in die Wahl des Formalismus einfließen, wie aufwändig eine Einarbeitung in diesen für Entwicklerinnen und Entwickler ohne Kenntnisse über Petrinetze ist. Darüber hinaus sollte der Support des Petrinetz-Formalismus durch Werkzeuge gegeben sein, damit einfach Modelle von nebenläufigen Sagas erzeugt werden können.

P/T Netze

Die einfachste Möglichkeit ein Modell des Ablaufs zu konstruieren, besteht darin P/T-Netze zu verwenden. Sie zeichnen sich durch ihre vergleichsweise simple Syntax und einfache Erlernbarkeit aus. Da sie einen der grundlegendsten Formalismen der Petrinetze darstellen, sind Werkzeuge zur Erstellung weit verbreitet. Sie sind nicht Turing-mächtig und bieten daher eine Reihe exzellenter formaler Analysemöglichkeiten. Darüber hinaus können P/T-Netze als simples Vektoradditionssystem interpretiert werden und eröffnen somit Möglichkeiten Mittel der linearen Algebra auf sie anzuwenden.

Ohne weiteres Zutun können keine expliziten Services oder Operationen spezifiziert werden, um damit lokale Transaktionen zu referenzieren. Daher ist es zwingend erforderlich eine Erweiterung des Formalismus einzusetzen. So können die Transitionen mit Labels versehen werden. Auf diese Art und Weise ist es möglich, dass im Netz zwischen wiederholbaren, Pivot und kompensierbaren Transaktionen unterschieden werden kann.

In einer Saga kann insbesondere die Pivottransaktion fehlschlagen, daher erscheint es notwendig, dass der gewählte Formalismus dieses Verhalten (möglicher Fehlschlag) abbilden kann. In P/T-Netzen kann dies jedoch nicht gewährleistet werden, da eine Transition nur entweder vollständig schalten oder nicht schalten kann. Auch, wenn das Verhalten mit etwa zwei Transitionen abgebildet wird, wobei eine für den Erfolg und die andere für den Fehlschlag der zugehörigen Transaktion steht, kann die intendierte Semantik nicht realisiert werden. Ohne Änderung der zu Grunde liegenden Semantik des P/T-Netzes (vgl. Petrinetzsemantiken in Abschnitt 2.2.5) kann keine Entscheidung für eine von zwei aktivierten Transitionen auf Basis externer Einflüsse erzwungen werden. Entweder können beide nicht feuern oder eine der beiden feuert nicht-deterministisch, die Auswahl erfolgt aber nie auf Grund von äußeren Kriterien. Dies gilt insbesondere, wenn der Ablauf der Saga durch die Simulation eines Netzes gesteuert werden soll und nicht eine Netzdarstellung den Lauf einer ohnehin laufenden Saga abbildet.

Würde eine entsprechende Änderung der Semantik dennoch vorgenommen werden, oder aber das Netz erst in ein anderes Modell transformiert werden, bevor dieses dann ausgeführt wird, können zusätzliche Überlegungen formuliert werden. Neben der Wahl zwischen erfolgreicher und unerfolgreicher Transition, welche anhand der zugehörigen Transaktion gefeuert wird, kann bei der Konstruktion einer Saga dann zusätzlich argumentiert werden, dass ein Misserfolg eine definierte und berechenbare Konsequenz nach sich zieht. Alle zuvor ausgeführten lokalen Transaktionen müssen im Falle eines Fehlschlags kompensiert werden. Der gesamte Aspekt des Kompensierens kann aus dem Anteil der Saga vor der Pivottransaktion berechnet werden und müsste an sich nicht selbst modelliert werden.

Wird also die Ausführung der Saga anders gehandhabt als die direkte Ausführung des abgebildeten Netzes, so kann die Notwendigkeit für eine Unterscheidung zwischen Erfolg und Misserfolg im Modell entfallen, da sich das Verhalten nach einem Misserfolg aus dem erfolgreichen Verlauf ableiten lässt. Dabei muss jedoch beachtet werden, dass es sich bei diesem Verfahren auch um eine Abkehr der zu Grunde liegenden Semantik des P/T-Netzes handelt.

Schlussendlich kann also festgehalten werden, dass mit den zuvor genannten Einschränkung eine Modellierung einer Saga durch P/T Netze generell möglich ist.

Gefärbte Netze

Die nächste zu betrachtende Stufe der Formalismen bilden die gefärbten Netze. Im Gegensatz zu P/T-Netzen unterstützen sie verschiedene Typen von Marken und hätten mehr Möglichkeiten, verschiedene Daten, welche zwischen Transaktionen der Saga übertragen werden sollen, zu modellieren. Darüber hinaus können durch guards Transitionen daran gehindert werden, mit einer Marke in einer bestimmten Farbe zu schalten. Solange die betrachtete Farbmenge des Netzes endlich bleibt, ist das gefärbte Netz nicht Turing-mächtig und bedeutend besser zu analysieren als ein Turing-mächtiger Formalismus.

Durch entsprechende guard-Inschriften an Transitionen könnte zwischen fehlgeschlagenen und erfolgreichen Transaktionen des letzten Schrittes unterschieden werden. Jedoch kann durch einen Feuervorgang die erzeugte Marke nicht dynamisch geändert werden, je nachdem, welchen Ausgang die zu Grunde liegende Transaktion hatte. Dies liegt darin begründet, dass die ausgehenden Kanten auch in gefärbten Netzen mit lediglich einer Markenfarbe bzw. Variable beschrieben sind. Das Problem, welches bereits im vorigen Abschnitt zu P/T Netzen beschrieben wurde, besteht mit gefärbten Netzen also weiterhin.

Gefärbte Netze besitzen zudem eine nicht zu unterschätzende Komplexität und könnten eine Herausforderung beim Erlernen darstellen. Die Darstellung kann aufgrund der verwendeten Markenfarbe jedoch kompakter erfolgen, als dies bei der Verwendung von P/T-Netzen möglich wäre. Darüber hinaus existiert eine Vielzahl an Werkzeugen für gefärbte Petrinetze, wie beispielsweise die CPN-Tools (Ratzer u. a., 2003).

Referenznetze

Referenznetze sind der mit Abstand komplexeste der hier vorgestellten Petrinetzformalismen. Sie überzeugen durch ihre Eigenschaft mittels Referenzierung komplexe hierarchische Strukturen auszudrücken und (Referenzen auf) nicht-triviale Objekte als Marken halten zu können. In ihrer Implementation im Simulator Re-

NEW können sie beliebige Java-ähnliche Statements beim Feuern ihrer Transitionen ausführen. Dabei ist die Funktionalität nicht auf die Ausdrücke innerhalb der Inschriften begrenzt, sondern viel mehr können beliebige Methoden angeschlossener Klassen, Objekte und Bibliotheken ausgeführt werden.

Diese Eigenschaften führen dazu, dass entgegen den beiden Formalismen der P/T-Netze und der gefärbten Netze das Ergebnis einer Transition sehr wohl nachträglich unterschieden werden kann. Der Ausgang einer Transaktion kann beispielsweise genutzt werden, um in einem geeigneten Ergebnisobjekt ein Feld zu setzen, welches diesen Ausgang abbildet. Die folgenden Transitionen können dann durch guards auf dieses Feld prüfen und ggf. das Feuern verweigern bei einem Fehlschlag der davor liegenden Komponente.

Der umfassendste Simulator und die umfassendste Umgebung für die Erstellung von Referenznetzen ist Renew. Durch die geringe Auswahl an Werkzeugen ist die Entwicklung auf den Einsatz von Java als Programmiersprache eingeschränkt. Darüber hinaus sind Referenznetze Turing-mächtig, wodurch sie auch sehr komplexe Sachverhalte abbilden können, jedoch auch Analysemöglichkeiten durch die allgemein bekannten Ergebnisse bezüglich Turing-mächtiger Sprachen limitiert sind. Dennoch wurde bereits argumentiert, dass zum Abbilden beliebiger Daten bereits gefärbte Netze mit beliebigen Farbmengen notwendig sind, welche bereits Turing-mächtig sind. In dieser Hinsicht geht durch die Turing-Mächtigkeit der Referenznetze somit kein Vorteil verloren.

Durch ihre Komplexität können Referenznetze eine erhebliche Einstiegshürde bei der Entwicklung von Petrinetz-Sagas bilden. Auch durch die Eigenschaft, dass Operationen als Java-Anschriften an Transitionen zu finden sein können, bergen Referenznetze das Potenzial schnell unübersichtlich werden zu können. Daher ist bei der Konstruktion von Referenznetzen größere Sorgfalt nötig, eine Fähigkeit die insbesondere auch Erfahrung mit ihrem Umgang erfordert. Um eine saubere Struktur eines Referenznetzes zu forcieren, können Templates wie beispielsweise NetComponents aus den Arbeiten Cabacs[1] helfen.

12.2.2. Fehlschläge

Einer der interessantesten Aspekte bei der Konstruktion von Petrinetz-Sagas umfasst die Handhabung von fehlgeschlagenen Transaktionen. Die Transitionen einer Petrinetz-Saga symbolisieren je die Ausführung einer bestimmten Transaktion. Daher würde es naheliegen, in die Transitionen eine Art Fehlschlagssemantik einzubauen.

[1] Siehe dazu die entsprechenden Arbeiten (Cabac, 2003, 2009, 2010)

Eine Art von Fehlschlagssemantik existiert bereits in Form der Task-Transition[2], welche im konzeptuellen Kapitel 5 zur Agentenkommunikation auf einer höheren Abstraktionsebene angewendet wurde, um die Nutzbarkeit des Saga-Patterns herzuleiten und dieses Kapitel zu motivieren. Diese Art von Transitionen kann fehlschlagen und damit die konsumierten Marken zurück in den Vorbereich der Task-Transition legen. Bei der Anwendung im inneren Kontext von Petrinetz-Sagas besteht dabei jedoch das Problem, dass nach einer fehlgeschlagenen Transaktion gegebenenfalls das Netz einen Abbruch der Saga und damit ein Kompensieren der bisher ausgeführten Transaktionen einleiten muss. Zu diesem Zweck muss das Netz in der Lage sein, diese Art von Fehlschlägen erkennen zu können. Werden jedoch einfach die Marken zurück in den Vorbereich der fehlgeschlagenen Transition gelegt, so kann der Zustand des Vorbereichs nicht von dem unterschieden werden, der vor dem erstmaligen Feuern der Transition bestand. Auf abstrakter Ebene ist dies das gewünschte Verhalten der Kommunikation von Agenten, im lokalen Fall sind jedoch genau aus diesem Grund Task Transitionen nicht unmittelbar erneut anwendbar. Darüber hinaus wird bei einer Petrinetz-Saga durch eine Transition auch nur die Ausführung einer Transaktion dargestellt. Auch bei einem Fehlschlag als Ausgang der ausgeführten Transaktion war die Ausführung selbst jedoch erfolgreich. Damit wäre die Semantik der Task-Transition an dieser Stelle auch nicht vollständig korrekt.

Eine Alternative besteht darin, jede Transaktion durch zwei Transitionen im Saga Netz darzustellen. Eine der beiden Transitionen würde dabei für eine erfolgreiche Ausführung einer Transaktion stehen und die andere für eine fehlgeschlagene. Bei der konkreten Umsetzung würde auch hier eine Abhängigkeit zum verwendeten Basisformalismus bestehen. Beim Einsatz von P/T-Netzen könnten zwar sowohl die Transition für den Fehlschlag als auch die Transition für den Erfolg alternativ geschaltet werden, die Auswahl müsste allerdings auf Basis der dahinter liegenden Transaktion erfolgen. Auch dies wäre wie im vergangenen Abschnitt erläutert ein Eingriff in die Semantik von P/T-Netzen. Darüber hinaus müsste streng genommen der Ausgang der Transaktion bereits vor dem entsprechenden Feuern der zugehörigen Transition feststehen, um auf der Basis eine Entscheidung treffen zu können. Dadurch stellt sich die Frage, an welcher Stelle die Ausführung der Transaktion ausgelöst werden soll, wenn die Koppelung an den Feuervorgang Probleme bereitet. Das Zwischenschalten einer Transition, welche lediglich die Ausführung der Transaktion abbildet und der mit ihr alternierende Einsatz von Erfolgs- und Fehlschlagstransitionen könnte dieses Problem umgehen. Dabei würde das Netz jedoch deutlich größer ausfallen und große Sagas könnten potenziell Probleme hinsichtlich der Übersichtlichkeit aufweisen.

Ein weitere Aspekt bezüglich dem Fehlschlagen von Transaktionen ist, dass in der Literatur kompensierbare Transaktionen meist nur durch ebendiese Eigen-

[2] vgl. (Aalst u. a., 1999; Jacob, 2002).

schaft charakterisiert sind. Über Fehlschläge dieser Art von Transaktionen wird selten diskutiert. Da aber die Annahme, dass jede Transaktion ungeachtet ihrer Trivialität oder lokalen Verortung fehlschlagen kann, sollte im Rahmen der Petrinetz-Sagas auch für kompensierbare Transaktionen eine Fehlschlagsemantik definiert werden. Da nicht bekannt ist, ob diese Transaktionen problemlos wiederholbar sind, sondern nur, dass sie kompensierbar sind, sollte eine etwaige Fehlschlagsemantik auch nur auf diese Eigenschaft aufbauen. Es liegt daher nahe, die gesamte Saga im Falle eines Fehlschlags einer kompensierbaren Transaktion abzubrechen. In diesem Fall müssen alle davor ausgeführten Transaktionen entsprechend kompensiert werden.

Dies ist im sequenziellen Fall unproblematisch, da einfach die Reihenfolge der ausgeführten Transaktionen rückwärts kompensiert werden kann. Durch die Einführung von Nebenläufigkeit ist diese Reihenfolge jedoch nicht mehr offensichtlich. Bei der Bestimmung der Kompensationsreihenfolge ist es hilfreich, darauf zu schauen, welche Transaktionen überhaupt nebenläufig zueinander ausgeführt werden konnten. Dies sind eben diese Transaktionen, welche jeweils gegenseitig nicht auf ihre Ergebnisdaten zurückgreifen müssen. Bei der Betrachtung einer gesamten Petrinetz-Saga mit allen ihren nebenläufig Anteilen müssen ebendiese Daten wieder kompensiert werden können, wenn alle von den Daten abhängigen Transaktionen zurückgerollt wurden. Konkret bedeutet dies, dass eine Transaktion erst dann kompensiert werden kann, wenn in alle Plätzen, in die die zugehörige Transition im Erfolgsfall Marken abgelegt hat, durch zu kompensierenden Transaktionen gehörenden Transitionen erneut Marken abgelegt wurden. Effektiv bedeutet dies, dass eine Invertierung aller Kanten des Netzes vor der Pivottransition das gewünschte Verhalten erzeugt.

Eine Herausforderung speziell durch die Integration von Nebenläufigkeit in das Modell, besteht jedoch darin, nebenläufig ablaufende Transaktionen gegenseitig von Fehlschlägen zu unterrichten, da es sonst zu Verklemmungen kommen kann. Dieses Verhalten soll im Folgenden illustriert werden. Spätestens zum Pivotelement oder am Ende der Saga ist eine Zusammenführung bzw. Synchronisation der einzelnen nebenläufigen Transaktionen notwendig. Wenn nun spezielle Transitionen für Fehlschläge in das Netz integriert werden und diese anstelle der Erfolgstransitionen geschaltet werden, so fehlen ab einem gewissen Punkt Marken, um eine Synchronisation mit dem anderen nebenläufigen Strang einzugehen. Für diese Besonderheit ist es sinnvoll, ein Beispiel zu betrachten.

Abbildung 12.3 zeigt beispielhaft die Idee einer Petrinetz-Saga auf der Basis eines P/T-Netzes. Das Beispiel setzt eine Semantikänderung voraus, sodass eine entsprechende Transition anhand des Ausgangs der zugehörigen Transaktion schalten kann.

Im Beispiel ist lediglich eine Transition für Fehlschläge exemplarisch abgebildet. Dabei handelt es sich um die Transitionen $t2f$, welche schaltet, wenn die zu

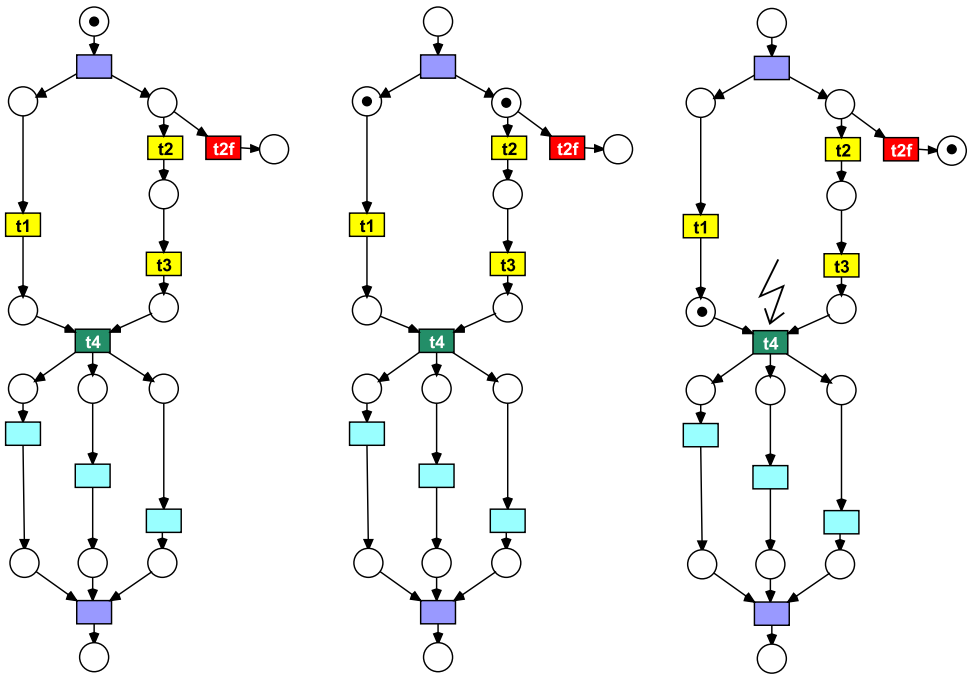

Abbildung 12.3.: Fehlschläge in nebenläufigen Sagas sind nicht auf die Lokalität begrenzt

$t2$ zugehörige Transaktion fehlschlägt. Für ein weiteres korrektes Verhalten des Netzes wäre es notwendig, ab dieser Stelle alle bereits ausgeführten Transaktionen zu kompensieren. Da an dieser Stelle jedoch nur die Verklemmungsproblematik verdeutlicht werden soll, ist der restliche Teil des Netzes nicht abgebildet.

Nachdem die initiale Transition geschaltet hat, befindet sich hier je eine Marke im jeweiligen Vorbereich von $t1$ und $t2$. Schaltet nun $t1$, da die Transaktion der $t1$ zugeordnet ist, erfolgreich verlaufen ist, liegt eine Marke im Vorbereich von $t4$. Kommt es jedoch zum Fehlschlag bei der Transaktion von $t2$, so wird logischerweise $t3$, welches von $t2$ abhängt, niemals geschaltet. Folglich erreicht niemals eine Marke den Vorbereich von $t4$, der gleichzeitig Nachbereich von $t3$ ist. Existiert nun keine globale Fehlschlagserkennung, so würde das Netz an dieser Stelle verklemmen, da $t4$ ewig auf die zweite Marke im Vorbereich warten würde, um aktiviert zu werden. Gleichzeitig könnte zwar $t2$ kompensiert werden (nicht abgebildet), die Saga könnte jedoch durch die Verklemmung auch so nicht zu einem Ende kommen, da keine Information in der Lokalität von $t1$ zu finden ist, welche eine Kompensation von $t1$ rechtfertigen würde.

Da die Verklemmung nicht unmittelbar von der Lokalität der nicht aktivierten Transition abhängt, folgt die Notwendigkeit einer globalen Fehlschlagserkennung. Ein gegenseitiges Unterrichten ist generell unproblematisch, da ein Orchestrator die Ausführung der Petrinetz-Saga betreut. Dieser besitzt ohnehin eine globale Sicht auf den Kontrollfluss der konkreten Saga und kann entsprechende Schaltvorgänge umsetzen. Dieses Ergebnis wird im Abschnitt 12.4 zur Transformation erneut aufgegriffen. Das im Beispiel angesprochen Kompensieren wird ebenfalls dort erörtert und ausgeführt werden.

Während im Kontext von Petrinetz-Sagas insbesondere orchestratorbasierte Sagas betrachtet werden, lohnt eine kurze Überlegung bezüglich dieses Sachverhaltes in choreografiebasierten Sagas. Diese besitzen keinen globalen Koordinator und das Wissen über den nächsten Sagateilnehmer ist lokal beim aktuellen Teilnehmer verankert. Für eine globale Fehlschlagserkennung kann daher der Fehlschlag trotzdem an den nächsten Teilnehmer weitergeleitet werden, welcher den Fehlschlag dann ebenfalls ohne lokale Aktion an den wieder nächsten Teilnehmer weiterleitet. Der Service, welcher das Pivotelement darstellt, bekommt so früher oder später die Information über einen Fehlschlag an einer Stelle und kann einen globalen Kompensiervorgang einleiten. Während diese Variante auch zum Erfolg führt, ist der Aufwand jedoch offensichtlich höher, da in ungünstigen Fällen viele Services involviert werden müssen, welche lediglich die entsprechende Information des Fehlschlags weiterleiten, ohne dass sie lokal produktive Arbeit verrichten.

12.2.3. Datenweitergabe

Sagas beschreiben eine Folge oder im Falle der Petrinetz-Sagas ein nebenläufiges Modell von verschiedenen lokalen Transaktionen. Dabei kann in den seltensten Fällen davon ausgegangen werden, dass alle diese lokalen Transaktionen in kompletter Unabhängigkeit voneinander ausgeführt werden können. Abhängigkeiten drücken sich vor allem in etwaigen benötigten oder erzeugten Daten aus. Daher ist es unabdingbar, die Thematik der Datenweitergabe zwischen unterschiedlichen lokalen Transaktionen zu adressieren.

Der erste nahe liegende Ansatz besteht darin, dass jede lokale Transaktion all diejenigen Daten an den Orchestrator weitergibt, welche für die folgende Transaktion notwendig sind. Dieser naive Ansatz bringt jedoch eine Vielzahl an Nachteilen mit sich. Somit müsste implizit Wissen über nachfolgende Transaktionen in einer Transaktion verankert werden, um entscheiden zu können, welche Daten weitergegeben werden sollen. Eine derartige engere Kopplung ist in einem skalierenden System niemals von Vorteil. Eine weitere Problematik besteht darin, dass bestimmte Services nicht nur Daten erhalten, die gegebenenfalls nicht für sie bestimmt sind, sondern auch solche, die bestimmten Geheimhaltungs- bzw. Schutzkriterien unterlegen sind, wie beispielsweise personenbezogene Daten. Darüber hinaus können die Daten auch je nach Anwendungsfall sehr umfangreich ausfallen. Eine Weitergabe durch den Orchestrator, welcher zu einer Zeit viele Sagas auf einmal koordiniert und betreut, kann diesen somit zum Flaschenhals werden lassen.

Betrachtet man jedoch den Kontext, in dem Sagas in modernen verteilten Systemen eingesetzt werden, stellt sich die Frage, ob eine vollständige direkte Weitergabe der Daten explizit notwendig ist. Microservices sollten in einer fundierten Architektur ohnehin entlang bestimmter Einheiten im abgebildeten System geschnitten sein. Wird nun der Ansatz verfolgt, dass zusammengehörige Daten weitestgehend innerhalb von einem Service bearbeitet werden können, reduziert sich die Notwendigkeit für die Menge ausgetauschter Daten zwischen Services.

Darüber hinaus benötigen Microservices ohnehin eine Basis, auf der Information zwischen Services geteilt werden können. In vielen Fällen wird dies auf Basis des Event-Sourcings[3] umgesetzt. Es kann daher davon ausgegangen werden, dass eine lokale Transaktion entsprechend ihres Ausgangs für andere Services relevante Events publiziert. In diesem Fall kann sich die Datenweitergabe gegebenenfalls auf die ID des Events beschränken oder aber gänzlich ausbleiben, wenn nachträglich ausgeführte Transaktionen vor ihrer Ausführung für eine vollständige Verarbeitung aller neuen Events sorgen.

[3] vgl. Abschnitt 2.6.1

In diesem Hinblick ist es jedoch erforderlich, mit Bedacht die verschiedenen Szenarien zu implementieren: Werden sowohl Sagas als auch unabhängige Events im System eingesetzt, so sollte die Reaktion auf bestimmte Events, welche einer Saga zugehörig sind, verzögert werden, bis die zugehörige lokale Transaktion vom Orchestrator angestoßen wird. Setzt das System ausschließlich Sagas zur Kommunikation ein und werden somit Events nur nach Aufforderung von den einzelnen Services verarbeitet, entfällt diese zusätzliche Komplexität. Dabei sollte jedoch bedacht werden, dass Sagas einen speziellen Fall von Operationen abbilden (charakterisiert durch die Dreiteilung zwischen kompensierbaren, Pivot- und wiederholbaren Transaktionen). Nicht alle Formen von übergreifenden Datenoperationen sind mit dieser Charakteristik vereinbar. Somit stellt es gegebenenfalls in sich eine Herausforderung dar, das gesamte Konsistenzmanagement des Systems auf Sagabasis zu implementieren.

Ein weiterer Aspekt ergibt sich speziell durch die neu eingeführte Nebenläufigkeit innerhalb der Saga. Während bei einer Folge von Operationen keine Zweideutigkeit bei den Daten der Voroperation bestehen kann, so können im nebenläufigen Fall zwei oder mehr unmittelbar zuvor liegende Ergebnisdatensätze existieren. Petrinetz-Sagas und das Saga-Pattern sind absichtlich sehr allgemein gehalten, daher wäre die Definition einer allgemeingültigen Methodik zur Integration mehrfacher Datensätze nicht zielführend. Vielmehr sollte bei der Implementation darauf geachtet werden, dass etwaigen Konsumenten einer Petrinetz-Saga Bibliothek die Möglichkeit eingeräumt wird, um eine eigene Aggregation mehrfacher Ergebnisdatensätze zu implementieren.

12.3. Definition einer Petrinetz-Saga

Die zuvor aufgeführten verschiedenen Umsetzungsmöglichkeiten für Petrinetz-Sagas ermöglichen nun die Entscheidung für eine konkrete Ausgestaltung, sowie die Definition der Syntax einer Petrinetz-Saga.

12.3.1. Entscheidung

Nach Abwägung der verschiedenen Vor- und Nachteile ist die Entscheidung für eine Darstellung als gelabeltes P/T-Netz gefallen. P/T-Netze bieten *nicht* die Möglichkeit, zwischen Fehlschlag und Erfolg der jeweiligen Transition zu unterscheiden. Durch die oben ausgeführten Umstände, dass Fehlschlag-Transitionen aus dem erfolgreichen Pfad durch eine Saga berechnet werden können, erscheint die Notwendigkeit, diese Transitionen abbilden zu können, jedoch nicht mehr unbedingt erforderlich.

P/T-Netze überzeugen durch ihre Einfachheit und sind daher in überschaubarer Zeit ohne große Vorkenntnisse erlernbar. Labels sind erforderlich, um Services und Endpunkte bzw. Transaktionen adressieren zu können.

Während die reine Darstellung der Petrinetz-Saga durch ein P/T-Netz erfolgen soll, stellt sich die Frage, in welcher Form ein etwaiger Simulator dieses Netz interpretieren soll. Hierbei müsste vorhandene Nebenläufigkeit innerhalb des Netzes explizit implementiert werden und zusätzlich eine Form von Rückwärtsschalten der Transitionen nach Fehlschlägen angeboten werden.

Diese Änderung würden direkt eine Semantikänderung der P/T-Netzdarstellung bedeuten. Referenznetze konnten ihre Vorteile vor allem im Rahmen ihrer Ausdrucksmächtigkeit verorten. Durch die grundlegende Ähnlichkeit der beiden Formalismen liegt es somit nahe, zur Simulation das P/T-Netz zunächst in ein Referenznetz zu transformieren, welches das gewünschte Verhalten mit normaler Referenznetzsemantik abbilden kann.

Da Referenznetzen konkrete Objekte als Marken referenzieren können, kann das implementierte Datenformat aus Abschnitt 12.2.3 durch Vererbung leicht integriert werden. Die Implementation der Fehlschläge wird noch einmal gesondert im Transformationskapitel adressiert, da es sich dabei primär umd as Hinzufügen berechenbarer Netzteile handelt. Alle folgenden Abschnitte dieses Kapitels basieren auf der Grundannahme, dass eine Petrinetz-Saga durch ein P/T-Netz definiert, in ein Referenznetzen transformiert und dann als solches für die Ausführung der Saga simuliert wird.

12.3.2. Syntax und Struktur

Während die Einführung des Petrinetzformalismus als Grundlage generelle Nebenläufigkeit erlaubt, so kann die Charakteristik einer Saga nicht unmittelbar aufrechterhalten werden, wenn nicht die drei Blöcke an Transaktionsarten sequenziell belassen werden[4].

Somit ergibt sich für jede Petrinetz-Saga die Unterteilung in drei Bestandteile: Transitionen vor der Pivot-Transition, welche den kompensierbaren Transaktionen entsprechen, der Pivot Transition, die der Pivottransaktion entspricht, sowie Transitionen nach der Pivot-Transition, welche den wiederholbaren Transaktionen entsprechen.

[4]Die Aufweichung dieser Bedingung könnte als Gegenstand weiterer Arbeiten denkbar sein.

Zyklen

Das Einführen eines Petrinetz-Formalismus anstatt einer festen Aktionsfolge ermöglicht potenziell auch die Abbildung von Zyklen in der Ausführung. Dabei stellt sich jedoch die Frage, wie sich die Semantik eines solchen Zyklus definieren sollte. Darüber hinaus kann jeder Zyklus nur zu einem produktiven Ergebnis führen, wenn in einer bestimmten Form eine Art Abbruchkriterium definiert ist. Unendlich laufende Zyklen liefern kein endgültiges Ergebnis, könnten aber durch Seiteneffekte sinntragende Elemente der Modellierung sein. Der Grundgedanke des Saga-Patterns liegt darin, eine Alternative zur verteilten Transaktionsausführung zu sein und somit nach endlicher Zeit ein Ergebnis zurückzumelden. Daher kann argumentiert werden, dass die Integration eines Zyklus mit dem zuvor beschriebenen Verhalten nicht innerhalb der erwarteten oder sinnvollen Anwendungsfälle liegt und daher nicht weiter beachtet werden kann.

Während Zyklen in P/T-Netzen grundlegend definiert werden können, soll für den hier beschriebenen Kontext von Sagas eine Überführung von der statischen Aktionsfolge in ein nebenläufiges Modell genügen und Zyklen zunächst nicht betrachtet werden. Eine weitere Fassung von Petrinetz-Sagas könnte dann Überlegungen speziell zu Zyklen im Modell enthalten und damit weit komplexere Abläufe abbilden, als es mit herkömmlichen Sagas möglich wäre.

Pfade und Alternativen

Ferner soll in dieser ersten Variante von Petrinetz-Sagas keine (nicht-deterministische) Auswahl von Pfaden durch das Netz vorgesehen sein. Explizit bedeutet dies, dass keine Verzweigung an Plätzen vorgesehen ist und alle Verzweigungen und Zusammenführungen von Transitionen ausgehen. Darüber hinaus existiert ein expliziter Startplatz für die Saga und ein expliziter Endplatz. Sagas müssen so aufgebaut sein, dass aus jeder erreichbaren Markierung eine Markierung existiert, bei der lediglich eine Marke im Endplatz liegt. Petrinetz-Sagas in ihrer hier vorgestellten Form sind somit sowohl (korrekte) Workflownetze als auch Kausalnetze.

Aus diesem Grund bieten sie exzellente Möglichkeiten der Analyse und Verifikation. Da Petrinetz-Sagas jedoch nicht mit der klassischen Petrinetz Semantik direkt simuliert werden, sondern wie zuvor motiviert per Transformation in ein Referenznetz transformiert werden, können nicht alle Theoreme aus diesem Bereich uneingeschränkt und direkt benutzt werden. Verifikationsgedanken werden ebenfalls kommenden Arbeiten überlassen, Abschnitt 12.6.3 gibt jedoch bereits einige Gedankenanstöße in diese Richtung.

Beschriftungen

Neben der Struktur des Netzes, welches die Petrinetz-Saga beschreibt, gilt es eine Syntax für die Transitionslabel zu definieren. Durch die Ausführung bezüglich der Datenstruktur, welche zur Weitergabe von Daten eingesetzt werden soll, ist ersichtlich, dass eine explizite Definition von Daten an dieser Stelle entfallen kann. Die relevanten Informationen beschränken sich somit auf die Adressierung des zu kontaktierenden Services und einem Identifikator der dort auszuführenden Transaktion. Lediglich das Pivotelement benötigt zusätzlich noch einen Identifikator, um es von einem möglichen sequentiellen Anteil innerhalb der kompensierbaren oder wiederholbaren Transaktionen abzugrenzen. Dieser ist notwendig, um in folgenden Schritten automatisierte Transformationsalgorithmen verlässlich konstruieren zu können, welche zwischen Transitionen vor und nach der Pivottransaktion unterscheiden können.

Da in der späteren Verarbeitung dieser Strings entsprechende Trennzeichen keinen nennenswerten Unterschied in der Performance bewirken, ist die Wahl relativ frei. Da eine Implementation im Kontext der Programmiersprache Java später jedoch durch die Grundlage, welche der Simulator Renew stellt, wahrscheinlich ist, sollte auf ein Escape Zeichen der Sprache Java entsprechend verzichtet werden. In Java können Funktionen von Klassen oder Objekten direkt mit dem Methodenreferenzoperator (»::«) referenziert werden. Aus diesem Grund fiel die Entscheidung, auch die Trennung zwischen adressiertem Service und gewünschter Transaktion mit diesen Trennzeichen zu realisieren.

Zuletzt sollten noch die erste und die letzte Transition innerhalb der Saga eine besondere Bezeichnung erhalten. Dies begründet sich ebenfalls darin, dass die Implementation etwaiger Transformationsalgorithmen somit erleichtert werden kann. Die Eigenschaft von Petrinetz-Sagas ebenfalls (korrekte) Workflownetze zu sein, legt dies ebenfalls nahe. Die Transition hinter dem Startplatz erhält somit das Label »init« und die Transition vor dem Endplatz erhält das Label »finish«.

Die Syntax entspricht somit im Wesentlichen der Darstellung der Vision in Abbildung 12.2.

12.4. Transformation

Wie in Abschnitt 12.3.1 erläutert wurde, soll eine gegebene Petrinetz-Saga von ihrer P/T-Netzdarstellung in eine ausführbare Referenznetzdarstellung transformiert werden. Dieses Unterfangen ist im Kern eine deterministische Transformation, ist durch ihre verschiedenen Eigenschaften jedoch trotzdem komplex. Im folgenden Abschnitt werden die Anforderungen an diese Transformation noch ein-

mal detailliert aufgegriffen und ein Schritt für Schritt Algorithmus beschrieben, um die Transformation auszuführen.

12.4.1. Planung und Aufbau

Zunächst ist zu klären, ob die gewünschte Funktionalität nicht bereits durch bestehende Arbeiten und Technologien erbracht werden kann. Die Transformation umfasst als Ausgangspunkt ein P/T-Netz und als Zielstruktur ein Referenznetz. In diesem Kontext ist insbesondere der »*Renew Metamodelling and Transformation*« (RMT) Ansatz[5] von Haustermann und Mosteller zu nennen. RMT kann dazu verwendet werden, grafische Domain Specific Languages (DSLs) in eine Referenznetzdarstellung zu überführen. In der neuen Darstellung können diese dann simuliert bzw. ausgeführt werden. Somit ist RMT als Metamodellierungstool zu verstehen, da es verwendet werden kann, um Modelldefinitionen seinerseits wiederum zu definieren. RMT definiert dabei Übersetzungsschlüssel für die Lokalitäten verschiedener Elemente der DSL in äquivalente Referenznetzkonstrukte.

Werden nun Petrinetz-Sagas oder speziell die zugrunde liegenden P/T-Netze als DSL betrachtet, liegt es nahe, RMT für diese Aufgabe einzusetzen. Die Problematik entsteht jedoch dadurch, dass Metamodelle in RMT stets durch die Lokalität der Modellierungselemente definiert werden. Während dies für weite Teile der Petrinetz-Saga ausreichend ist, benötigen jedoch gewisse Bestandteile Wissen, welches über die Lokalität des Bestandteils hinausgeht, um erfolgreich transformiert werden zu können. Als Beispiel ist hierbei die Unterscheidung zwischen Transitionen vor und nach dem Pivotelement zu nennen. Während die davor liegenden Transitionen eine entsprechende Fehlerbehandlung mit Abbruch und Kompensieren der Saga abbilden müssen, müssen Fehlerbehandlungstransitionen nach dem Pivotelement lediglich die Transaktion erneut starten, sobald der zugehörige Service verfügbar ist. Daher kann RMT in diesem Fall unglücklicherweise nicht umfänglich eingesetzt werden.

Folglich ist es notwendig, den Transformationsalgorithmus selbstständig zu implementieren. Da es sich hauptsächlich um die Transformation eines Graphen handelt, basieren viele Bestandteile der Transformation auf erweiterten Breiten- oder Tiefensuchealgorithmen[6]. Dabei ist die Reihenfolge der Umsetzung der einzelnen Schritte hilfreich, um dabei Sonderfälle in der Abarbeitung zu vermeiden und Aufwand einzusparen. So ist es beispielsweise sinnvoll, den Teil der Referenznetzdarstellung, welcher das Kompensieren der Saga abbildet, zunächst mit den entsprechenden Operationen (Inschriften) zu versehen, bevor er mit dem Rest integriert wird. So können in einem Schritt einfach alle Inschriften zu kompen-

[5] vgl. (Mosteller, Cabac und Haustermann, 2016)

[6] Für eine Einführung zu diesen Algorithmen siehe beispielsweise das Standardwerk (Cormen u. a., 2001).

sierenden Operationen umgeschrieben werden. Dabei ist zu beachten, dass die vorgeschlagene Reihenfolge zwar günstig ist, aber nicht zwangsläufig so umgesetzt werden muss.

Als letzte Option wäre eine Kombination der oben genannten Algorithmen zusammen mit RMT denkbar. Eigene Algorithmen sind dann nur an den Stellen erforderlich, an denen globales Wissen eingesetzt werden muss. Der Aufbau von Petrinetz-Sagas bringt jedoch selbst bereits eine erhebliche Komplexität mit sich. RMT selbst weist ebenfalls eine nicht geringe Komplexität auf. Aus diesen Gründen wurde im Rahmen der Arbeit von einer Kombinationslösung abgesehen, um die Komplexität des Gesamtsystems auf einem vertretbaren Niveau zu halten.

Die folgenden Unterabschnitte bezeichnen die einzelnen notwendigen Schritte bei der Konstruktion einer Referenznetzdarstellung aus P/T-Netz Petrinetz-Sagas. Dabei wird exemplarisch die Petrinetz-Saga aus Abbildung 12.2 zu ihrem Referenznetzgegenstück transformiert. Die jeweiligen Zwischenstände nach den einzelnen Schritten werden grafisch dargestellt. Dabei ist zu beachten, dass die Grafiken die Bezeichnung »rollback« bzw. »zurückrollen« verwenden anstatt »compensate« bzw. »kompensieren«. Dies liegt darin begründet, dass die Operation aus Sicht der gesamten Saga einem Zurückrollen entspricht. Der erreichte Zustand nach allen Kompensationsoperationen entspricht dem Zustand vor der Ausführung. Auf dieser Basis erfolgte auch die Argumentation bezüglich der Task-Transition in Kapitel 5.

12.4.2. Kompensieren der Prä-Pivottransitionen

Als Erstes kann der Bereich der Saga, welcher für die Kompensieroperationen verantwortlich ist, betrachtet werden. Wie im Abschnitt 12.2.2 beschrieben, sollte im Zweifel das Abbrechen der Saga und Kompensieren der bereits erfolgten Transaktionen aus jedem Zustand vor Ausführung des Pivotelements möglich sein.

Ein Zustand bedeutet in einem Petrinetz stets eine bestimmte Markierung. Ziel beim Kompensieren sollte nun sein, dass im Fehlerfall sämtliche Schaltvorgänge, welche bereits innerhalb dieser speziellen Saga ausgeführt wurden, in umgekehrter Reihenfolge »zurückgeschaltet« werden. Die hier angesprochene Reihenfolge bezieht sich auf die partielle Ordnung der Transitionen in einem Workflownetz (da eine Petrinetz-Saga nach Definition auch ein Workflownetz ist). Konkret bedeutet dies, dass Transitionen, die beim bisherigen Lauf der Saga nebenläufig schalten konnten, auch beim Kompensieren nebenläufig schalten können sollen. Daher liegt es nahe, alle Plätze im Bereich vor der Pivottransition inklusive ihrer Lokalitäten zu kopieren und die darin enthaltenen Kanten zu invertieren. Wenn nun im Fehlerfall alle Marken vom Teil des Netzes, welcher den positiven Lauf der Saga beschreibt, in den jeweils äquivalenten Platz in Kompensier-Teil der Saga

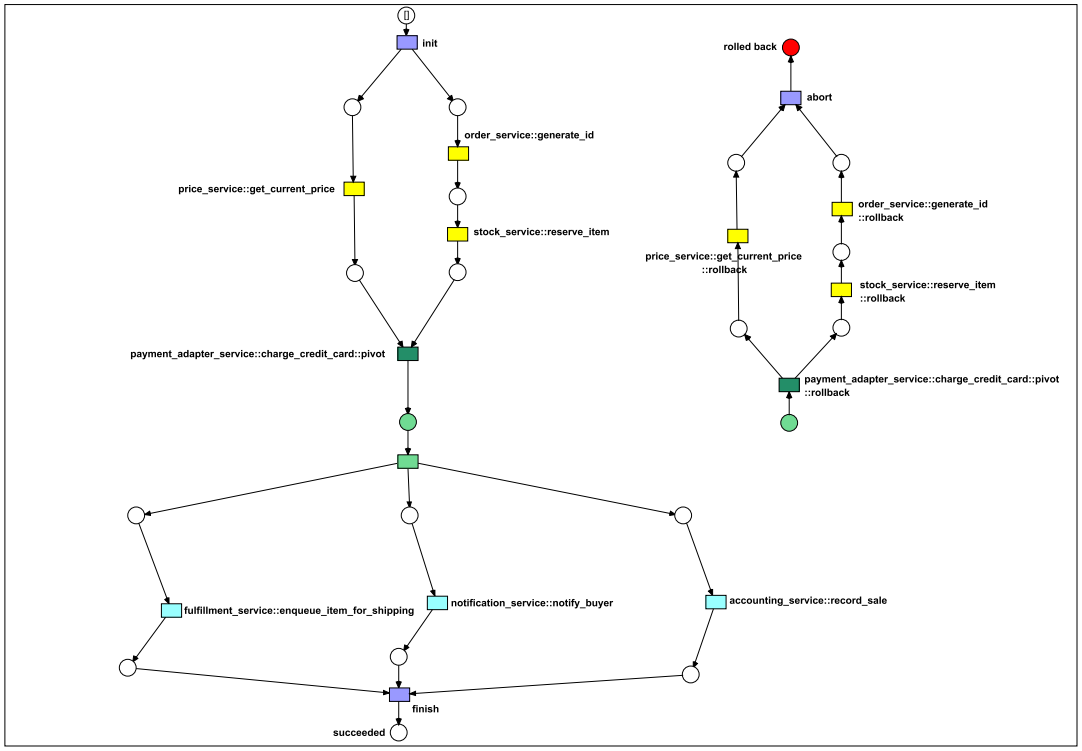

Abbildung 12.4.: Erzeugen der kompensierenden Transitionen

verschoben werden, kann über die invertierten Kanten genau die oben genannte Bedingung der Reihenfolge eingehalten werden.

Da der normale Ausführungsanteil des Netzes nicht verändert werden sollte, soll zunächst die Kopie des gesamten Teilnetzes vor der Pivottransition erfolgen. Danach kann das Invertieren aller Kanten innerhalb der Kopie erfolgen. Schlussendlich können alle Beschriftungen bestehend aus Service und angesprochener Transaktion in konkrete und ausführbare Kompensieroperationen umgewandelt werden. Die konkrete Anwendung auf das Beispiel ist in Abbildung 12.4 dargestellt.

12.4.3. Transitionsanschriften für Erfolgsfälle

Nach der Konstruktion des zuvor genannten Teils des Netzes oder wahlweise auch nebenläufig dazu können die Beschriftungen des Netzanteils, welcher den erfolgreichen Lauf der Saga abbildet, zu konkreten Operationen umgewandelt werden. Da in diesem Netzteil alle vorhandenen Transitionen der Ausführung einer Transaktion entsprechen, kann diese Operation auf alle Transitionen angewendet werden.

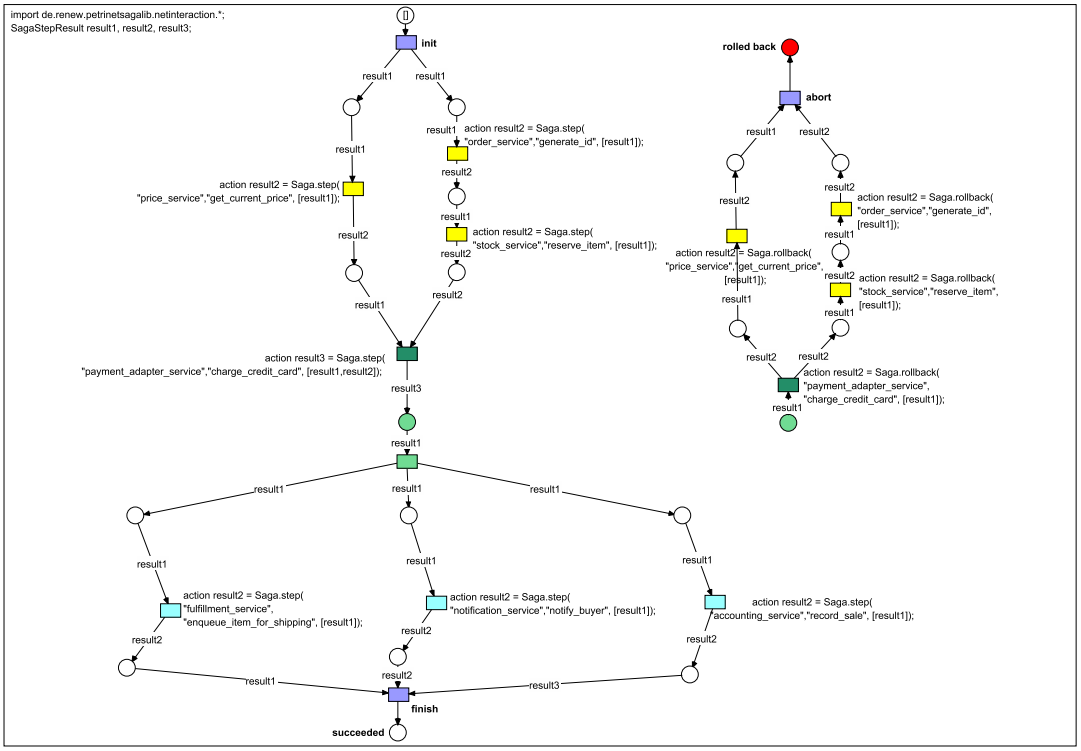

Abbildung 12.5.: Erzeugen der Transitionsanschriften

Eine Sonderregelung muss jedoch für die Pivottransition umgesetzt werden: Im Falle eines Fehlschlags der Pivottransaktion sollen die nachfolgenden Transitionen, welche den wiederholbaren Transaktionen zuzuordnen sind, nicht potenziell aktiviert werden. Dies begründet sich darin, dass zu einem späteren Zeitpunkt zu jeder Transition, welche eine wiederholbare Transaktion abbildet, eine entsprechende Wiederholungstransition generiert werden soll. Diese spezielle Wiederholungstransition bewirkt, dass bei einem Fehlschlag der ursprüngliche Zustand (Markierung der Lokalität der wiederholbaren Transaktion) vor Ausführung der Transition wiederhergestellt wird. Da die wiederholte Transaktion unter Umständen Informationen darüber erhalten muss, dass sie wiederholt ausgeführt wird, muss diese auch mit einem Ergebnis, welches potenziell ein Fehlschlag aussagt, umgehen können. Schlägt nun die Pivottransaktion fehl, so könnten die ersten wiederholbaren Transitionen nach dem Pivotelement nicht unterscheiden, ob es sich um einen lokalen Fehlschlag und eine Wiederholung handelt, oder ob der Fehlschlag vom Pivotelement ausging und die Saga abgebrochen werden muss.

Um diese Problematik zu umgehen, bietet es sich an, hinter der Pivottransition einen weiteren Platz und eine weitere Transition einzubinden, welche nur dann weiterschaltet, wenn die Pivottransaktion erfolgreich war. Diese Transition ent-

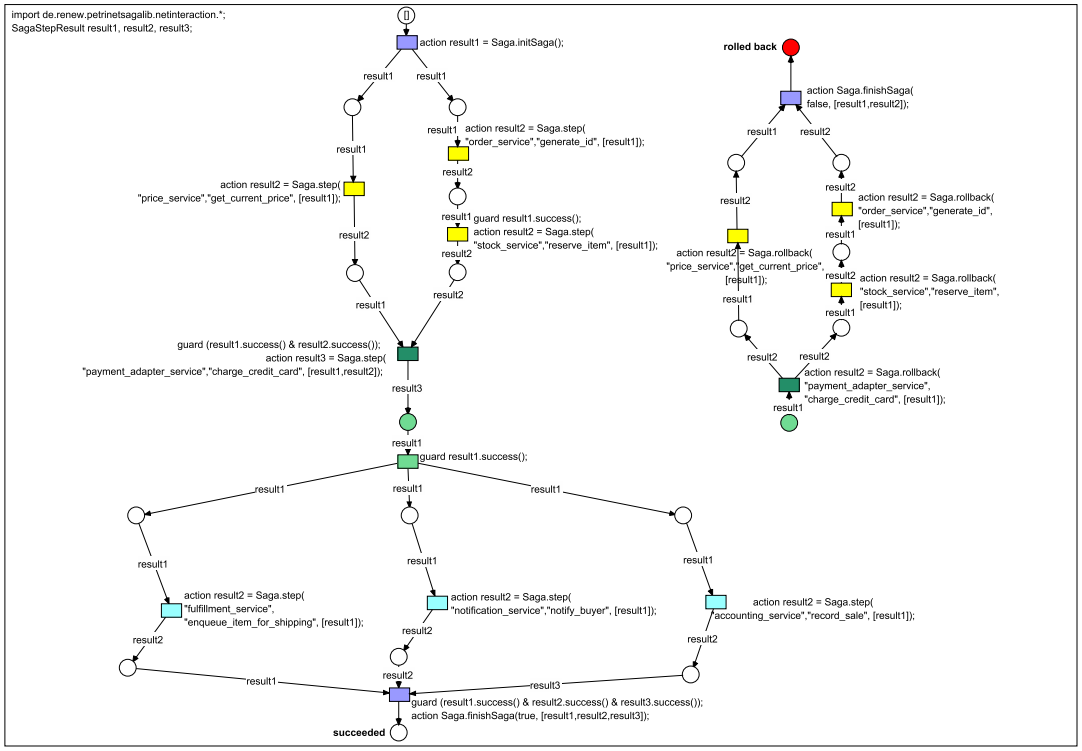

Abbildung 12.6.: Datenweitergabe und Erzeugung von Guards

hält keine produktive Inschrift und hat einzig den Grund das zuvor beschriebene Problem zu lösen.

12.4.4. Daten, Ergebnisse und Guards

In einem nächsten Schritt muss nun die Integration der Datenweitergabe erfolgen. Dabei kann in allen Netzanteilen sukzessive über alle Transitionen iteriert werden. Jede Ausführung einer Transaktion weist im Zweifel Abhängigkeiten zu allen Datensätzen der unmittelbar davorliegenden Ausführungen von Transaktionen auf. Darüber hinaus erzeugt die Ausführung der zugehörigen Transaktionen einen eigenen neuen Ergebnisdatensatz, welcher an alle nachfolgenden nebenläufigen Transaktionen weitergeleitet werden muss. Abbildung 12.5 zeigt die Umwandlung inklusive Transitionsanschriften und Datenobjekten.

Auf Basis der Referenznetzsemantik ergibt sich somit, dass bei n eingehenden Kanten und somit n verschiedenen Ausgangsdatensätzen insgesamt $n + 1$ Referenzen auf Ergebnismengen benötigt werden. Die eine weitere Referenz findet Verwendung, um den neu erzeugten Datensatz zu referenzieren. Ungeachtet der

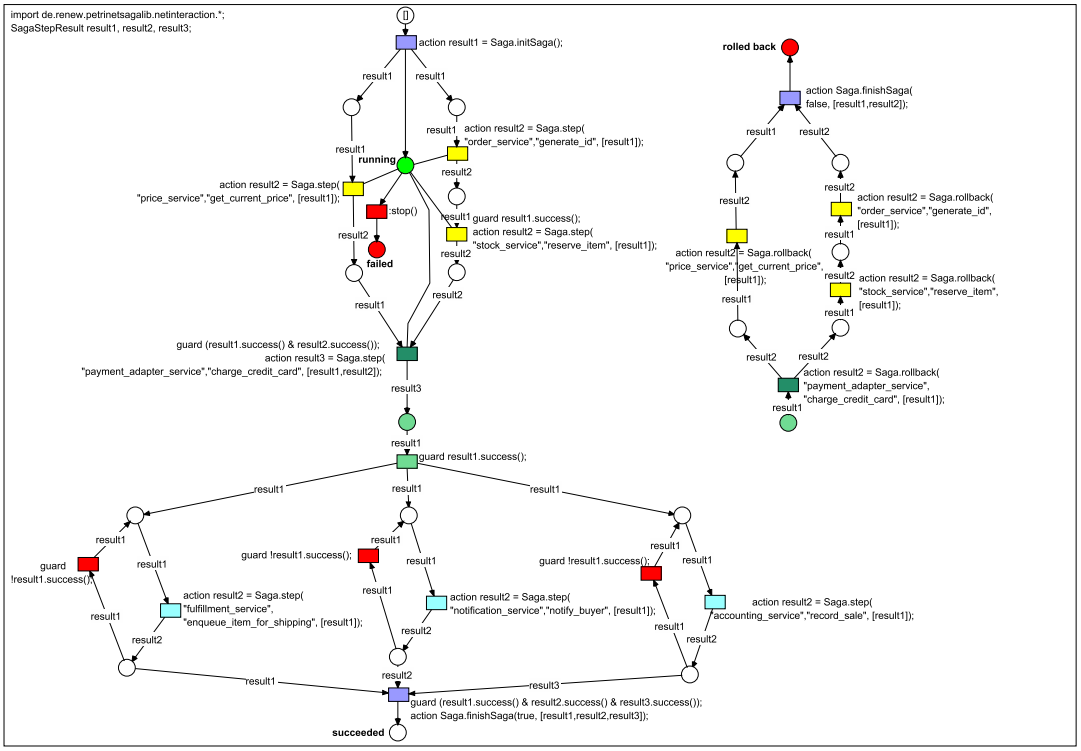

Abbildung 12.7.: Integration von Fehlschlagserkennung

Anzahl der ausgehenden Kanten kann stets dieselbe Referenz verwendet werden. In der Implementation muss beachtet werden, dass etwaige vom Verwender der Bibliothek vorgegebenen Aggregationsalgorithmen auf die Liste der vorigen Ergebnisdatensätze angewandt wird.

Diese verschiedenen Referenzen müssen nun sowohl als Kanteninschriften als auch in den Aufruf, welcher als Inschrift der Transitionen bereits gespeichert wurde, integriert werden. Dabei kann die Transitionsanschrift zunächst mit einem Template erzeugt werden, welches dann an dieser Stelle gefüllt wird.

Nachdem die Ergebnisreferenzen in das Netz integriert wurden, können entsprechende Guards erzeugt werden. Dabei sollte eine Saga nur dann fortgesetzt werden, wenn es nirgendwo einen Fehlschlag gab. Dies bedeutet, dass die Erfolgsindikatoren der Ergebnisobjekte als Konjunktion einer aussagenlogischen Auswertung unterzogen werden sollten. Die so erzeugten Guards können zu allen Transitionsanschriften hinzugefügt werden und sind in Abbildung 12.6 dargestellt.

12.4.5. Fehlschlagserkennung

Mit der Fertigstellung der grundlegenden Struktur und den Inschriften, von sowohl den erfolgreichen Verläufen der Saga als auch dem Kompensieren von einem beliebigen erreichbaren Zustand der Saga aus, verbleibt nun lediglich die Fehlerbehandlung. Als letzte Operation wurden zuvor bereits entsprechende Guards hinzugefügt, welche erfolgreiche Ergebnisse interpretieren können. Um nun auch fehlerhafte Transaktionsausgänge verarbeiten zu können, müssen entsprechende Transitionen hinzugefügt werden.

Im Falle der wiederholbaren Transaktionen fällt diese Fehlerbehandlung vergleichsweise simpel aus. Hierzu genügt es, eine Transition neben die betrachtete Transition zu legen, welche mit der gleichen Lokalität verbunden ist, jedoch invertierte Kanten aufweist. Ein Guard garantiert, dass diese Wiederholungstransition nur dann schalten kann, wenn die zugehörige Transaktion fehlgeschlagen ist und somit dieser Fehlschlag in der Ergebnismenge verzeichnet ist. Die Wiederholungstransition stellt die Markierung der Lokalität der wiederholbaren Transitionen wieder her, sodass diese ein weiteres Mal schalten kann.

Komplizierter gestaltet es sich jedoch im Fall der kompensierbaren Transaktion und der Pivottransaktion. Im Falle der kompensierbaren Transaktion muss zusätzlich die Erkennung von weiteren Fehlschlägen in nebenläufigen Transaktionen bedacht werden. Da die Pivottransition per Definition keine nebenläufigen Transitionen besitzt, entfällt diese Notwendigkeit an dieser Stelle.

Zunächst soll der Fall eines lokalen Fehlschlags betrachtet werden. In diesem Fall kann der entsprechende Fehlschlag aus dem Ergebnisobjekt ausgewertet werden und eine zu konstruierende Transition bewegt das Ergebnisobjekt in den äquivalenten Platz auf der Seite der Petrinetz-Saga, welche für das Kompensieren verantwortlich ist. So kann dem Service, bei dem die Transaktion fehlgeschlagen ist, die Möglichkeit eingeräumt werden, etwaige Aufräumarbeiten im Kontext der Saga anzustoßen. Im Normalfall sollte bei einer sauberen Implementation hier jedoch keine Arbeit anfallen, da die fehlgeschlagene Transaktion bereits automatisch für ein Zurückrollen seitens der Datenbank gesorgt hat und keine Kompensation notwendig ist.

Findet ein Fehlschlag an einer Stelle innerhalb der Saga statt, so sollten (global) keine weiteren Transaktionsschritte innerhalb dieser Saga eingeleitet werden. Zu diesem Zweck muss eine globale Fehlschlagserkennung zusätzlich und neben der lokalen Fehlschlagserkennung integriert werden. Dazu bietet es sich an, einen einfachen zentralen Platz in die Referenznetzdarstellung der Petrinetz-Saga zu integrieren. Alle Transitionen auf der Seite, die den normalen Durchlauf der Saga repräsentiert, müssen mittels Testkante überprüfen, ob in diesem Platz eine Marke vorhanden ist. Diese Marke symbolisiert, dass sich bisher global kein Fehl-

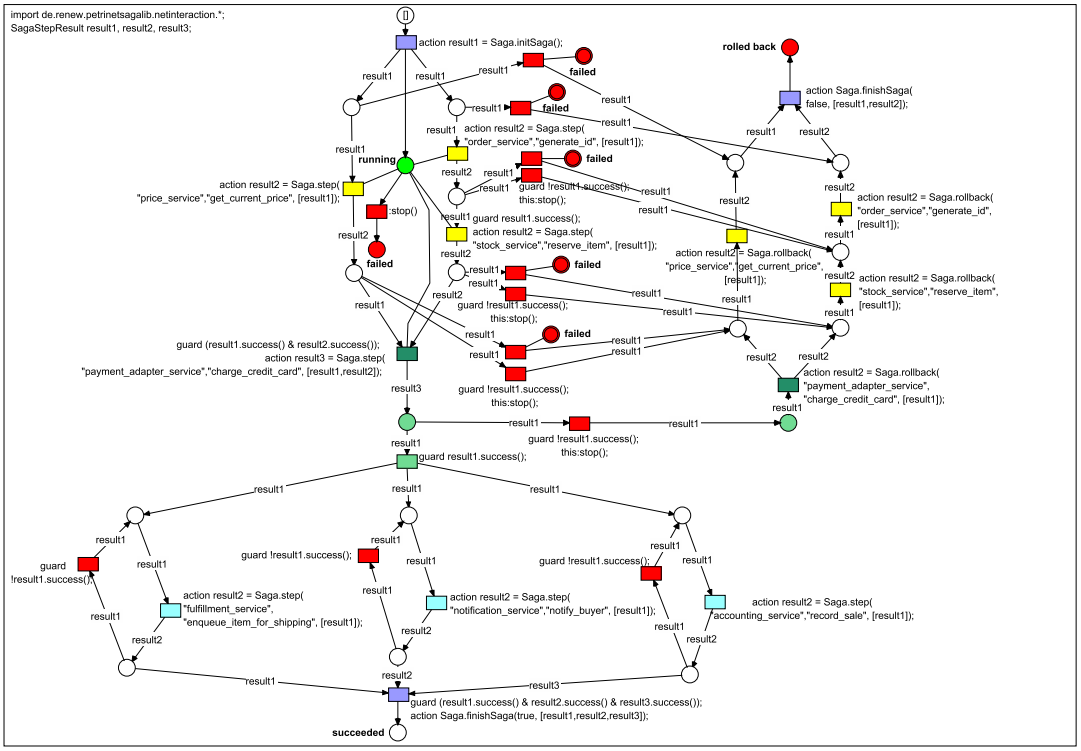

Abbildung 12.8.: Transformierte Petrinetz-Saga

schlag ereignet hat. Sie sollte bei der Initialisierung der Saga generiert werden. Die Konstruktion dieses Platzes ist in Abbildung 12.7 dargestellt.

Findet nun ein lokaler Fehlschlag statt, muss gleichzeitig diese Marke aus dem Platz abgezogen werden und in einen weiteren Platz gelegt werden, indem sie das Gegenteil, nämlich das Auftreten wenigstens eines Fehlschlags symbolisiert. Die Transitionen für globale Fehlschläge können dann mittels Testkante prüfen, ob in diesem Platz eine Marke vorhanden ist und dann ungeachtet des lokalen Ergebnisobjekts eine Überführung in den jeweiligen äquivalenten Platz auf der Kompensier-Seite des Netzes einleiten.

Mit diesen Schritten kann nun eine Referneznetzdarstellung der Saga erzielt werden, welche nach üblicher Referenznetzsemantik simulierbar ist und die Sagaoperationen ausführen kann. Die finale Referenznetzvariante der Beispiel-Saga ist in Abbildung 12.8 aufgeführt.

12.5. Ablauf einer Petrinetz-Saga

Im Folgenden wird nun eine detaillierte Ablaufbeschreibung der Protokolle innerhalb von Petrinetz-Sagas gegeben. Dazu dienen die Interaktionsprotokolle in den beiden Abbildungen 12.9 und 12.10. Die erste Abbildung beschreibt den Ablauf bei der Initialisierung einer Saga. Dabei wird angenommen, dass eine nutzende (verteilte) Anwendung des Petrinetz-Saga-Systems eine Anfrage für die Ausführung einer Petrinetz-Saga an das System stellt. Das System führt dabei die Transformation nach dem im letzten Abschnitt beschriebenen Ablauf aus. Im Gegensatz zur Initialisierung nimmt das zweite Diagramm an, dass die Initialisierung bereits abgelaufen ist und zeigt sodann den Ablauf eines einzelnen Feuervorgangs inklusive Ausführung der zugehörigen Transaktion innerhalb einer referenznetzbasierten Petrinetz-Saga.

Beide Abläufe beschreiben im Wesentlichen Abläufe auf dem *Orchestrator* der Saga.

12.5.1. Initialisierung einer Petrinetz-Saga

In diesem Abschnitt wird der vorgeschlagene Ablauf zum Initialisieren der Abarbeitung einer Petrinetz-Saga beschrieben. Der Ablauf nimmt dabei die Verwendung des Referenznetzsimulators Renew an. Entlang der Architektur von Renew liegt es nahe, die Abarbeitung von Petrinetz-Sagas als Plugin zu realisieren. Als Format für die Persistenz von Petrinetz-Sagas wurde hier PNML[7] gewählt, da es universell einsetzbar ist und mit vielen Petrinetzsimulatoren und -editoren kompatibel ist. PNML Dialekte existieren sowohl für P/T Netze als auch für Referenznetze.

Die Initialisierung tangiert somit ein Petrinetz-Saga-Plugin, entsprechende Routinen zur Transformation, entsprechende PNML Parser und Generatoren sowie den Rest des Renew Ökosystems für die Simulation. Wie eingehend erläutert ist der Ablauf der Initialisierung anschaulich in Abbildung 12.9 dargestellt.

Zunächst wird ein Auftrag durch einen Nutzer erstellt, welcher die Initialisierung anstößt. Mitgeliefert wird dabei die P/T-Netz Version der Saga, welche dann aus dem PNML Format ausgelesen werden kann. PNML basiert auf XML, sodass entsprechende Bibliotheken zum Einsatz kommen können, falls kein direkter PNML Parser verfügbar ist. Danach übergibt das Petrinetz-Saga-Plugin die P/T-Netz Daten an den Transformator. Dieser führt sodann alle im Abschnitt 12.4 zur Transformation beschriebenen Schritte durch. Nach der Transformation kann ein Generator die PNML Repräsentation der Referenznetzvariante erzeugen, welche

[7] http://www.pnml.org/ - Zuletzt abgerufen am 30.07.2021

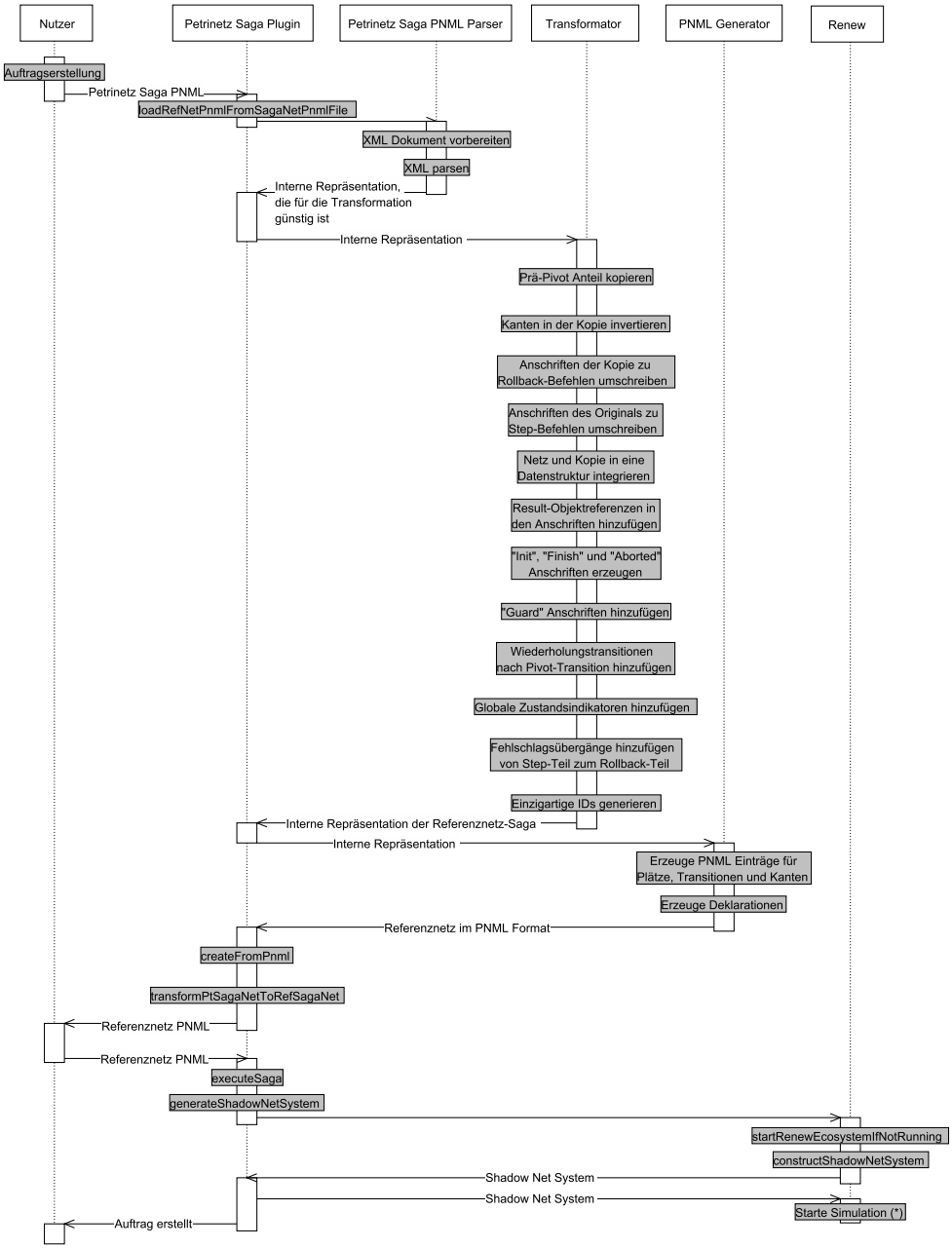

Abbildung 12.9.: Ablauf der Initialisierung einer Petrinetz-Saga

dann zur Simulation erneut geparsed wird. Der Grund für diesen Zwischenschritt besteht darin, dass auf diese Weise die Möglichkeit angeboten werden kann, beliebige Sagas auf Referenznetzbasis auch direkt verwenden zu können. Diese müssen nach ihrer Konstruktion als PNML-Referenznetz gespeichert werden und können dann an dieser Stelle anstatt dem transformierten Netz eingesetzt werden. Nachdem die Saga auf Referenznetzbasis eingesetzt wurde, startet die Simulation und damit die Initialisierung des Renew Simulationskerns mit einer einzelnen Netzinstanz des Petrinetz-Saga-Referenznetzes. Im Simulationskern wird das Netz zunächst in eine Schattennetzdarstellung kompiliert und diese dann simuliert.

12.5.2. Ausführung eines Feuervorgangs innerhalb der Saga

Die eigentliche Ausführung der Petrinetz-Saga basiert auf der Simulation der Referenznetzdarstellung. Dabei werden immer wieder einzelne Transitionen gefeuert, welche bestimmten Transaktionen in Services entsprechen. Den vorgeschlagenen Ablauf eines einzelnen Feuervorgangs zeigt Abbildung 12.10.

Zum Einsatz kommen hierbei der Renew Simulator, das Petrinetz-Saga-Plugin, welchem auch ein Saga Manager angehört, ein spezifisches Saga Framework mit zugehöriger Datenbank und Saga Framework, der Nachrichtenbroker und der entfernte Service, auf welchem die Transaktion ausgeführt wird. Das spezifische Saga Framework ist an dieser Stelle noch allgemein gehalten, da verschiedene Implementationen bereitstehen bzw. denkbar sind.

Nach dem Beginn eines Feuervorgangs werden zunächst die Daten aus den vorangegangenen Aufrufen aggregiert. Die Aggregation erfolgt anwendungsfallbezogen und vorgegeben durch den Entwickler bzw. Modellierer. Der Saga Manager entscheidet, ob es sich um einen lokalen Schritt auf dem Koordinator handelt[8] oder aber um einen entfernten Service. Mittels Datenbank und Service transformiert das eingesetzte Saga Framework die Anfrage in ein Event, welches an den Nachrichtenbroker übermittelt wird. Durch Polling[9] erhält der entfernte Service die Informationen bezüglich der auszuführenden Transaktion, welche sodann ausgeführt wird. Das Ergebnis wird an den Nachrichtenbroker zurückgeleitet, wo es vom Saga Framework beim Orchestrator ausgelesen werden kann. Dieser wertet die Ergebnisse aus, konstruiert daraus entsprechende Marken und schließt den Feuervorgang ab, indem diese Marken in den Nachbereich der Transition gelegt werden.

[8] Lokale Schritte auf dem Koordinator sind ohne Verteilung realisierbar und wurden daher nicht gesondert behandelt.

[9] Entsprechend des Microservice Patterns werden keine smart pipes eingesetzt, daher erfolgt auch keine aktive Weiterleitung

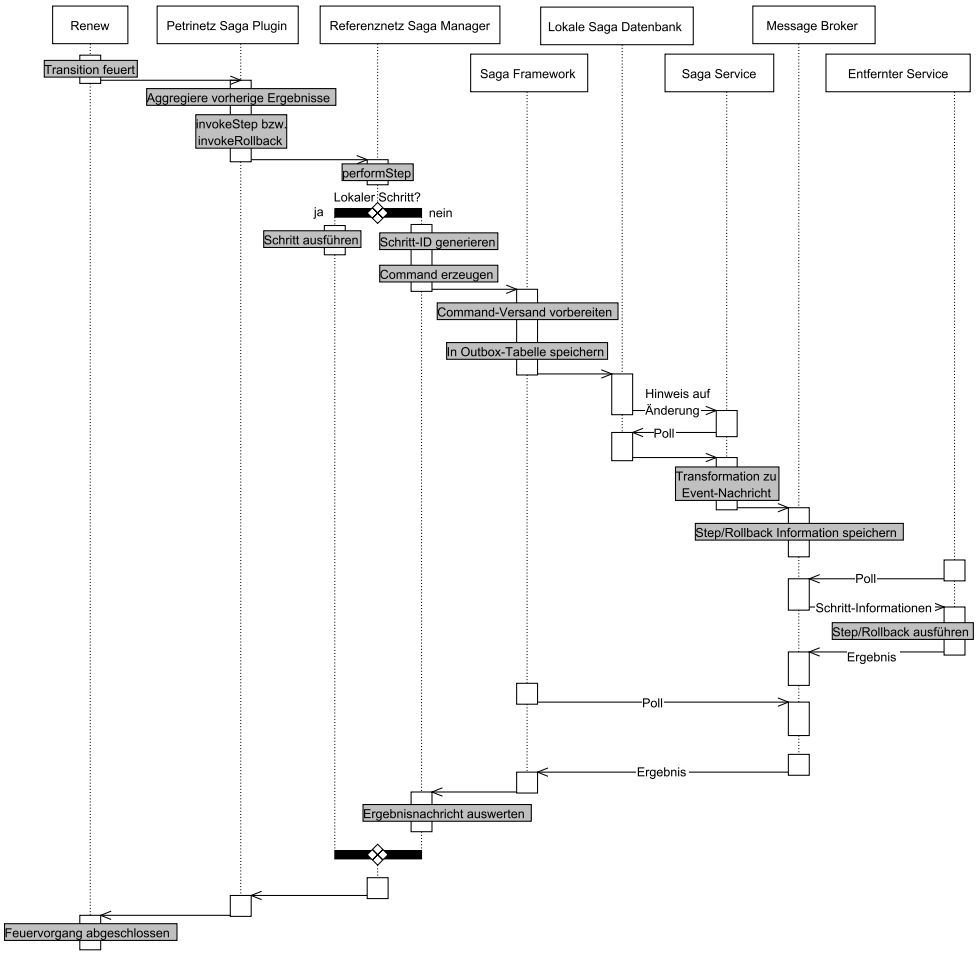

Abbildung 12.10.: Ausführung eines einzelnen Feuervorgangs einer Transition

12.6. Weitere Aspekte von Petrinetz-Sagas

Dieser Abschnitt adressiert alle verbleibenden Aspekte bezüglich der Einführung von Petrinetz-Sagas. Durch die Anteile der Arbeit hindurch wurde stets ein Entkoppelungs- und Dezentralisierungsgedanke verfolgt. Die Motivation dabei lag darin, dass zentralisierte Systeme anfälliger für Ausfälle sind und sich in einem stark belasteten System zum Bottleneck entwickeln können. Unter diesem Gesichtspunkt erscheint die Einführung eines zentralen Orchestrators für die Abarbeitung von Sagas kontraintuitiv. Eins der Ziele dieses Abschnitts ist die Erörterung, wie dennoch eine Entkoppelung erreicht werden kann.

Neben der Rolle des Orchestrators sollten Abstürze und Ausfälle zumindest kurz thematisiert werden. Diese Aspekte sind wichtige Faktoren in jedem verteilten System. Als weiteren Aspekt garantieren Sagas keine Isolation im Sinne der ACID Eigenschaften von Transaktionen. Folglich sollte auf den Isolationsaspekt noch einmal gesondert eingegangen werden. Darüber hinaus wurde die Thematik bezüglich Verifikation in Petrinetz-Sagas angesprochen. Es ist nicht direkt offensichtlich, welche Möglichkeiten hierbei bestehen, sodass der Ausblick in diese Richtung zumindest angerissen werden sollte.

12.6.1. Dezentrale Orchestration

Alle bisherigen Ausführungen zu der Konstruktion von herkömmlichen oder petrinetzbasierten Sagas haben stets den Einsatz eines Orchestrators angenommen. Nun erzeugt aber die Existenz eines zentralen Orchestrators genau eine Situation, welche ursprünglich vermieden werden sollte. Bei der Einführung von Sagas im Grundlagenkapitel in Abschnitt 2.5.2 wurde ebenfalls die Implementation mittels Choreografie vorgestellt. Es wurde bereits motiviert, dass Choreographie-basierte Sagas kein geeigneter Ansatz für den Petrinetz Kontext darstellen. Dieser Umstand wird an dieser Stelle noch einmal weiterführend erläutert.

Choreografiebasierte Sagas weisen eine Menge an Problemen auf: Durch das Fehlen einer Kontrollinstanz müssen einzelne Services Informationen über zumindest den oder die Nachfolger besitzen und so die Saga fortführen. Wie auch im Rahmen der Einführung erläutert, erhöht dies die Kopplung innerhalb des Systems. Dies ist eine Eigenschaft, die in einem skalierenden System niemals wünschenswert ist, da sie die Komplexität erhöht und Skalierung erschwert. Darüber hinaus müsste die Struktur als Petrinetz nur als Grundlage dienen, um entsprechende Funktionalität in die jeweiligen Services zu implementieren. Ein einfaches Ausführen der Petrinetz-Saga, wie sie im Rahmen dieses Kapitels vorgestellt und skizziert wurde, wäre bei der Choreografie-basierten Lösung nicht einfach umsetzbar. Viel mehr müssten Einzelimplementationen in jedem Service auf Basis jeder einzelnen

Transition in der Petrinetz-Saga erfolgen. Eine etwaige Datenaggregation müsste darüber hinaus bei jedem Service vor Ausführung der lokalen Transaktion erfolgen oder aber an einer dedizierten anderen Stelle.

Im Kontext der Anwendung auf verteilte synchrone Kanäle ergibt sich ein weiterer und noch gravierenderer Punkt: Bei der Ausführung von einem synchronen Kanal sind die jeweiligen Synchronisationspartner erst im Moment der Ausführung bekannt. Das Deployment einer choreografiebasierten Saga umfasst jedoch die Implementation der jeweiligen Nachfolger im Code des jeweiligen Services. Da diese Nachfolger zur Kompilierzeit noch nicht bekannt sind, kann dies nicht gewährleistet werden. Eine globale Fehlschlagserkennung, wie sie durch Nebenläufigkeit notwendig wird (vgl. Abschnitt 12.2.2), ist ebenfalls deutlich schwerer zu realisieren ohne zentrale Koordinationsinstanz. Um choreografiebasierte Sagas einsetzen zu können, wären also weitere Anpassungen notwendig.

Tatsächlich sind mögliche Lösungen sowohl für choreografiebasierte Ansätze als auch den bisher vorgestellten auch orchestrationsbasierten Ansatz denkbar. Beide Lösungen sollen im Folgenden skizziert werden.

Als eine der angesprochenen »weiteren Anpassungen« könnte eine choreografiebasierte Petrinetz-Saga bei Feststehen der Synchronisationspartner generiert werden. Bei Ausführung und Weitergabe an die nächste lokale Transaktion könnte die aktuelle Netzinstanz der Saga mitgeschickt werden. Wenn der empfangende Service seine eigene Identität und die referenzierte lokale Transaktion kennt, kann dieser die aktuelle Position in der geschickten Saga erkennen. Lediglich bei mehrfacher Ausführung der gleichen Transaktion auf dem gleichen Service müsste eine Art Zeiger (beispielsweise durch eine entsprechende Markierung) auf den entsprechenden Stand mitgesendet werden. Auf dieser Basis könnte die Übermittlung an den oder die nächsten Teilnehmer der Saga statisch so implementiert werden, dass diese stets aus dem Netz mit aktuellem Zustand abgeleitet wird.

Eine Umsetzung von dieser Art würde erfordern, dass entweder ein Interpreter das Saga-Netz dynamisch interpretieren müsste, oder aber, dass eine neue Referenznetzsimulation mit dem gelieferten Netz und einer entsprechenden passenden initialen Markierung gestartet wird. Diese Simulation bzw. Interpretation würde lediglich einen einzigen Schritt ausführen, danach terminieren und die weitere Abarbeitung an weitere Services senden. Eine Ausnahme könnte integriert werden, falls mehrere lokale Transaktionen nacheinander auf demselben Service ausgeführt werden müssen. Darüber hinaus müsste eine Form der Serialisierung von Netzinstanzen gewährleistet sein. Bisher arbeitet beispielsweise das Distribute Plugin jedoch nur mit dem Konzept von Distribute Netzinstanzen, die auf Basis von Stubs zwar lokal existieren, deren Implementation jedoch an einem entfernten Ort liegt. Eine Form der Übertragung von Netzinstanzen ist dort nicht implementiert. Als weiterer Aspekt gilt es zu betrachten, dass bei dieser Art der Ausführung gezielt und sehr regelmäßig der Kontext der Referenznetzsimulation

verlassen wird, indem stets nur ein oder wenige Schritte simuliert werden und danach das Netz verpackt und weitergeschickt wird.

Während dieser Ansatz vollständig ohne Koordinator auskommt, ist er doch an vielen Stellen offensichtlich relativ ineffizient.

Als zweiten Lösungsansatz werden orchestrationsbasierte Sagas betrachtet. Da eine Saga stets beim Feuern eines verteilten synchronen Kanals ins Leben gerufen werden soll, besteht für den Kontext der Saga eine eindeutige Zuordnung zu bestimmten Teilen des Gesamtsystems. Dieser Umstand kann ausgenutzt werden, damit beim Feuervorgang des verteilten synchronen Kanals ein lokaler Orchestrator hinzugeschaltet werden kann. Im Normalfall geht die Ausführung einer Synchronisation stets von einem Downlink aus, da dieser eine konkrete Netzinstanz referenziert. Es sei an dieser Stelle noch einmal darauf hingewiesen, dass es sich dabei nicht auch unmittelbar um die Richtung des Informationsflusses handeln muss.

Die Architektur des Mushu-Plattformmanagements sieht einen Ablaufmanager für komplexe Kommunikation vor. Daher liegt es nahe, die Fähigkeit Sagas zu koordinieren an dieser Stelle im Mushu-Plattformmanagement zu verorten. Problematisch ist dabei jedoch, dass die Kommunikation über synchrone Kanäle wie beschrieben durch einen Downlink initiiert wird und daher im Rahmen der Realiserungskonzepte der einzelnen Bestandteile der Mushu-Architektur zunächst Plattform-lokal ablaufen würde. Sofern Kommunikationspartner und Ablauf in Form einer Petrinetz-Saga definiert sind, kann diese allerdings unabhängig von den Teilen der Simulation betrieben werden, welche über den verteilten synchronen Kanal kommunizieren wollen. Somit handelt es sich dabei um eine nebenläufige Simulation mehrerer Referenznetzsimulationen, wie sie bereits in Abschnitt 11.4.1 im Realisierungskonzept zur Mushu-Plattform diskutiert wurde.

Eine der dort vorgestellten Lösungsmöglichkeiten war der Einsatz von »Prozesse(n) auf einer physikalischen Recheneinheit«. Dabei ist der Einsatz innerhalb einer Plattform nicht unbedingt zielführend, ist für das hier vorliegende Problem jedoch sehr passend, da ohnehin eine Entkoppelung in separate Entitäten angestrebt wird. Wird die Methode zusätzlich mit dem Ansatz der Containerisierung verbunden, wie es im Rahmen des Realisierungskonzepts zum Mushu-Plattformmanagement bereits vorgestellt wurde, wird eine portable Kapselung eines Saga Managers erreicht. Dies entspricht dann der Grundidee des Ablaufmanagers im Mushu-Plattformmanagement.

Falls durch den Einsatz separater Prozesse wider Erwarten ein beträchtlicher Overhead an Startzeit der Prozesse zur Abarbeitung der Saga entsteht, ist eine weitere Variante denkbar. Somit könnten spezielle Saga-Hostsysteme konstruiert werden, welche dem »Simulationen in einem Simulator«-Ansatz der nebenläufigen Simulation mehrerer Referenznetzsimulationen folgen. Diese Referenznetzsi-

mulatoren besitzen einen sehr eingegrenzten, spezifischen Anwendungshorizont – nämlich die Simulation von Petrinetz-Sagas. Dadurch könnten die Problematiken von statischen Datenstrukturen, wie sie im Abschnitt zum »Simulationen in einem Simulator«-Ansatz beschrieben wurden, gezielt umgangen werden. Diese Umsetzung hätte den Vorteil, dass eine Menge an Saga-Exekutoren im Plattformmanagement vorgehalten werden könnte, welche sodann nicht extra gestartet werden müssten.

Wenn stets dynamisch ein Orchestrator lokal zur Ausführung gestartet wird (bzw. mehrere bereitstehen), entfällt die Einführung eines zentralisierten Koordinators und damit die Einführung eines Single points of failure. Vielmehr kann dieser Ansatz so betrachtet werden, dass eine Art Taskforce zur Abarbeitung des Feuervorgangs zusammengestellt wird. Darin spielt der lokale Orchestrator die Rolle des Verwalters und alle Teilnehmer an der Saga die Mitglieder. Aus einer anderen Sicht kann auch argumentiert werden, dass in diesem Fall die Abarbeitung der Saga durch das Plattformmanagement bereitgestellt wird, jedoch in Form der zusammengestellten Taskforce.

Zuletzt kann im Kontext der Berechnung verteilter synchroner Kanäle argumentiert werden, dass eine Referenznetzsimulation, welche zur Abarbeitung einer Saga eingesetzt werden soll, selbst wiederum ein verteiltes System darstellt. Da diese Referenznetzsimulation jedoch überhaupt erst eingesetzt werden soll, um die verteilte Ausführung von Referenznetzsimulation zu ermöglichen, würde an dieser Stelle ein zyklisches bzw. ein Henne-Ei-Problem entstehen. Dass dieses Problem nicht praktisch auftritt, liegt darin begründet, dass die Referenznetzsimulation nur zur Vorgabe der Struktur der Saga eingesetzt wird, nicht aber innerhalb der lokalen Transaktionen, aus der sich die Saga zusammensetzt. Dadurch und durch den Orchestrator-Ansatz entsteht die Begebenheit, dass die Referenznetzsimulation zur Ausführung der Saga ein rein lokales Netz ist, bei dem die verteilte Ausführung in der Implementation der einzelnen Transitionen gekapselt ist. Diese weisen wiederum dadurch einen deutlich eingeschränkteren Funktionsumfang auf und sind nicht auf die aufwändige Implementation der verteilten komplexen Kommunikation angewiesen.

12.6.2. Abstürze und Ausfälle

Petrinetz-Sagas sind in erster Linie Sagas. Sie sind konzipiert, um in einer Umgebung, bei der die Verfügbarkeit einzelner Netzwerkressourcen durch etwaige Netzwerkpartitionierung nicht immer gewährleistet werden kann, zuverlässig zu operieren und das Gesamtsystem früher oder später in einen konsistenten Zustand zu überführen. Darüber hinaus können einzelne Netzwerkkomponenten und damit Teilnehmer der Saga auch gänzlich abstürzen. Dabei handelt es sich um eine

schwerwiegendere Version der vorübergehenden Trennung vom Netzwerk, da unter Umständen auch Zustände verloren gehen können.

Zwar bieten zustandslose Anwendungen in dieser Hinsicht ideale Voraussetzungen, da sie, wie der Name suggeriert, keinen Zustand besitzen, der verloren gehen könnte. Dennoch wurde bereits in Abschnitt 10.2.5 dargelegt, warum eine vollständig zustandslose Implementation der Mushu-Plattform nur schwer umzusetzen ist. Auch andere Komponenten können nicht immer mit dem Luxus der Zustandslosigkeit aufwarten. Folglich sollte die Umsetzung einer Petrinetz-Saga auf derartige Ausfälle vorbereitet sein und den ausgefallenen Knoten die benötigten Daten wiederholt bereitstellen können, sofern dies erforderlich ist, um einen aktuellen Schritt abzuschließen.

Abstürze von Sagateilnehmern

Es soll als Voraussetzung formuliert werden, dass Teilnehmer einer Saga nach Abschluss eines Schrittes die etwaigen lokalen Änderungen absturzsicher persistiert haben müssen. Dies gilt mit der Ausnahme von solchen Daten, welche nur der lokalen Information dienen, jedoch keinen Einfluss auf den etwaigen Rückgabedatensatz des Schrittes haben. Würde diese Einschränkung nicht formuliert werden, wäre es nicht von außen ersichtlich, wenn lokaler Datenverlust auftritt. Die Konvergenz des Gesamtsystems in einen konsistenten Zustand durch eine Saga könnte nicht mehr garantiert werden.

In der Umsetzung in Referenznetzsimulationen sollte diese Persistenz ebenfalls gewährleistet werden. Andernfalls könnte die Korrektheit der Berechnungen nur im Kontext von Netzwerkpartitionen, nicht aber bei umfassenden Ausfällen gewährleistet werden. Dieser Abschnitt adressiert explizit die Überlegungen im Kontext von Petrinetz-Sagas. Die generelle Umsetzung der Wiederherstellbarkeit von Referenznetzsimulationen ist durch die Nebenläufigkeit und Komplexität nicht trivial. Dies gilt insbesondere, wenn sowohl Nebenläufigkeit als auch Performance der Simulation nicht substanziell ausgebremst werden sollen. Einzelheiten zu diesem Thema wurden bereits in Abschnitt 11.6.6 zu True-Concurrency Checkpoints erörtert.

Abstürze des Orchestrators

Ein Spezialfall entsteht, wenn anstatt einem Teilnehmer der Saga der Orchestrator selbst ausfällt. In diesem Fall müsste der Zustand des Orchestrators wiederhergestellt werden. Der Orchestrator ist im Rahmen der vorgeschlagenen Realisierung ebenfalls eine Referenznetzsimulation.

Aus diesem Grund gelten für ihn die gleichen Überlegungen, welche auch bei der eigentlichen Referenznetzsimulation selbst gelten. Eine Persistenz mittels Checkpoint wäre eine umfassende Lösung für dieses Problem. Speziell durch die Eigenschaft der Orchestrierung von Sagas entstehen allerdings weitere denkbare Lösungsmöglichkeiten. Der Einsatzzweck ist sehr viel spezieller als allgemeine Referenznetzsimulation, da beispielsweise nur eine Netzinstanz zurzeit besteht.

Es wäre möglich, für die Saga einen eindeutigen Bezeichner für jeden Schritt zu generieren. Beim Abarbeiten der Saga könnte der Simulator dann das erfolgreiche Abarbeiten des jeweiligen Schritts persistieren, zum Beispiel in einem Log. Stürzt der Orchestrator ab, so kann er die Saga von vorne erneut ausführen und dabei bereits ausgeführte Schritte anhand des Logs überspringen. Dieser Ansatz ist allerdings nur so lange funktional, wie kein Nicht-Determinismus innerhalb der Saga zulässig ist. Zyklen und ein Abwenden von Kausalnetzen als P/T-Netz-Grundlage in weiteren Iterationen des Formalismus würde den Einsatz von oben beschriebenen allgemeineren Checkpoints notwendig machen.

12.6.3. Verifikation nebenläufiger Sagas

P/T-Netz basierte Sagas sind in dem aktuellen Entwurf des Formalismus Kausalnetze und korrekte Workflownetze. Diese Eigenschaften machen den Formalismus sehr einfach und erlauben den umfassenden Einsatz von Verifikationswerkzeugen. Die Problematik dabei besteht jedoch darin, dass viele Verifikationsmethoden die Voraussetzung besitzen, dass das entsprechende Netz auch mit einer gängigen Petrinetzsemantik betrachtet wird.

Bei der Simulation von Petrinetz-Sagas wird jedoch zunächst ein entsprechendes Referenznetz erzeugt, welches dann mit Referenznetzsemantik simuliert werden kann. Eine direkte Simulation des P/T-Netzes würde jedoch nicht gänzlich falsch sein. Dabei würde im Wesentlichen die Möglichkeit, dass Sagas fehlschlagen können, nicht abgebildet werden. Ist dieser Aspekt nicht im Rahmen der Verifikation von Interesse, können gängige Verifikationsmethoden eingesetzt werden.

Werden bei der Verifikation Fehlschläge ignoriert, kann zu Recht die Frage formuliert werden, welche Aussagekraft derartige formale Überprüfungen besitzen. Daher ist es eine umfassendere Lösung, entsprechende Verifikationsansätze direkt am generierten Referenznetz anzusetzen. Die Verifikation von Referenznetzen ist nicht trivial und ist Gegenstand gegenwärtiger Untersuchungen. Einen ersten Schritt in die Richtung unternehmen die Arbeiten (Willrodt, 2019; Willrodt und Moldt, 2019; Willrodt, Moldt und M. Simon, 2020) zum Werkzeug MoMoC. Auf der Basis von MoMoC und einem der Prototypen für das Plattformmanagement aus dem Rahmen dieser Arbeit integriert die Arbeit (Engelhardt, 2020) diese beiden Implementationen zum Werkzeug DistributedAnalysis.

Da wie beschrieben die Abarbeitung von Petrinetz-Sagas ein Spezialfall einer Referenznetzsimulation darstellt, ist es denkbar, in zukünftigen Untersuchungen spezielle Verifikationsmethoden an diesem Anwendungsfall anzupassen.

12.6.4. Isolation

Sagas garantieren Atomizität, Konsistenz und Dauerhaftigkeit (Transaktionseigenschaft A, C und D sind erfüllt). Sagas garantieren dabei jedoch keine Isolation der Operationen. Folglich obliegt die Implementation von Isolation der Anwendungsebene und dem Entwickler, welcher ein Saga Framework einsetzt. In der allgemeinen Fassung von Petrinetz-Sagas liegt dieser Umstand außerhalb des Betrachtungsbereiches und muss ebenfalls vom Entwickler, welcher die konkrete Petrinetz-Saga entwickelt, beachtet werden.

Aus diesem Grund ist die Diskussion über Isolation nur sinnvoll innerhalb des hier intendierten Anwendungszweckes, nämlich der verteilten Schaltung von synchronen Kanälen. Seiteneffekte durch fehlende Isolation entstehen dann, wenn Änderungen durch andere Transaktionen gelesen bzw. überschrieben werden, welche später wieder zurückgerollt oder erneut überschrieben werden. Dabei können alle gängigen Anomalien in Datenbanken auftreten, wie beispielsweise Dirty Read, Lost Update, usw. Wird jedoch der Anwendungsfall betrachtet, so können derartige Seiteneffekte lediglich durch »action« Inschriften[10] an Transitionen und durch fehlerhafte Verfügbarkeit von Marken in Vor- oder Nachbereichen entstehen.

In Abschnitt 5.2.3 zu multilateralen Kommunikation wurden bereits drei Phasen der Kommunikation ausgemacht: Eine lesende Phase, eine Entscheidungsphase und eine Aufarbeitungsphase. Aus diesen drei Phasen wurde die Einsetzbarkeit des Saga-Patterns abgeleitet. Auf die Realisierung bezogen können diese drei Phasen nun konkretisiert werden.

In der lesenden Phase finden vor allem Bindungssuchen statt. Die Phase endet damit, dass alle entsprechenden Marken, sofern ein Feuervorgang möglich ist, gesperrt werden, sodass sie nicht durch andere Transitionen konsumiert werden können. Ein intuitiver Ansatz für die entscheidende Phase wäre, dass hierbei alle Transitionen gemeinsam feuern. Die Semantik einer Pivottransition entspricht jedoch der Ausführung einer einzigen lokalen Transaktion auf einem Service bzw. im Anwendungsfall auf einer einzigen Plattform. Aus diesem Grund ist der Feuervorgang nicht unmittelbar durch die Pivottransition abbildbar. Da jedoch beim Erreichen der Pivottransition alle Marken bereits entsprechend gesperrt sein müssen, kann der Feuervorgang aus Sicht der Markierung nicht mehr fehlschlagen. Da

[10] Diese Inschriften werden erst beim Feuern und nicht schon bei der Bindungssuche ausgewertet, da sie nach Erwartung Seiteneffekte verursachen.

nun keine fachlichen Gründe mehr gegen den Erfolg sprechen, kann die Abarbeitung des Feuerns in die wiederholbare Phase der Saga verschoben werden.

Wie eingehend erläutert, können Seiteneffekte auch durch action-Inschriften entstehen. Die Ausführung einer action-Inschrift ist daher ein geeigneter Kandidat für das Pivotelement der Saga. Die obige Einschränkung auf eine lokale Transaktion bleibt jedoch bestehen, sodass der Einsatz von Petrinetz-Sagas eine Einschränkung auf maximal eine action-Inschrift pro aufgerufenen verteilten synchronen Kanal bedeutet. Action-Inschriften werden erfahrungsgemäß nicht häufig direkt an synchronen Kanälen eingesetzt, daher ist die Einschränkung vermutlich für die meisten Fälle verschmerzbar.

Das Feuern einer action-Inschrift benötigt die ausgewerteten Parameter, welche unter Umständen durch andere Unifikationen im Rahmen der Auswertung erst berechnet werden müssen. Bei der Betrachtung der Saga fällt allerdings auf, dass das Pivotelement stets vor den wiederholbaren Elementen liegt. Wie eben festgestellt, findet der wirkliche Feuervorgang jedoch erst innerhalb der wiederholbaren Elemente statt. Wird nun eine action-Inschrift als Pivotelement interpretiert, so muss diese logischerweise vor allen andern lokalen Feuervorgängen durchgeführt werden. Durch die Einschränkung auf bis zu einer action-Inschrift pro Aufruf eines verteilten synchronen Kanals entsteht jedoch auch an dieser Stelle kein Problem, da alle sonstigen Parameter für die action-Inschrift bereits während der Bindungssuche ausgewertet werden konnten und daher bekannt sein müssen. Folglich stellt es kein Problem dar, die action-Inschrift als erste umzusetzen.

Das Zurückrollen einer Saga durch Kompensieren der einzelnen kompensierbaren Transaktionen ist wenig kompliziert. Da vor dem eigentlichen Feuern keine Änderungen vorgenommen werden, sondern lediglich Markensperren vergeben werden, müssen diese Sperren nur entsprechend aufgehoben werden.

Die hier getroffenen Überlegungen bilden im Folgenden die Grundlage für die Konstruktion einer Erweiterung des Distribute Plugins: Resilient Distribute, dessen Ziel die Umsetzung von Abhärtung im Sinne der Cloud-Nativity im Kontext von verteiltem Renew umfasst.

12.7. Sagabasierte verteilte Referenznetzsimulationen

In Kapitel 5 wurde der abstrakte Aufbau komplexer Agentenkommunikation beschrieben. Es wurde motiviert, dass die Kommunikation grundlegend durch Anwendung des Saga-Patterns abgebildet werden kann. Die konkrete Ausgestaltung im Rahmen von Referenznetzen und die Erweiterung auf eine nebenläufige Vari-

ante wurde in diesem Kapitel vorgestellt und diskutiert. Das Saga-Pattern gibt die generelle Abfolge von lokalen Transaktionen wieder, macht jedoch keine Vorgaben bezüglich der technischen Umsetzung dieser Abfolge. Bei der Realisierung konkret im Kontext von Renew wurde die Betrachtung jedoch bislang abstrakt gehalten und lediglich die Operationen des Orchestrators als Netz abgebildet.

Aus diesem Grund motiviert dieser Abschnitt, wie eine konkrete Gesamtimplementation im Kontext von Renew aussehen kann. Dazu gilt es die grundlegende Form der Implementation festzulegen, Möglichkeiten der verteilten Identifikation zu erörtern, etwaige Änderungen bezogen auf den Simulationsalgorithmus zu klären sowie die Ableitung eines verteilten synchronen Kanals auf eine zugehörige Saga zu erörtern. Im Rahmen des Abschnitts wird sich die Notwendigkeit für eine Abwandlung des Distribute Plugins ergeben. Diese Abwandlung trägt den Namen »Resilient Distribute« und zielt auf die Abhärtung des Plugins ab, sowie den zugehörigen Technologiestack. Die dafür notwendigen Überlegungen werden ebenfalls dargelegt.

12.7.1. Sagas für Referenznetzsimulationen

Petrinetzbasierte Sagas sollen im Rahmen der Arbeit für komplexe Agentenkommunikation eingesetzt werden. Diese Kommunikationen entsprechen im Kontext von Renew im Wesentlichen dem Feuern verteilter synchroner Kanäle. Für die Implementation verteilter synchrone Kanäle steht bereits das Distribute Plugin zur Verfügung. Dieses bildet gerade in seiner Implementation jedoch nicht alle Feinheiten einer Saga ab.

Die Implementation des Distribute Plugins greift seinerseits in den Simulationskern von Renew ein. Falls dieses Verhalten nicht angepasst werden soll, wäre es ein möglicher Schritt daher verteilte synchrone Kanäle nicht mehr unmittelbar durch das Distribute Plugin auszuführen, sondern ein (plattformübergreifendes) Netz für die Ausführung einer Saga zwischenzuschalten. Die Saga wiederum könnte dann mit dem Distribute Plugin (und mehreren Aufrufen) realisiert werden. Eine Referenznetzsimulation mit Distribute Plugin würde somit zum eingesetzten Saga-Framework werden.

Der alternative Ansatz besteht darin, das Distribute Plugin anzupassen. In dieser Variante würde das Distribute Plugin durch einen Unterbau auf Sagabasis neu implementiert werden. Der Einsatz eines externen Saga Frameworks wird somit potentiell erforderlich.

In beiden Fällen sollte die Syntax des Distribute Plugins gewahrt bleiben, um Rückwärtskompatibilität zu Netzen und Lösungen zu gewährleisten, welche auf der Basis des Distribute Plugins umgesetzt wurden. Die beiden Lösungsansätze werden im Folgenden diskutiert.

Saga durch Distribute

Wenn eine Saga durch eine verteilte Referenznetzsimulation und damit durch das Distribute Plugin realisiert wird, steht an der zentralen Stelle die Distribute Registry, welche auf einer (Java) RMI Registry basiert. Sie würde wie der Nachrichtenbroker im System operieren und hätte die Aufgabe inne, jedem Teil der Simulation entsprechend Daten bereitzustellen.

Diese Form der Implementation ist insofern naheliegend, als sie die gesamte Strecke zur Realisierung der Kommunikation mit Referenznetzen abbildet. Ein entstehendes Problem ist dabei jedoch die Zentralität der RMI Registry. Diese besitzt im Normalfall keine Replikation und auch keine Ausfallsicherheit. Kommt es zu einem Absturz der RMI Registry ist normalerweise keine weitere Kommunikation mehr möglich und bereits gestartete Kommunikationen müsste über Mechaniken wie beispielsweise True-Concurrency Checkpoints (vgl. Abschnitt 11.6.6) persistiert werden. Dieses Verhalten entspricht kaskadierenden Fehlschlägen und sollte in jedem skalierbaren System verhindert werden. Entsprechendes gibt auch der Ansatz der Cloud-Nativity in Bezug auf »Abhärtung« vor, welcher Teil der Basis der Mushu-Plattformarchitektur darstellt.

Der bereits erwähnte Ansatz aus (Baratloo u. a., 1998) in Form einer replizierten Registry für Java RMI ist aus der Zeit gefallen und wird nicht weiter vorangetrieben, sodass ein Einsatz nicht zukunftsfähig wäre.

Ein weiteres Problem besteht darin, dass alte Netzinstanzen nicht automatisch aufgeräumt werden und auch nach Ende der Simulation noch in der RMI Registry bestehen würden. Gleichzeitig wurde die Distribute Registry nicht für den Einsatz nebenläufiger Simulationen konstruiert und bietet keine Kapselung auf Simulationsebene. Während dies nicht zwangsläufig zum Problem führen muss, ist es jedoch für nebenläufige Simulationen eine unsaubere Lösung. Beide Aspekte könnten durch die Einführung individueller Namen bezogen auf eine Simulation gelöst werden. Dies würde allerdings eine Änderung der Distribute Registry bedeuten, welche bei diesem Ansatz jedoch eigentlich ausgeschlossen werden sollte.

Zusammengefasst mag der Ansatz durch seinen durchgehenden Einsatz von Referenznetzen naheliegen, ist jedoch für den hier geplanten Anwendungsfall aus den aufgeführten Gründen nicht geeignet.

Distribute durch Saga

Auf der Basis dieser Überlegungen wird klar, dass der Ansatz, das Distribute Plugin nicht abzuändern, nicht haltbar ist. Wird nun von einer Änderung am Plugin selbst ausgegangen, so sollte nach Möglichkeit zumindest die bekannte Schnittstelle durch Distribute nicht grundlegend geändert werden, sodass alte

Lösungen weiterhin funktionieren. Das Ziel ist es dennoch Distribute auf eine entsprechende Saga-Semantik umzukonstruieren.

Um dieses Ziel umzusetzen, sind die Überlegungen aus dem Abschnitt 12.6.4 bezüglich Isolation hilfreich: Die Bindungssuche kann vor der Saga erfolgen, Marken können über alle Teilnehmer hinweg in einer kompensierbaren Phase gesperrt werden, eine entsprechende »action«-Inschrift kann als Pivot-Transition dienen und das Erzeugen und Verbrauchen einzelner Marken kann abschließend erfolgen. Das Erzeugen und Verbrauchen einzelner Marken kann nach dem erfolgreichen Sperren der Marken und Feuern der Pivot-Transition nur noch aus technischen temporären Gründen fehlschlagen und daher beliebig oft neu versucht werden.

Ein deutlicher Vorteil bei dem Einsatz von Sagas als Unterbau umfasst die Möglichkeit, das Feuern eines verteilten synchronen Kanals zeitlich zu strecken. Einzelne Teilnehmer können während des Feuerns offline gehen oder von den anderen Teilnehmern getrennt werden und können den Vorgang fortsetzen, sobald wieder eine Verbindung besteht. Diese Eigenschaft war in der damaligen Konstruktion von Distribute nicht fokussiert, ist aber äußerst hilfreich, sobald die manuelle Ausführung des Protokolls in eine unbeaufsichtigte Form übergeht.

12.7.2. Verteilte synchrone Kanäle zu Petrinetz-Sagas

Im vergangenen Teil des Kapitels wurden Sagas auf Petrinetz-Basis vorgestellt. Ebenso wurde herausgearbeitet, dass der Distribute Algorithmus auf der Basis von nebenläufigen Sagas ausgeführt werden soll. Eine Frage ist dabei jedoch noch offen: Wie können entsprechende Sagas dynamisch zur Laufzeit aus den Transitionen einer zu simulierenden verteilten Referenznetzsimulation abgeleitet werden? Dieser Abschnitt dient der Vorbereitung der Beantwortung dieser Frage.

Um eine Aussage darüber zu treffen, wie Sagas die Bindungssuche und den Feuerprozess unterstützen können, liegt es nahe, ein Blick auf den entsprechenden Simulationsalgorithmus in Renew zu werfen. Nach (Kummer, 2002) besteht dieser aus der Unifikation, dem Triggering, der Definition von Datenstrukturen und dem Feuern selbst. Die Auflösung von Referenzen ist eine Grundfrage der zitierten Arbeit und ist daher nicht gesondert als spezieller Anteil des Simulationsalgorithmus ausgewiesen, spielt jedoch selbstverständlich eine entscheidende Rolle.

Referenzen lokal aufzulösen ist durch das mögliche globale Wissen im Simulator (in der Plattform) wenig kompliziert. Referenzsysteme wie beispielsweise das Java Objektsystem bieten dabei eine starke Grundlage und Unterstützung. Problematisch werden diese Umsetzungen jedoch bei der Verteilung. Hier besteht kein generelles globales Wissen mehr und es muss auf eindeutige Identifikatoren bzw. Namen zurückgegriffen werden, welche dann als Referenzen fungieren können.

Wird also der Rahmen der lokalen Plattform verlassen, muss ein Übersetzungsschritt von lokaler zu globaler Identifikation erfolgen.

Unter der Annahme, dass Referenzen aufgelöst werden können, gilt es nun zu überlegen, wie die Bindungssuche bei einem verteilten synchronen Kanal abläuft. Um die Komplexität des Algorithmus handhabbar zu halten, wurde in (Kummer, 2002) das Konzept der Up- und Downlinks eingeführt. Downlinks halten eine Referenz auf eine andere Netzinstanz und können damit gezielt einen bestimmten Satz an Uplinks (bezogen auf das Gesamtsystem) ansprechen, um eine Synchronisation zu initiieren. Aus diesem Grund sollten neben der Identifikation auch grundlegende Informationen zu Uplinks jeder Netzinstanz, welche an verteilten Schaltprozessen teilnehmen soll, gespeichert werden. Mit diesen Informationen kann eine Vorauswahl der möglichen Bindungspartner erfolgen und die Bindungssuche nach dem Distribute Modell direkt zwischen den entsprechenden Plattformen ausgehandelt werden. Bei Bindungssuchen handelt es sich (effektiv[11]) um rein lesende Prozesse, daher ist das komplexere Saga Modell an dieser Stelle noch nicht erforderlich.

Die Anteile zu Unifikation und Datenstrukturen werden dabei durch das Distribute Plugin vorgeben und semantisch nicht wesentlich verändert. Insbesondere die Unifikation beschränkt sich dabei auf die einfacheren Send-Channels von Distribute, da durch eine echte verteilte Unifikation ein nicht unerheblicher Overhead an Nachrichten entstehen würde. Ebenfalls kann beim Triggering auf die Lösung im Rahmen von Distribute zurückgegriffen werden.

Falls eine Bindung gefunden wurde, konstruiert der Initiator eine entsprechende einfache Petrinetz-Saga, bei der zunächst alle entsprechenden Marken auf allen Plattformen gesperrt werden, danach als Pivottransition eine etwaige »action«-Inschrift ausgeführt wird und abschließend Marken generiert und verbraucht werden. Die so konstruierte Saga wird sodann an einen verfügbaren Orchestrator versandt. Der Orchestrator beginnt mit der Transformation und Simulation der Saga entsprechend der im übrigen Kapitel ausgeführten Überlegungen. Der generelle Ablauf des Feuervorgangs ist entsprechend in Abbildung 12.11 für zwei Plattformen und einer möglichen »action« Inschrift auf der entfernten Plattform dargestellt.

12.7.3. Globale Eindeutigkeit und Identifikation

Die Notwendigkeit eines Systems für globale Identifikation bzw. Eindeutigkeit wurde im vergangenen Abschnitt erörtert. Darüber hinaus wurden entsprechende Anforderungen bereits im Konzept in Abschnitt 5.3.1 und Abschnitt 7.2.1 formu-

[11] Bindungssuchen nutzen sogenannte State Recorder, um die Semantik einer rein lesenden Operation zu erreichen.

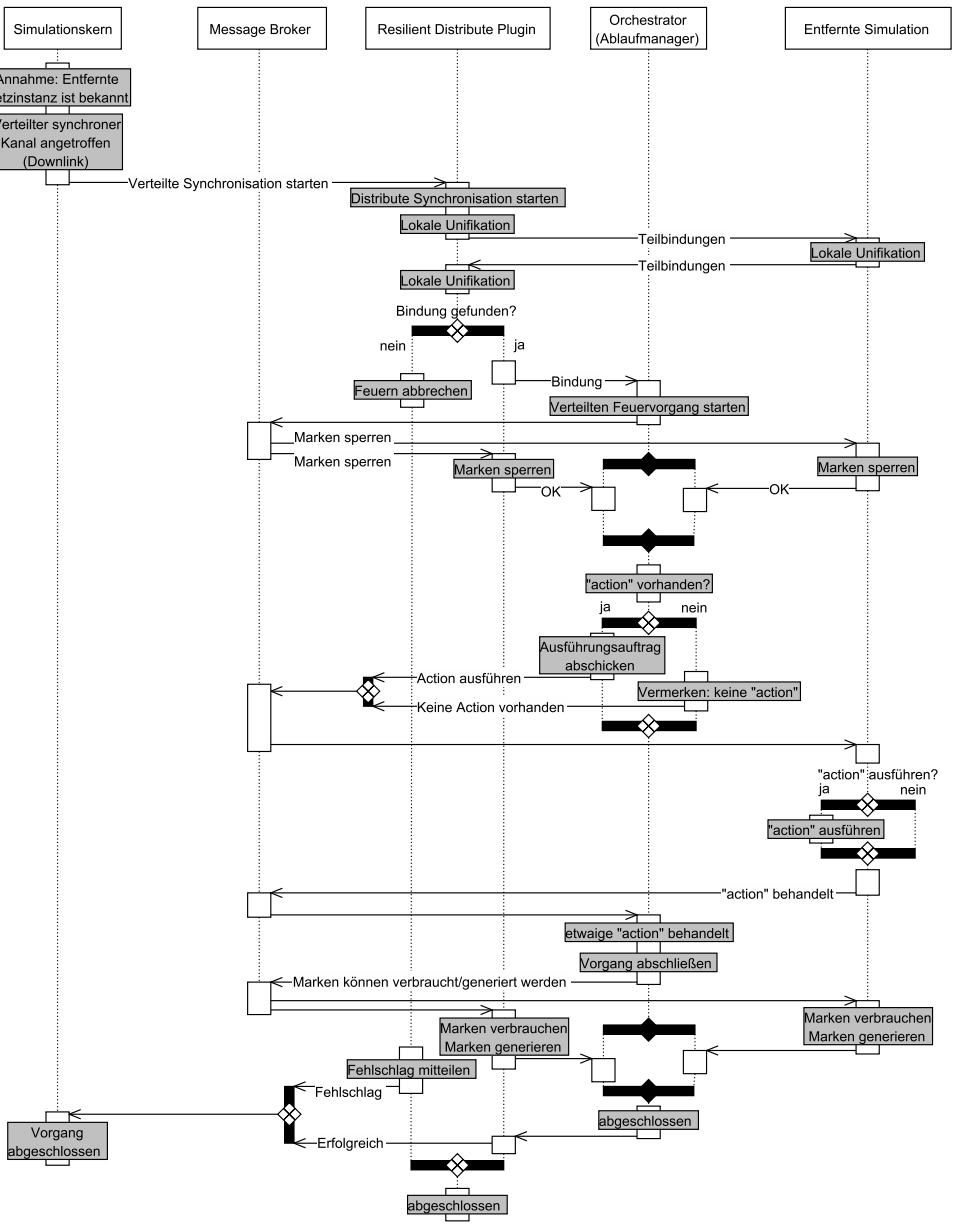

Abbildung 12.11.: Feuern verteilter synchroner Kanäle auf Basis von Petrinetz-Sagas

liert. Als Ausgangspunkt kann die entsprechende Implementation im Distribute Plugin dienen und zu diesem Zweck untersucht werden. Distribute verwendet ganzzahlige Netzinstanz-IDs und eine zentrale Registry, in der diese einzigartig zugewiesen werden. Folglich liegt es nahe, für die Lösung in Resilient Distribute die äquivalente Komponente, nämlich den Nachrichtenbroker heranzuziehen. Damit wäre das Management von Identifikation de facto eine Komponente des Plattformmanagements.

Der Nachrichtenbroker ist allerdings im Gegensatz zur Distribute Registry eine rein passive Komponente, ist nicht hierarchisch geordnet und daher nicht in der Lage, die gleichen Funktionalitäten auszuführen. Darüber hinaus ist es auch nicht wünschenswert, wenn Funktionalitäten weiterhin zentral verwaltet werden, da jede zentral verwaltete Einheit negative Einflüsse auf die Skalierbarkeit des Gesamtsystems hat. Eine logische Konsequenz daraus wäre es, die Identifikation auf die jeweiligen Plattformen abzuwälzen. Dabei muss jedoch ein Verfahren eingesetzt werden, welches trotz der lokalen Vergabe eine globale Eindeutigkeit garantiert.

Der Einsatz von UUIDs kann hier erwägt werden. Diese sind zwar nicht vollständig gegen Kollisionen geschützt, sind aber ausreichend lang, sodass Kollisionen für alle praktischen Anwendungsfälle vermieden werden. Die Vermittlung der jeweiligen Identifikatoren muss jedoch immer noch zwangsläufig über das persistente Kommunikationsmedium des Plattformmanagements erfolgen: den Nachrichtenbroker.

Für die Referenzierung einer bestimmten Netzinstanz verwendet Distribute neben der Nummer als eindeutigen Identifikator das Konzept eines stringbasierten Schlüssels. Dieser wird in der Distribute Registry angefragt und falls eine zugehörige Netzinstanz existiert, welche sich zuvor unter diesem Schlüssel registriert hat, werden die Kontaktmöglichkeiten zu dieser Netzinstanz herausgegeben. Dieses System ist im Wesentlichen auch auf einem Nachrichtenbroker umsetzbar, dafür muss sich die Plattform selbst aber wiederum geeignet ausweisen können, um die Netzinstanzen zuordnen zu können.

Um einer Plattform die Möglichkeit der Zuordnung zwischen Identifikation und Netzwerkkoordinaten einer anderen Plattform zu ermöglichen, könnte beispielsweise eine Service Discovery eingesetzt werden. Eine sehr einfache Implementation wäre ein einfaches Vorhalten an einer ausgezeichneten Stelle auf dem Nachrichtenbroker. Das Problem, dass bei Distribute eine einzelne Registry nicht mehrfache nebenläufige Simulationen verwalten konnte, kann an dieser Stelle ebenfalls durch die Einführung eines weiteren Identifikators für die Gesamtsimulation selbst gelöst werden. Entsprechend diesem Identifikator kann ein Teilbereich des Nachrichtenbrokers verwendet werden, um dort entsprechende Informationen zu aktiven Plattformen vorzuhalten.

Beim Start von neuen Plattformen kann der Identifikator der Gesamtsimulation an jede neue Plattform durch das Plattformmanagement weitergegeben werden. Dieser Identifikator ist für die gesamte Simulation in all ihren Komponenten identisch und dient der Isolation auf der technischen Implementationsebene des Nachrichtenbrokersystems. Zusammenfassend sind daher jeweils eine UUID für die Simulation, je für jede Plattform bzw. Simulator und je für jeden Agenten bzw. jede Netzinstanz notwendig.

12.7.4. Einfache Kommunikation via Nachrichtenbroker ohne Sagas

Für eine vollwertige Umsetzung von Resilient Distribute ist neben dem Feueralgorithmus eine entsprechende Anpassung von Bindungssuche und Triggering erforderlich, da diese so auch über den Nachrichtenbroker erfolgen muss. Da es sich bei all diesen Anteilen um direkte Eingriffe in den Simulationsalgorithmus handelt, sollte damit nicht leichtfertig umgegangen werden.

In Abschnitt 5.2.2 wurden die verschiedenen betrachteten Modalitäten der Kommunikation herausgearbeitet: einfache und komplexe Kommunikation. Einfache Kommunikation wurde in den Abschnitten der Arbeit insbesondere in Kapitel 11 zur Konstruktion der Plattform primär für die Kommunikation zwischen Plattformen oder Plattformmanagement und Plattformen eingesetzt.

Einfache Kommunikation zwischen Agenten auf Plattformen und somit in der Realisierung zwischen Netzinstanzen ist durch die Implementation komplexer Kommunikation auf dieser Ebene grundsätzlich auch möglich. Für einfache Anwendungsfälle wie beispielsweise die Modellierung einer Fernsehansprache und zugehöriger Zuschauer kann aber argumentiert werden, dass die Implementation einer komplexen Kommunikation übersetzt ist.

Daher ist eine einfache Implementation als Zwischenstufe zur vollständigen Implementation verteilter nebenläufiger Sagas für Referenznetzsimulationen denkbar. Die Erhaltung der Ausfallsicherheit durch einen entsprechenden passiven Nachrichtenbroker ist jedoch wünschenswert, sodass weiterhin auf Implementationen ohne die klassische Distribute Registry zurückgegriffen werden soll. Ohne Änderungen am Simulationsalgorithmus vorzunehmen, sollte der Versand und Empfang zum und vom Nachrichtenbroker in entsprechenden Netzen realisiert werden.

Das Distribute Plugin arbeitet bedingt durch Java RMI mit dem Konzept von Netzinstanz-Stubs. Somit besteht für einen gegebenen Up- oder Downlink ein entsprechender lokaler (virtueller) Partner. An dieser Stelle kann angesetzt werden und auf Basis von entsprechenden Netzen eine Kommunikation mit dem Nachrichtenbroker eingeleitet werden. Zu diesem Zweck müssen lediglich entsprechende

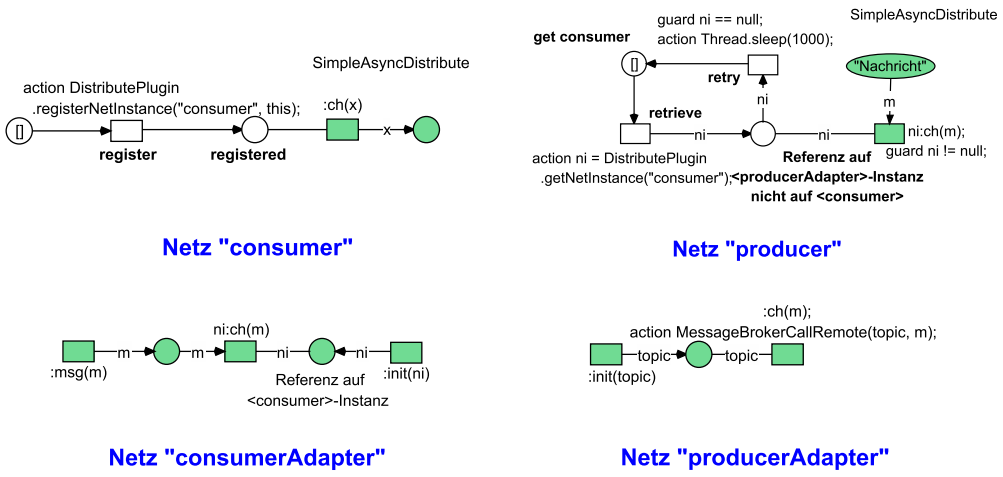

Abbildung 12.12.: Beispiel: Einfache Nachrichtenbroker Kommunikation

Adapternetze generiert werden. Der Aufruf eines solchen Kanals wäre somit keine echte Synchronisation mehr und würde auch nur in der Terminologie der Arbeit einfache Kommunikation unterstützen, würde jedoch die technologischen Grundlagen schaffen, welche ebenfalls für die Implementation der komplexen Kommunikation auf Sagabasis notwendig sind.

In Abbildung 12.12 ist ein entsprechendes, sehr einfaches Producer-Consumer-Beispiel dargestellt. Dieses ist in einer Diskussion des Autors dieser Arbeit zusammen mit Daniel Moldt, Michael Haustermann und Alexander Senger im Rahmen der Arbeit (Senger, 2021) entstanden. Es umfasst die vier Netze: `consumer`, `consumerAdapter`, `producer` und `producerAdapter`. In `producer` und `consumer` sind jeweils die bekannten Interaktionen mit dem Distribute Plugin abgebildet. Bei der Registrierung der `consumer`-Netzinstanz wird deren Identifikation auf dem Nachrichtenbroker verfügbar gemacht. Gleichzeitig muss für jeden Uplink eine entsprechende Konstruktion im Netz `consumerAdapter` bestehen, mit der sich der Uplink synchronisieren kann. Sobald eine Nachricht eingeht, wird durch Programmlogik und sogenannte Netzstubs eine Marke auf die Stelle vor dem zugehörigen Downlink im Adapternetz gelegt. Netzstubs ermöglichen den Zugriff auf synchrone Kanäle durch klassische Methodenaufrufe. Dargestellt ist dies durch den synchronen Kanal `:msg(m)`. Auf der anderen Seite im `producer` Netz kann die entsprechende Netzinstanz angefordert werden. Anstatt wie beim Distribute Plugin eine Referenz auf einen lokalen Stub der Netzinstanz zu verwenden, wird eine Referenz auf einen entsprechenden Adapter eingesetzt. Das `producerAdapter` Netz bietet einen entsprechenden Uplink und eine Nachrichtenbroker Interaktion mit dem Aufruf des zugehörigen synchronen Kanals.

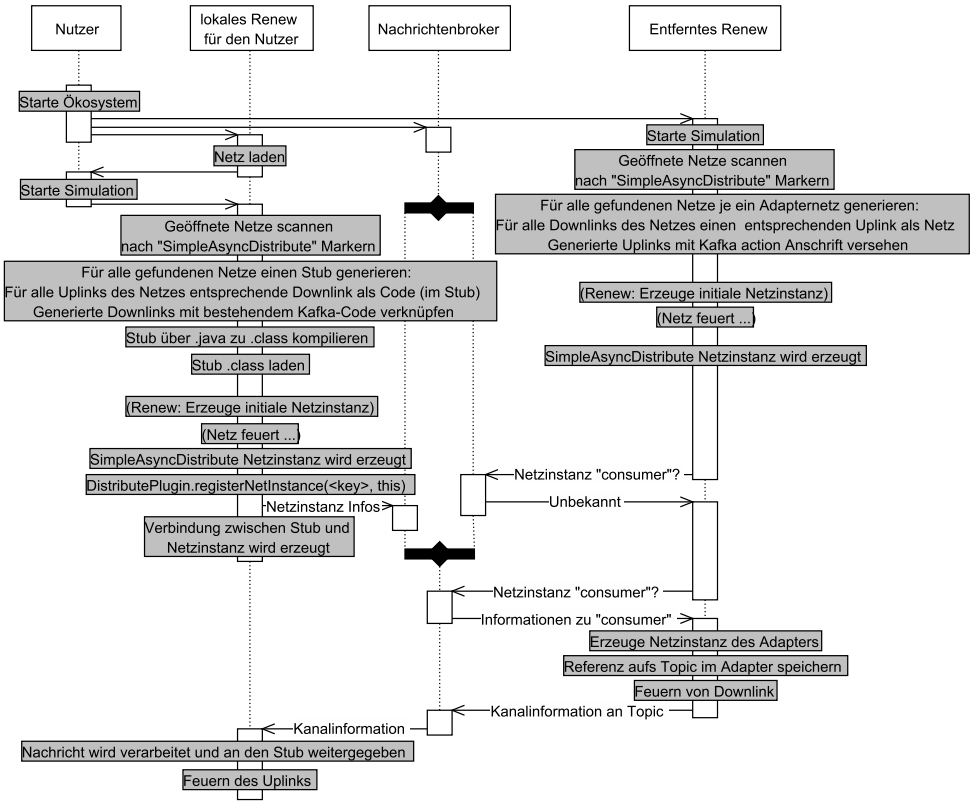

Abbildung 12.13.: Ablaufdiagramm: Einfache Kommunikation und Feuervorgang per Nachrichtenbroker

In diesem Beispiel ist die Variante gewählt, in der das `consumer` Netz nur einen Uplink besitzt und das `producer` Netz nur einen Downlink. Die entsprechenden Konstruktionen müssten für Uplinks im `producer`-Netz und Downlinks im `consumer`-Netz ausgetauscht werden. Der Nachrichtenbroker entspricht somit semantisch einem Platz zwischen den beiden Adapternetzen.

Soll dieses Vorhaben mit Renew umgesetzt werden, so müssen nur einige Anteile neu implementiert werden. Alle lokalen Prozeduren bezüglich synchroner Kanäle sind bereits vorhanden. Sofern die Identifikation von Netzinstanzen bereits umgesetzt ist, muss lediglich die Generierung der entsprechenden Adapternetze folgen. Dabei kann beim Registrieren einer Netzinstanz am Distribute Plugin ein entsprechender Adapter erzeugt werden, da die Netzinstanz selbst übergeben wird. Auf der anderen Seite könnte beim Abrufen der entfernten Netzinstanz stattdessen direkt ein entsprechendes Adapternetz zurückgeliefert werden, welches entsprechende Kanalaufrufe abfängt und in Nachrichtenbroker Nachrichten übersetzt. Abbildung 12.13 zeigt den zugehörigen Ablauf im Kontext von Renew, welches ebenfalls in der Diskussion im Kontext der Arbeit (Senger, 2021) enstanden ist.

Entsprechend der Semantik einfacher Kommunikation unterstützt dieses Protokoll keinen multilateralen Informationsaustausch. Ebenso wird vom Sender der Nachricht keine Empfangsbestätigung angefordert. Im Beispiel wird nach dem Feuern des Kanals `ni:ch(m)` im Netz `producer` nicht auf eine Antwort gewartet. Das weitere Verhalten vom `producer` ist unabhängig davon, ob und wann die Nachricht `m` beim `consumer` Netz eintrifft.

12.7.5. Komplexe Kommunikation

Ist einfache Kommunikation nicht hinreichend, muss auf die volle komplexe Kommunikation zurückgegriffen werden. Zu diesem Zweck müssen zusätzlich zur Identifikation und Auflösung der Referenzen neben dem zuvor beschriebenen Feueralgorithmus via Sagas ebenfalls das Triggering und die Bindungssuche bedacht werden.

Das Distribute Plugin stellt umfangreiche Lösungen zur Implementation von verteilter Bindungssuche und verteiltem Triggering bereit. Die Algorithmen sollen im Kern nicht verändert werden, nur ihre jeweiligen verteilten Schritte sollen via einfacher Kommunikation erfolgen. Dafür ist eine Übersetzung von Java RMI Aufrufen zu entsprechend anderen Aufrufen notwendig. Analog zu dem Einsatz von Petrinetz-Sagas beim Feueralgorithmus wäre der Einsatz einfacher Kommunikation durch Netze wie zuvor beschrieben eine Möglichkeit der Umsetzung. Gleichermaßen kann aber im Kontext der Webservices argumentiert werden, dass für derart einfache Aufrufe HTTP ein geeignetes Protokoll darstellt. Auch der Einsatz des ohnehin vorhandenen Nachrichtenbrokers analog zur einfachen Kommunikation zwischen Netzinstanzen wäre möglich. Aus diesem Grund wird lediglich ein Interface der Kommunikation umrissen. Die detaillierten Anforderungen an die Komponenten des Triggerings und der Bindungssuche sind im Wesentlichen in den Quellen (Kummer, 2002) und (M. Simon und Moldt, 2016) beschrieben. Daher werden die Rahmenaspekte nur kurz wiederholt, aber nicht im Detail ausgeführt.

Ausgetauschte Daten müssen nach wie vor bezogen auf gewählte Implementation serialisierbar vorgehalten werden, wie dies auch bereits bei der Java RMI Lösung der Fall war. Darüber hinaus wird ebenfalls der Ansatz, dass der initiierende Downlink als Ausgangspunkt (»root«) der Suche behandelt wird, beibehalten.

Triggering

Das Triggering umfasst im Kern die Sammlungen `TriggerableCollection` und `TriggerCollection`. Diese sind abstrakt gehalten und können durch beliebige Objekte repräsentiert werden. Bekannte Inkarnationen von Triggern sind Platzinstanzen und solche von Triggerables entsprechend Transitionsinstanzen. Distri-

bute löst die Zugriffe auf diese Sammlungen durch sogenannte Accessors. Diese bieten Methoden zum Hinzufügen und Entfernen an und Hilfsfunktionen wie die Berechnung des Hashcodes. `TriggerableCollections` können ebenfalls Suchen vorschlagen und auf den entfernten Vergleich von `TriggerableCollections` zurückgreifen. Diese Bestandteile müssen entsprechend durch Schnittstellen abgebildet werden.

Bindungssuche

Die Bindungssuche besteht im Kern aus den Komponenten `Searcher`, `Binder` und `Finder`. Dabei pflegt ein `Searcher` eine Reihe von `Bindern`. `Binder` wiederum versuchen eine einzelne Bindung zu erzeugen und danach rekursiv den Kontrollfluss an den `Searcher` zurückzugeben. Wird bei diesem Prozess eine vollständige Bindung gefunden, wird diese in einem `Finder` Objekt gesichert. `Finder` Objekte bilden die Grundlage für den Feuerprozess.

Der Bindungssucheprozess in Distribute erweitert diesen Ansatz durch ähnliche Accessors, wie sie bereits beim Triggering zum Einsatz gekommen sind. Darüber hinaus wird das Konzept der »root search« eingeführt. Diese startet die wesentlichen Teile des Algorithmus und liegt für gewöhnlich beim initiierenden Downlink, sofern der aktuelle Aufruf kein Teil eines übergeordneten Aufrufs ist.

Die anzupassenden Teile umfassen im Wesentlichen die Accessors zu den oben genannten Teilen des Suchalgorithmus. Dies umfasst insbesondere den `Searcher-Accessor` und den `RemoteSuccessNotifier`, in denen die Übertragung via Java RMI ausgetauscht werden muss.

12.7.6. Gesamtentwurf von Resilient Distribute

Abschließend soll in diesem Abschnitt der gesamte erarbeitete Ablauf der »Resilient Distribute« Lösung zusammengefasst werden. Resilient Distribute umfasst die Umsetzung von einfacher und komplexer Kommunikation zwischen Agenten, wie sie in Kapitel 5 eingeführt wurde. Die zentralen Aspekte bei der Umsetzung sind dabei der weitestgehende Verzicht auf synchrone Kommunikation bei seiteneffektbehafteten Operationen sowie die Vermeidung einzelner kritischer Komponenten im Gesamtsystem, deren Ausfall potenziell Auswirkungen auf alle anderen Systemkomponenten hätte (kaskadierende Fehlschläge). Für die beiden Kommunikationsformen sind unterschiedliche Anteile vom gesamten Resilient Distribute Entwurf erforderlich, wie in Abbildung 12.14 dargestellt ist.

Die Identifikation von Agenten, Plattformen und Simulationen, bzw. von Netzinstanzen, Simulatoren und Simulationen in der technisch Sicht, wird durch jeweils lokal vergebene UUIDs realisiert. Diese sind nicht konfliktfrei, Namenskonflikte

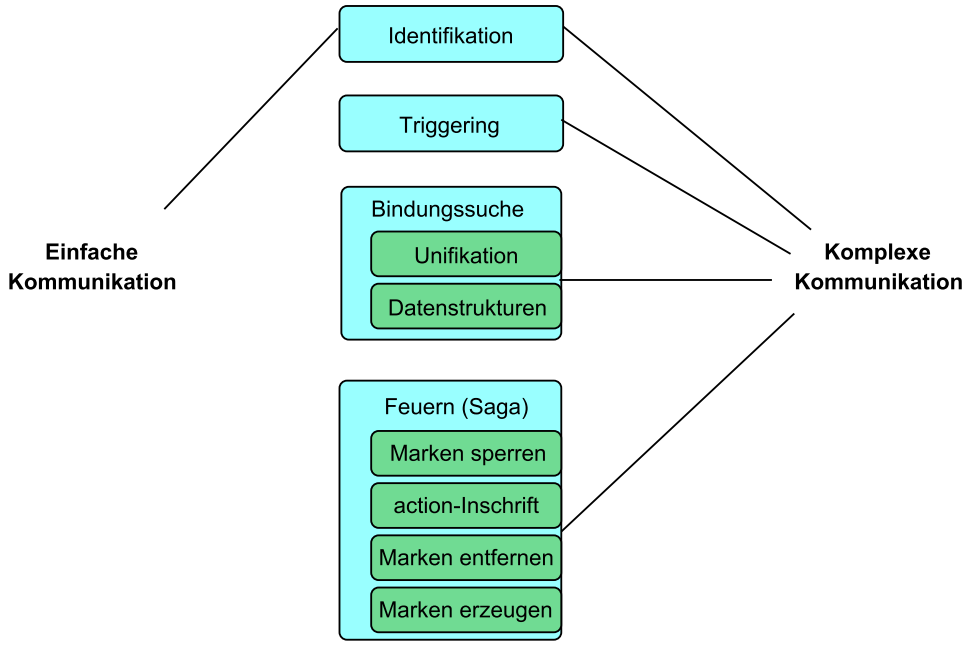

Abbildung 12.14.: Anteile des Simulationsalgorithmus bezogen auf die Kommunikationsform

treten aber extrem selten auf. Für den Abruf der Identifikatoren dient eine geeignete Service Discovery auf der Ebene des Plattformmanagements, welche im einfachsten Fall durch einen Nachrichtenbroker realisiert werden kann.

Auf der Basis der Identifikation ist bereits einfache Kommunikation möglich, welche sich durch fehlende Empfangsbestätigungen und unilateralen Informationsaustausch auszeichnet. Sie entspricht dem klassischen Versand einzelner Nachrichten an einen oder mehrere Empfänger. Resilient Distribute realisiert diese Kommunikationsform mit Adapternetzen, die jeweils lokale Gegenstücke zu den Up- und Downlinks bilden. Ein Hinweis »SimpleAsyncDistribute« im Netz gibt Aufschluss hierauf. Der Vorteil bei der Realisierung liegt darin, dass der Kommunikationspartner zum Zeitpunkt des Nachrichtenversands keine Verbindung zum System haben muss und die Nachricht später abrufen kann.

Ist einfache Kommunikation nicht ausreichend, kann der Ablauf für komplexe Kommunikation entsprechend Abbildung 12.11 auf Seite 352 durch den Modellierer gewählt werden. Die Unterscheidung erfolgt durch das Auslassen des Hinweises »SimpleAsyncDistribute« sowie der Verwendung der Distribute Send Channel Syntax (vgl. Abschnitt 2.8.8 im Grundlagenkapitel).

Dabei überwacht der verteilte Triggering-Algorithmus welche Transitionen erneut auf Aktiviertheit überprüft werden müssen. Die verteilte Bindungssuche liefert so-

dann `Finder`-Objekte, welche Informationen zu einer validen Bindung enthalten. Beide Algorithmen sind im Kern unverändert im Vergleich zur ursprünglichen Implementation von Distribute, lediglich die Kommunikation sollte nicht mehr über Java RMI erfolgen. Die genaue Kommunikationsform sollte nicht festgelegt werden, sondern viel eher ein entsprechendes Interface definiert werden. Für eine konkrete Umsetzung bietet sich auch hier der Einsatz des Nachrichtenbrokers an, um die Asynchronität der Operationen auch bei Bindungssuche und Triggering zu erhalten.

Ein erfolgreich erzeugtes Finder-Objekt enthält Informationen zu gebundenen Marken, Kanälen, Plätzen und Transitionen. Aus diesem Objekt wird dann beim initiierenden Downlink die Beschreibung einer Petrinetz-Saga generiert, bei der in der kompensierenden Phase nebenläufig alle relevanten Marken gesperrt werden, in der Pivot-Phase bis zu einer etwaigen Transition mit »action«-Inschrift ausgeführt wird und in der wiederholbaren Phase nebenläufig alle Marken konsumiert und produziert werden. Durch die Natur der Saga ist der Ablauf zu jeder Zeit unterbrechbar, falls einer der Teilnehmer nicht verfügbar ist. Die fehlende Isolation der Saga wird abgefangen durch das Sperren der Marken auf Applikationsebene.

Nach Konstruktion der Saga wird diese an einen Saga-Orchestrator (»Ablaufmanager« im Kontext vom Mushu-Plattformmanagement) übergeben, welcher diese dann unabhängig vom initiierenden Downlink ausführt.

Insgesamt ergibt sich so eine robuste Implementation komplexer Kommunikation, welche im Wesentlichen nicht mehr von der ständigen Verfügbarkeit der Teilnehmer, dem Koordinator und dem Kommunikationsmedium abhängt. Permanente Ausfälle betreffen nur den einzelnen Feuervorgang, ziehen jedoch keine Unbeteiligten mehr mit.

Dies gilt unter der Voraussetzung, dass der eingesetzte Nachrichtenbroker seinerseits wiederum ausreichende Replikation und damit Ausfalltoleranz aufweist. Diese Anforderungen wird jedoch im Normalfall von allen modernen Lösungen angeboten.

12.8. Technische Gesamtübersicht der Mushu-Umsetzung mit Renew

Mit der Verfügbarkeit von komplexen Kommunikationsmechaniken stehen nun alle Komponenten bereit, um eine Gesamtbeschreibung der möglichen Realisierung der Mushu-Architektur vorzustellen. Die Komponenten integrieren dabei die einzelnen Bestandteile, welche in Kapitel 9 bis Kapitel 12 vorgestellt wurden. Abbildung 12.15 stellt einen Ausschnitt aus dem gesamten integrierten System

auf technischer Ebene dar und erweitert damit die (Teil)abbildung 10.4 aus Kapitel 10. Dabei liegt der Fokus auf den einzelnen physikalischen und logischen Einheiten und nicht auf dem Ablauf der Interaktionen der Komponenten miteinander. Diese sind zu komplex, um sie in einem integrierten Diagramm darzustellen. Daher sei für die Interaktionen auf die Ausführungen der vergangenen Kapitel und auch auf die konzeptuellen Kapitel 5 bis Kapitel 7 verwiesen.

In der Abbildung finden sich die bekannten Komponenten des Management-Systems und des Clustermanagers. Auch die Workerknoten sind in ihrer ursprünglichen Form vertreten. Auf der Ebene der einzelnen Plattform (also innerhalb eines Headless Renew Simulators innerhalb eines Workerknotens) ist das umfangreiche Cloud-Native Renew-Plugin hinzugekommen, welches einen Großteil der Mushu-spezifischen Struktur der Plattform bündelt.

Auf der Ebene der Kommunikation ist das Distribute Plugin innerhalb der Plattform durch das Resilient Distribute Plugin ersetzt worden. Es kommuniziert mit einem Orchestrator für petrinetzbasierte Sagas, welcher auch repliziert vorliegen kann. Der Orchestrator besitzt ein Petrinetz-Saga-spezifisches Plugin, um mit dem Kommunikationsmedium zu kommunizieren. Das Kommunikationsmedium sollte selbst in einer replizierten Form vorliegen und aus den Ausführungen dieses Kapitels kann abgelesen werden, dass ein Nachrichtenbroker als Kommunikationsmedium die geeignetste Lösung darstellt. Die drei Instanzen des Nachrichtenbrokers sind dabei exemplarisch dargestellt und die Kommunikation zwischen Petrinetz-Saga-Plugin, Resilient Distribute Plugin und Nachrichtenbroker kann mit beliebigen Instanzen des Nachrichtenbrokers durchgeführt werden.

Des Weiteren wurde wie bereits angekündigt die Idee der Primärinstanz innerhalb der Simulation nicht weiter übernommen und eine mögliche GUI auf der Workstation wird abstrakt beschrieben. Eine solche GUI ist nicht Teil dieser Arbeit, würde sich aber am Nachrichtenbroker und dem Status Monitor orientieren, um Simulationsabläufe und Ergebnisse und Teilergebnisse der Simulation auszugeben. Der angesprochene Status Monitor ist nun eine eigenständige Komponente, welche die Aggregationsfunktionen erfüllt, wie sie in Abschnitt 10.8.2 dargestellt wurde.

Als eine weitere zusätzliche Komponente wurde nunmehr auch die Image Registry eines CI/CD Servers in der Abbildung aufgenommen. Die Image Registry stellt die Container-Grundlagen für alle containerisierten Anwendungen bereit. Sie wird durch den Clustermanager angesprochen und liefert den physikalischen Maschinen (auf der nativen Ebene) die Images, aus denen Container erzeugt werden können. Sie stellt auf abstrakter (Mushu-)Ebene daher insbesondere auch die Plattformdefinition bereit.

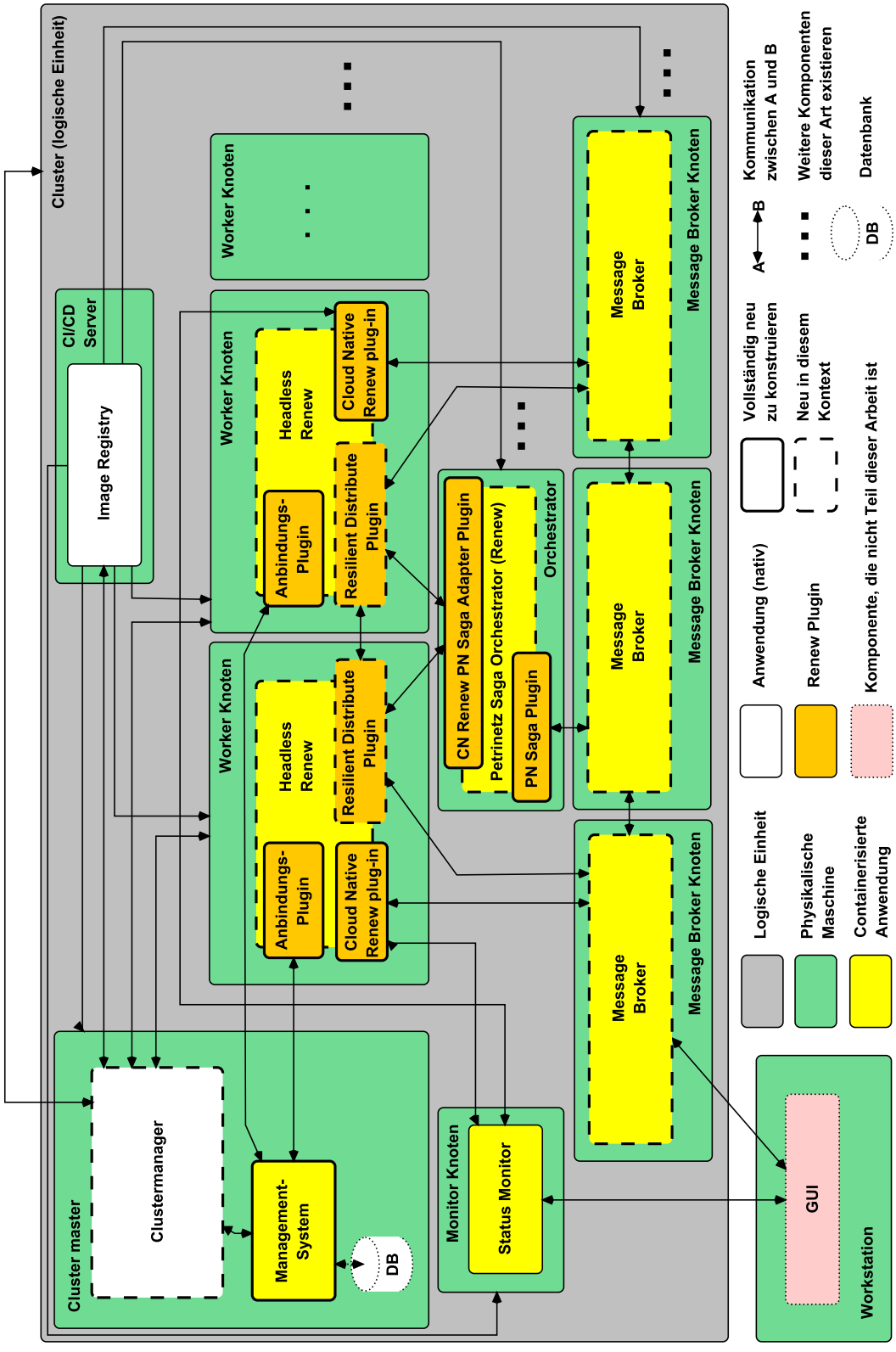

Abbildung 12.15.: Schematische Darstellung der vorgeschlagenen technischen Realisierung von MUSHU

12.9. Zusammenfassung - Umsetzung der Agentenkommunikation

In Kapitel 5 wurde bereits motiviert, dass sich komplexe Kommunikation im Sinne der Definition der Arbeit als verteilte Transaktionen bzw. als verteilte Saga modellieren lässt. Die dabei fehlende Nebenläufigkeit und Unterstützung durch einen Formalismus wurde im Rahmen dieses Kapitels dargelegt und eine entsprechende Lösung vorgestellt. Es ergab sich, dass Petrinetz-Sagas als P/T-Netz modelliert werden sollten und zur Ausführung in eine Referenznetzdarstellung transformiert werden sollten, um Einfachheit der Modellierung und Eigenschaften der Formalismen auszunutzen. Es wird der Einsatz eines Orchestrators angenommen, welcher im hier betrachteten Anwendungsfall »Resilient Distribute« repliziert auftreten kann. Resilient Distribute sieht die Abbildung vom Feuern verteilter synchroner Kanäle auf entsprechende nebenläufige Sagas vor sowie die Aushandlung und die Bindungssuche abseits von Sagas. Resilient Distribute versucht dabei so weit wie möglich die Schnittstelle des klassischen Distribute zu erhalten und auch die Algorithmen im Kern nicht zu ändern. Eine Umsetzung einfacher Kommunikation auf Basis einer einfachen Form von Resilient Distribute wurde ebenfalls diskutiert sowie die Einbettung in die gesamte Realisierung der Mushu-Architektur.

13. Subprotokolle: Heuristische Beschleunigung von Mustererkennung

Auf der untersten Ebene der Mushu-Architektur finden sich die Subprotokolle. Sie können – als Bausteine eingesetzt – umfangreiche Beiträge zum Ablauf von Agentenprotokollen leisten. Bei der Analyse einer Szene werden auf Basis von Wissen bzw. Kontextinformationen und Beobachtungen Modelle der Szene erzeugt. Diese Modelle sagen bestimmte Eigenschaften voraus und können an einzelnen Bereichen im Bildmaterial bestätigt werden. Für andere Regionen lassen sich so Hypothesen aufstellen. Die adressierten Modelle könnten konkret auch in der Form von Petri- oder Referenznetzen dargestellt werden. Da hier jedoch der Umgang und die Simulation auf einer höheren Ebene betrachtet wird, wird der Modellbegriff abstrakt gehalten.

Der gesamte Prozess ist komplex und jeden einzelnen Schritt detailliert zu adressieren, würde den Rahmen der Arbeit übersteigen. Aus diesem Grund soll in diesem Kapitel exemplarisch ein Teilbereich betrachtet werden, welcher durch ein Subprotokoll abgebildet werden kann.

Die hier vom Autor der Arbeit beigetragenen und vorgestellten Ergebnisse wurden in den Veröffentlichungen (Röwekamp und Haustermann, 2015) und (Röwekamp, 2016) vorgestellt. Die Konzepte wurden mit dem Fachpublikum der ICPR (International Conference on Pattern Recognition), sowie dem PNSE (Petri Nets and Software Engineering) Workshop diskutiert. Die Inhalte dieses Kapitels orientieren sich in großen Teilen an den genannten Veröffentlichungen.

13.1. Subprotokolle und Plattformfunktionalitäten

Die zu Beginn des Kapitels beschriebenen Bestandteile entsprechen Protokollen bzw. dem Durchgriff auf die Wissensbasis des Agenten. Dies gilt insbesondere in der Annahme, dass Agenten in Analogie zur realen Welt kontinuierlich ihre Umwelt interpretieren. Auf diesem Weg erhalten Agenten abstrakt gesehen die Möglichkeit sich ihrer Umgebung anzupassen und mit ihr geeignet zu interagieren.

Auf den ersten Blick könnten Plattformfunktionalitäten und Subprotokolle verwechselt werden, da sie beide Funktionalitäten auf einer sehr niedrigen Ebene anbieten bzw. modellieren. Der Unterschied liegt hierbei jedoch im Detail: Während Subprotokolle Fähigkeiten von Agenten abbilden, welche diese besitzen können, oder auch nicht, stehen Plattformfunktionalitäten stets allen Agenten und der Plattform selbst zur Verfügung. Sie sind damit – und auch auf Basis der Mushu-Architektur – Bestandteile der Plattform, auf der sich ein Agent befindet. Zieht der Agent um, ist es möglich, dass der Zugriff auf Plattformfunktionalitäten verloren geht, falls die neue Plattform diese nicht anbietet. Der Zugriff auf Subprotokolle ist jedoch nicht von einem Umzug betroffen.

Um das Konzept zu verdeutlichen, kann ein reales Beispiel betrachtet werden. Bei einer Konferenz unterhalten sich zwei Konferenzteilnehmer in einem Konferenzraum und ein dritter ist dort anwesend. Sie verwenden dabei eine Sprache, die für einen der beiden eine Fremdsprache ist und von dem Dritten nicht verstanden wird. Die Fähigkeit einzelne grammatikalische Strukturen aufzunehmen entspricht einem Subprotokoll, die Interpretation des gehörten Satzes einem Protokoll.

Während der erste Sprecher auf eine Vielzahl von geeigneten Subprotokollen zugreifen kann, kann der zweite Sprecher nur auf einige äquivalente Subprotokolle zugreifen. Der dritte Sprecher besitzt keine derartigen Subprotokolle und kann kein Modell aus den gesprochen Sätzen ableiten. Alle Teilnehmer behalten ihre Subprotokolle, auch wenn der Konferenzraum gewechselt wird.

Eine Plattformfunktionalität hingegen wäre in diesem Beispiel etwa die Raumbeleuchtung, welche von jedem der Anwesenden betätigt werden kann. Eine Nutzung kann erforderlich sein, wenn es dunkel ist, und etwas gezeigt werden soll. Die Plattformfunktionalität kann in den Protokollablauf integriert werden, ist und bleibt jedoch Teil der Plattform. Die Plattform kann ebenfalls auf die Funktionalität zugreifen, indem beispielsweise das Licht mit einer Zeitschaltuhr eingeschaltet wird ohne Einflussnahme der Konferenzteilnehmer.

Unter diesem Gesichtspunkt liegt es nahe, Teile der Interpretation von Szenen als Subprotokoll zu modellieren. Wie im Beispiel angedeutet, handelt es sich bei Subprotokollen um sehr elementare Dinge. Wird das Konzept nun auf den Ablauf bei der Analyse von Szenen bezogen, bedeutet dies zwangsläufig eine Auseinandersetzung mit mathematischen Grundlagen von dort angewandten Mechaniken und Prozeduren. Im folgenden Abschnitt wird dies weiter ausgeführt und die in diesem Kapitel getroffene Auswahl der Betrachtung motiviert.

13.2. Ausgangspunkt und Motivation

In vielen Analyseaufgaben spielen Algorithmen für die Mustererkennung eine zentrale Rolle. Die Forschung im Bereich der Mustererkennung fokussiert im Wesentlichen auf die Machbarkeit der Methode und legt Wert auf Qualität der Ergebnisse. Dies geschieht beispielsweise durch Angabe von Genauigkeit (englisch »precision«) und Trefferquote (englisch »recall«) auf einer bestimmten Beispielmenge an Bildern bzw. Sequenzen. Auf der anderen Seite existieren Echtzeitansätze, wie sie beispielsweise in Anwendung der Robotik notwendig sind. Diese Algorithmen basieren auf scharfen Anforderungen bezüglich der verfügbaren Zeit, aber auch der verfügbaren Rechen- und Batterieleistung. Sie erfolgen somit meist unter Berücksichtigung von vielfachen Ressourceneinschränkungen.

Was bei der skalierenden Simulation von Agenten jedoch wünschenswert wäre, wäre ein Mittelweg, der die Berechnung zwar beschleunigt, jedoch ohne die scharfen Einschränkungen der Echtzeitanwendung auskommt. Dies merkte auch einer der Gutachter des eingehend zitierten Beitrags des Autors (Röwekamp, 2016) an: » *This is a well-designed and well-documented algorithm and should be accepted for publication. Very few people focus on lowering the computational complexity of algorithms in the computer vision field.*« (Review 2993 zu Submission 1376, ICPR 2016) (dt.: Dies ist ein gut entworfener und gut dokumentierter Algorithmus, der für die Veröffentlichung akzeptiert werden sollte. Sehr wenige Leute fokussieren sich auf die Verringerung der Berechnungskomplexität von Algorithmen im Bereich der Bildverarbeitung.)

Gleichzeitig sollte durch eine Verbesserung möglichst gleich eine Vielzahl von Algorithmen betroffen sein. Während dies eine wünschenswerte Eigenschaft ist, ist es jedoch in der Regel komplizierter, allgemeine Verbesserungen für einen Großteil oder alle Algorithmen gemeinsam zu konzipieren. Daher soll an dieser Stelle eine spezielle Unterkategorie von Mustererkennungsalgorithmen betrachtet werden.

13.2.1. Featurebasierte Algorithmen und SIFT

Eine Vielzahl von Mustererkennungsalgorithmen wurde über die Zeit von der wissenschaftlichen Community vorgestellt, wovon viele featurebasiert arbeiten. Diesen speziellen Algorithmen ist gemein, dass sie keine Vergleiche auf den direkten Bilddaten anstellen, sondern die Umgebung von bestimmten Punkten des Bildes in einen Vektor im Featureraum transformieren und diese Vektoren vergleichen. Die dabei entstehenden Vektoren weisen im Normalfall eine große Zahl an Dimensionen auf und sind daher mit herkömmlichen Mitteln vergleichsweise ineffizient zu vergleichen. Wäre es also möglich, den Abgleich der Vektoren zu beschleunigen, wäre eine verwendbare Grundlage für viele dieser Algorithmen geschaffen.

Zum besseren Verständnis soll daher ein entsprechender Algorithmus herausgegriffen werden, welcher näher betrachtet wird: Scale Invariant Feature Transform (SIFT).

Mit SIFT können Bildausschnitte bzw. Objekte in anderen Bildern wiedererkannt werden unabhängig von ihrer Größe, Rotation und zu einem gewissen Grad auch von ihrer Rotation im 3-Dimensionalen, wobei dort nur um wenige Grad. Zusätzlich ist SIFT robust gegenüber Verdeckung des Objektes und toleriert je nach Situation große Mengen an Verdeckung. SIFT gilt daher als sehr robuster Algorithmus. SIFT geht zurück auf die Veröffentlichungen (Lowe, 1999, 2004).

An dieser Stelle sei noch einmal auf den Unterschied zwischen der in SIFT referenzierten »Skalierung« (S in SIFT) und dem Skalierungsbegriff, wie er innerhalb dieser Arbeit verwendet wird, hingewiesen. »Skalierung« in SIFT bezieht sich auf die Gausssche Skalierung, bei der ein Bild mit unterschiedlich starken Weichzeichnern, welche jeweils auf der Gausschen Glockenkurve basieren, bearbeitet wurde. Für die Definition von Skalierbarkeit im Rahmen der Arbeit siehe Abschnitt 2.6.

SIFT besteht aus zwei Hauptbestandteilen: Der Extraktion von Features mit Übersetzung in den sog. Feature-Space, sowie einer nächste-Nachbarn-Suche innerhalb des Feature-Spaces. Der Vergleich von Vektoren tritt im zweiten Bestandteil auf. Zum Verständnis wird jedoch zunächst der erste Teil kurz angerissen und insbesondere der hochdimensionale Feature-Space erläutert.

SIFT ist eine Kombination aus Feature-Algorithmus und nächste Nachbar Suche. Zunächst werden im Bild »interessante« Punkte (»Features«) berechnet. Die genaue Berechnung spielt für den Einsatz an dieser Stelle keine Rolle. In der direkten Umgebung der Features werden nun die Gradienten (Farbverläufe) mit ihrer Stärke und Richtung als Deskriptoren gespeichert. Dabei entsteht eine Wertemenge, die recht genau die Charakteristik der Umgebung des Features beschreibt. Werden diese Werte nun gemeinsam als ein Vektor betrachtet, ergibt sich ein Vektor in einem hochdimensionalen Raum, dem Feature-Space. Beispielsweise besitzt ein SIFT-Featurevektor 128 Dimensionen. Um nun die Umgebungen der Points of interest eines Eingabebilds mit denen in einer Datenbank zu vergleichen, genügt es einige nächste-Nachbar-Suchen mit Features des Eingabebildes im hochdimensionalen Feature-Space auszuführen. Einfache Beschleunigungsverfahren für diese Suche existieren, wie beispielsweise in (Celebi, Celiker und Kingravi, 2011) oder (Marukatat und Methasate, 2013). Keine dieser Verfahren ist aber explizit auf diesen Anwendungsfall ausgelegt.

13.2.2. Die Idee

Eine konkrete Anwendung für den Einsatz solcher featurebasierten Mustererkennungsalgorithmen ist die invertierte Suche auf einer Bilddatenbank mithilfe eines

Bildausschnittes. Das gewünschte Suchergebnis ist dabei das Gesamtbild, welches den Ausschnitt enthält. Dabei lässt sich beobachten, dass – vorausgesetzt der Algorithmus selbst arbeitet mit ausreichender Genauigkeit und Trefferquote – die überwiegende Zahl an Bildern der Bilddatenbank *keine* Treffer generieren sollten. Der Vergleich von Vektoren basiert im Wesentlichen auf der Berechnung einer Differenz der Vektoren und der Bestimmung der (euklidischen) Distanz dieser Differenz vom Ursprung oder anders formuliert der Vektorlänge. Die Distanz berechnet sich durch die Summe aller Quadrate, welche aufsummiert werden und aus der Summe die Wurzel gezogen wird. Das resultierende Ergebnis wird dann gegen einen Schwellenwert abgeglichen. Im skizzierten Szenario kann davon ausgegangen werden, dass dieser Schwellenwert vielfach überschritten wird, da die überwältigende Zahl an Bildern in der Bilddatenbank keine Übereinstimmung generieren wird.

Zahlen werden durch entsprechende Bits im Speicher repräsentiert, und beim gegebenen Anwendungsfall fällt auf, dass insbesondere die höherwertigen Bits für den Abgleich gegen den Schwellenwert relevant sind.

Bei der Berechnung der einzelnen Bits eines einzelnen Quadrates einer Zahl sind sehr viele Bits der Originalzahl direkt oder indirekt an der Berechnung beteiligt, sodass die vollständige Multiplikation durchgeführt werden muss, bevor eindeutige Aussagen bezüglich der jeweiligen Bits des Quadrats getätigt werden können. Somit können die höherwertigen Bits nur schlecht im Vorfeld zusammenorganisiert werden, ohne die Multiplikationsoperation zunächst auszuführen. Gäbe es nun also eine Operation, bei der eine direkte Zuordnung von Bits der Originalzahl zu Bits im Quadrat möglich wäre, könnte unter Umständen die gesamte Multiplikationsoperation entfallen und direkt mit den Originalzahlen gerechnet werden.

Dieser Ausgangsgedanke ist maßgeblich für die Konstruktion der in diesem Kapitel vorgestellten Methode.

13.3. Binary Squaring

Binary Squaring ist der Name der im Rahmen der Arbeit entwickelten Rechenoperation, die in bestimmten Berechnungen als heuristische Alternative zu herkömmlichen Multiplikationen verwendet werden kann. Binary Squaring setzt damit bei der Grundlage vieler Erkennungsalgorithmen an und kann die Ausführungsgeschwindigkeit von Matching Algorithmen (wie z.B. SIFT) deutlich beschleunigen.

Nach Voraussetzung soll jedes Bit der Ergebniszahl nur von exakt einem Bit der Ursprungszahl abhängen. Zusätzlich soll ein Quadrat möglichst genau abgebildet

werden. Aus diesen Gründen liegt es nahe, direkt die einzelnen Bits separat zu quadrieren, anstatt der gesamten Zahl.

Im Rahmen erster empirischer Beobachtungen bei der Anwendung auf Differenzen von Feature-Vektoren wurde diese Methode im Rahmen der Arbeit untersucht. Dabei fiel schnell auf, dass sich der Distanzwert der neuen Berechnung und der der klassischen Variante fast immer um einen Faktor, der in etwa zwei entsprach, unterschieden. Dieser Faktor ist nicht offensichtlich, da er in der Konstruktion der neuen Operation nirgendwo eine Rolle spielt. Um diese Auffälligkeit formal zu untersuchen, soll der neuen Operation ein ebenso formaler Rahmen gegeben werden.

Definition von Binary Squares

Um das Konzept von Binary Squares sinnvoll einzuführen, bietet es sich an, eine formale Sicht auf die Darstellung von Zahlen zu einzunehmen. Dafür soll zunächst die Binärrepräsentation einer Zahl definiert werden:

Satz 13.1. Für alle $n \in \mathbb{N}$ existiert ein $K \subseteq \mathbb{N}$ sodass $n = \sum_{k \in K} 2^k$.

Auf dieser Basis kann nun das (»Binary Square«) (dt.: Binärquadrat) definiert werden:

Definition 13.2. Das *Binary Square* einer Zahl $n \in \mathbb{N}$ mit $n = \sum_{k \in K} 2^k$ ist definiert als:

$$bsq(n) := \sum_{k \in K} 2^{2k}$$

Das Binary Square einer natürlichen Zahl x wird fortan als $bsq(x)$ bezeichnet. Somit ergibt sich beispielsweise $bsq(10) = bsq([1010]_2) = ([0010]_2)^2 + ([1000]_2)^2 = [1000100]_2 = 68$ im Gegensatz zu $10^2 = 100$. Es existieren jedoch auch Gleichheiten. So gilt beispielsweise, wenn die Zahl selbst nur aus einer Zweierpotenz besteht: $bsq(16) = 16^2 = 256$. Generell ist ein Binary Square mindestens ein Drittel so groß (bei Zahlen der Form $2^n - 1$) wie das Quadrat und maximal identisch bei Zweierpotenzen[1]. Eine beispielhafte Darstellung der Quadrate und Binary Squares der Zahlen 0 bis 4096 mit logarithmischer Skala auf der Ordinate findet sich in Abbildung 13.1.

[1] Auf einen Beweis hierzu sei an dieser Stelle verzichtet, da es sich zwar um eine interessante Beobachtung handelt, die jedoch keine direkte Bewandtnis in den weiteren Betrachtungen im Rahmen der Arbeit hat.

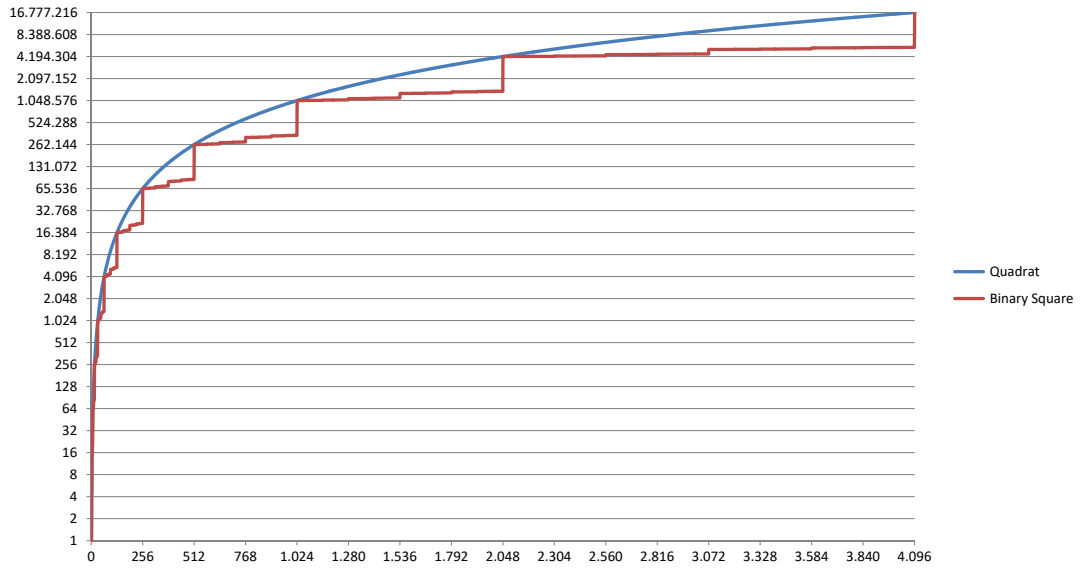

Abbildung 13.1.: Binary Squares und Quadrate der Zahlen 0 bis 4096. Die Ordinate weist dabei eine logarithmische Skala auf.

13.4. Zusammenhang zwischen Binary Squares und Quadraten

Auf den ersten Blick scheint kein großer bzw. hilfreicher Zusammenhang zwischen Binary Squares und herkömmlichen Quadraten zu bestehen. Tatsächlich offenbart sich dieser jedoch erst bei der Betrachtung größerer Summen von Quadraten bzw. Binary Squares. Zu diesem Zweck ist es hilfreich die Anzahl der folgenden Nullen (engl.: trailing zeroes) innerhalb der Binärdarstellung der Zahl zu betrachten. Diese gibt einen wesentlichen Aufschluss auf die Struktur des Binary Squares und wird als zentraler Stützpfeiler der Herleitung der Zusammenhänge dienen.

Diese Zahl der folgenden Nullen der Binärdarstellung soll im Folgenden als $t(n)$ zur Zahl n bezeichnet werden. In der Tat ist diese Funktion in der Mathematik nicht unbekannt. Sie trägt dort den Namen »2-adische Bewertung« (engl.: »2-adic valuation«) oder auch »Linealsequenz« (engl. »ruler sequence«). $t(n)$ wird in der mathematischen Literatur auch als $\nu_2(n)$ notiert, siehe beispielsweise (Amdeberhan, Manna und Moll, 2008). $t(n)$ ist nur schwer in geschlossener Form darzustellen, einige Ausnahmen existieren dabei für n der Form $n = k!$ für ein $k \in \mathbb{N}$ (Legendre, 1830). Da diese im hier betrachteten Fall jedoch als nicht ausreichend zu sehen sind, erfolgt die Definition über die trigonometrische Funktion Cosinus.

Tabelle 13.1.: Die Werte von $t(n)$ für $1 \leq n \leq 20$

n	$t(n)$	n	$t(n)$	n	$t(n)$	n	$t(n)$	n	$t(n)$
1	0	5	0	9	0	13	0	17	0
2	1	6	1	10	1	14	1	18	1
3	0	7	0	11	0	15	0	19	0
4	2	8	3	12	2	16	4	20	2

Die Grundidee basiert dabei darauf verschiedene Cosinusfunktionen mit Periode 2^x für ein $x \in \mathbb{N}$ miteinander zu multiplizieren, sodass sie bei Teilbarkeit von n durch eine bestimmte Zweierpotenz 1 als Wert ergeben und 0 sonst. Aufsummiert ergibt sich somit für jede Teilbarkeit durch eine Zweierpotenz ein um 1 größerer Wert. Somit bildet die Summe genau $t(n)$ ab.

Definition 13.3. Die Funktion $t(n)$ ist für $n \in \mathbb{N}_+$ definiert als:

$$t(n) := \sum_{k=1}^{\infty} cos(\frac{1}{2^k}\pi n) \prod_{j=1}^{k} cos(\frac{1}{2^j}\pi n)$$

13.4.1. Herleitung des Zusammenhangs

Lemma 13.4. $t(n)$ entspricht der Anzahl der folgenden Nullen der Binärdarstellung von n.

Beweis. Durch Induktion über die Anzahl z an folgenden Nullen in der Binärdarstellung von n.

Anfang: Sei $m \in \mathbb{N}_+$ mit $m \equiv 1(mod\ 2)$ eine ungerade Zahl und somit $z = 0$. Es folgt, dass $\frac{m}{2} = k + \frac{1}{2}$ für ein $k \in \mathbb{N}$ und somit, dass $cos(\frac{1}{2}m\pi) = cos(k\pi + \frac{\pi}{2}) = cos(\frac{\pi}{2}) = 0$. Da das Produkt innerhalb der Definition von $t(n)$ stets den Faktor $cos(\frac{1}{2}\pi n)$ enthält, folgt direkt: $t(m) = 0 = z$.

Annahme: Für ein $r \in \mathbb{N}$ gilt, dass $t(r)$ der Anzahl der folgenden Nullen der Binärdarstellung von z entspricht.

Schritt: Sei $m = 2r$. Dabei lässt sich beobachten, dass die Anzahl an folgenden Nullen in m gleich $z + 1$ ist. Darüber hinaus ist dies die einzige Möglichkeit, um die Anzahl der folgenden Nullen um eins zu erhöhen, während alle anderen Bits der Zahl unverändert bleiben. Sei $V = \{v \in \mathbb{N} | v \equiv 1(mod\ 2)\}$ die Menge aller

ungeraden natürlichen Zahlen. Nach z Summanden der Summe in der Definition von $t(r)$ enthält das Produkt in jedem weiteren Summanden stets den Ausdruck $cos(\frac{1}{2^{z+1}}\pi r) = cos(\frac{1}{2}\pi v) = 0$ für ein $v \in V$. Somit ist jeder folgende Summand gleich 0. Für $m = 2r$ tritt dieser Effekt einen Summand später auf (ab $z+2$), während alle anderen Summanden identisch bleiben[2]. Folglich unterscheidet sich die Summe lediglich in einem Summanden, bei dem $k = z+1$ gilt. Mit einem $v \in V$ folgt somit:

$$\begin{aligned} t(m) &= t(r) + cos(\frac{1}{2^{z+1}}\pi m) \prod_{j=1}^{z+1} cos(\frac{1}{2^j}\pi m) \\ &= t(r) + cos(\frac{1}{2^{z+1}}\pi m)^2 \prod_{j=1}^{z} cos(\frac{1}{2^j}\pi m) \\ &= t(r) + cos(v\pi)^2 \prod_{j=2}^{z+1} cos(\frac{1}{2^j}2\pi m) \\ &= t(r) + (-1)^2 \prod_{j=2}^{z+1} 1 = t(r) + 1 \end{aligned}$$

□

Lemma 13.5. Sei $n \in \mathbb{N}_+$. Es gilt:

$$bsq(n+1) = bsq(n) + \frac{1}{3}(1 + 2^{2t(n+1)+1})$$

Beweis. Sei $k \in \mathbb{N}_+$. Eine Addition von 1 zu k betrifft alle Bits durch die der Übertrag der Operation hindurchgezogen wird. Dabei werden all diese Bits nach der Addition zu 0 und das Bit, welches den Übertrag gestoppt hat, wird zu 1. Folglich ist die Anzahl der betroffenen Bits in $k+1$ genau $t(k+1)$.

Auf dieser Basis soll nun $bsq(k)$ und $bsq(k+1)$ betrachtet werden. Relevant für die Betrachtung ist vor allem der Summand der rechten Seite des Lemmas, welcher ebenfalls durch die Differenz von $bsq(k+1) - bsq(k)$ berechnet werden kann. In einem Binary Square sind alle Bits auf die doppelte Wertigkeit geshifted (Definition eines Binary Squares), daher weisen alle Bits, welche für einen ungeraden Exponenten stehen (Im Folgenden als »ungerade Bits« bezeichnet), in jedem Fall eine 0 als Wertigkeit auf. Wird nun die Differenz von $bsq(k+1) - bsq(k)$ berechnet,

[2] Dieser Effekt entsteht dadurch, dass in beiden Versionen der Summe (sowohl mit m, als auch mit r) jeder Faktor von jedem betrachteten Summanden zu $cos(x2\pi)$ umgeschrieben werden kann für ein $x \in \mathbb{N}$. Da jede Wahl für x zu $cos(x2\pi) = 1$ führt, sind die Summanden identisch.

hilft eine Betrachtung der Operation in einem Binärdarstellungssystem, welches die Verwendung eines Zweierkomplements ermöglicht. Dabei werden alle Bits in $bsq(k)$ invertiert und es wird 1 hinzuaddiert. Da allgemein $bsq(k+1) > bsq(k)$ gilt, kann die Summe $bsq(k+1) + \overline{bsq(k)}$ anstatt der Differenz $bsq(k+1) - bsq(k)$ betrachtet werden, ohne danach für weitere Betrachtungen auf die Einschränkung auf ein entsprechendes Darstellungssystem für Binärzahlen angewiesen zu sein.

Da ein Binary Square niemals eine Übertragskette generiert (da jedes zweite Bit immer 0 ist), genügt es lediglich die entsprechenden $t(k+1)$ Bits in k und die entsprechend zugehörigen Bits in $bsq(k)$ zu betrachten. Das Invertieren setzt alle ungeraden Bits auf die Wertigkeit 1, während $bsq(1) = 1$ gilt. Es folgt somit:

$$\begin{aligned}
bsq(k+1) - bsq(k) &= bsq(1) + \sum_{i=1}^{t(k+1)} 2^{2i-1} = 1 + \frac{1}{2} \sum_{i=1}^{t(k+1)} 4^i \\
&= 1 + \frac{1}{2} \sum_{i=0}^{t(k+1)-1} 4^{i+1} = 1 + 2 \sum_{i=0}^{t(k+1)-1} 4^i \\
&= 1 + 2\frac{4^{t(k+1)} - 1}{4-1} = 1 + \frac{2 \cdot 4^{t(k+1)} - 2}{4-1} \\
&= \frac{3}{3} + \frac{2 \cdot 4^{t(k+1)} - 2}{3} = \frac{1}{3}(1 + 2 \cdot 2^{2t(k+1)}) \\
&= \frac{1}{3}(1 + 2^{2t(k+1)+1})
\end{aligned}$$

Daraus folgt für beliebige $n \in \mathbb{N}_+$ direkt $bsq(n+1) = bsq(n) + \frac{1}{3}(1+2^{2t(n+1)+1})$ □

Mit diesen Lemmata kann nun eine alternative Darstellung für die Funktion $bsq()$ gegeben werden:

Satz 13.6. Für $n \in \mathbb{N}_+$ gilt:

$$bsq(n) = 1 + \frac{n-1}{3} + \frac{2}{3} \sum_{i=2}^{n} 4^{t(i)}$$

Beweis. Beachte, dass $bsq(1) = bsq(2^0) = 2^{2 \cdot 0} = 1$. Durch Abwickeln der rekursiven Gleichung aus Lemma 13.5 ergibt sich für $n \in \mathbb{N}_+$:

$$\begin{aligned} bsq(n) &= bsq(1) + \frac{1}{3}(1 + 2^{2t(1+1)+1}) + \frac{1}{3}(1 + 2^{2t(2+1)+1}) + \\ &\quad ... + \frac{1}{3}(1 + 2^{2t(n-1+1)+1}) \\ &= bsq(1) + \sum_{i=2}^{n} \frac{1}{3}(1 + 2^{2t(i)+1}) = 1 + \frac{1}{3}\sum_{i=2}^{n}(1 + 2^{2t(i)+1}) \\ &= 1 + \frac{n-1}{3} + \frac{1}{3}\sum_{i=2}^{n}(2^{2t(i)+1}) = 1 + \frac{n-1}{3} + \frac{2}{3}\sum_{i=2}^{n} 4^{t(i)} \end{aligned}$$

Die Korrektheit der Abwicklung kann mittels Induktion leicht gezeigt werden: *Anfang:* Sei $n = 1$. Es folgt:

$$bsq(1) = 1 = 1 + \frac{0}{3} + 0 = 1 + \frac{1-1}{3} + \frac{2}{3}\sum_{i=2}^{1} 4^{t(i)}$$

Annahme: Für ein beliebiges, aber fest gewähltes $n \in \mathbb{N}_+$ gilt bereits:

$$bsq(n) = 1 + \frac{n-1}{3} + \frac{2}{3}\sum_{i=2}^{n} 4^{t(i)}$$

Schritt: Betrachte $n + 1$. Es gilt nach Lemma 13.5 und Induktionsannahme:

$$\begin{aligned} bsq(n+1) &= bsq(n) + \frac{1}{3}(1 + 2^{2t(n+1)+1}) \\ &= \left(1 + \frac{n-1}{3} + \frac{2}{3}\sum_{i=2}^{n} 4^{t(i)}\right) + \frac{1}{3}(1 + 2^{2t(n+1)+1}) \\ &= \left(1 + \frac{n-1}{3} + \frac{2}{3}\sum_{i=2}^{n} 4^{t(i)}\right) + \frac{1}{3} + \frac{2}{3}4^{t(n+1)} \\ &= 1 + \frac{n-1}{3} + \frac{1}{3} + \frac{2}{3}4^{t(n+1)} + \frac{2}{3}\sum_{i=2}^{n} 4^{t(i)} \\ &= 1 + \frac{n}{3} + \frac{2}{3}\sum_{i=2}^{n+1} 4^{t(i)} \end{aligned}$$

□

Aus dieser Formel kann nun eine weitere Formel hergeleitet werden, welche die Summe aller Binary Squares zwischen zwei Zweierpotenzen beschreibt.

Satz 13.7. Für $r \in \mathbb{N}_+$ gilt:

$$\sum_{i=2^r}^{2^{r+1}-1} bsq(i) = \frac{7}{6}2^{3r} - \frac{1}{6}2^r$$

Beweis. Der Beweis zu diesem Satz ist umfangreich. Daher soll dieser in mehreren Schritten erfolgen. Zunächst kann Satz 13.6 auf die linke Seite der Gleichung angewandt werden:

$$\sum_{i=2^r}^{2^{r+1}-1} bsq(i) = \sum_{i=2^r}^{2^{r+1}-1} \left(1 + \frac{i-1}{3} + \frac{2}{3}\sum_{j=2}^{i} 4^{t(j)}\right)$$

Auf dieser Basis kann die Summenformel zerteilt werden, um so die einzelnen Summanden separat zu betrachten:

$$\begin{aligned}\sum_{i=2^r}^{2^{r+1}-1} \left(1 + \frac{i-1}{3} + \frac{2}{3}\sum_{j=2}^{i} 4^{t(j)}\right) &= \sum_{i=2^r}^{2^{r+1}-1} (1) + \sum_{i=2^r}^{2^{r+1}-1} \left(\frac{i-1}{3}\right) + \sum_{i=2^r}^{2^{r+1}-1} \left(\frac{2}{3}\sum_{j=2}^{i} 4^{t(j)}\right) \\ &= 2^r \qquad + s_1(r) \qquad + s_2(r)\end{aligned}$$

Im Folgenden können somit die einzelnen Summanden $s_1(r)$ und $s_2(r)$ bestimmt werden.

$$\begin{aligned}s_1(r) &= \sum_{i=2^r}^{2^{r+1}-1} \left(\frac{i-1}{3}\right) = \frac{1}{3}\sum_{i=2^r}^{2^{r+1}-1} (i-1) = \frac{1}{3}\sum_{i=2^r-1}^{2^{r+1}-2} (i) \\ &= \frac{1}{3}\left(\sum_{i=1}^{2^{r+1}-2} i - \sum_{i=1}^{2^r-2} i\right) = \frac{1}{3}\left(\frac{(2^{r+1}-2)(2^{r+1}-1)}{2} - \frac{(2^r-2)(2^r-1)}{2}\right) \\ &= \frac{1}{3}\left(\frac{\left(\left(2^{r+1}\right)^2 - 2^{r+1}\cdot 1 - 2\cdot 2^{r+1} + 2\right) - \left(2^{2r} - 2^r - 2\cdot 2^r + 2\right)}{2}\right) \\ &= \frac{1}{3}\left(\frac{\left(4\cdot 2^{2r} - 4\cdot 2^r - 2^{2r} + 2^r\right)}{2}\right) = \frac{1}{3}\left(\frac{\left(3\cdot 2^{2r} - 3\cdot 2^r\right)}{2}\right) \\ &= \frac{1}{3}\frac{3}{2}\left(2^{2r} - 2^r\right) = \frac{1}{2}\left(2^{2r} - 2^r\right) = 2^{2r-1} - 2^{r-1}\end{aligned}$$

Die Bestimmung des Summanden $s_2(r)$ ist durch das Vorkommen der Funktion $t(n)$ deutlich komplexer als die Bestimmung von $s_1(r)$. Die Kernidee bei der

Umformung besteht darin die doppelte Summe in

$$s_2(r) = \sum_{i=2^r}^{2^{r+1}-1} \left(\frac{2}{3} \sum_{j=2}^{i} 4^{t(j)} \right) = \frac{2}{3} \sum_{i=2^r}^{2^{r+1}-1} \left(\left(\sum_{j=1}^{i} 4^{t(j)} \right) - a^{t(1)} \right)$$
$$= \frac{2}{3} \left(\sum_{i=2^r}^{2^{r+1}-1} \left(\sum_{j=1}^{i} 4^{t(j)} \right) - \sum_{i=2^r}^{2^{r+1}-1} a^0 \right) = \frac{2}{3} \left(\sum_{i=2^r}^{2^{r+1}-1} \left(\sum_{j=1}^{i} 4^{t(j)} \right) - 2^r \right)$$

entsprechend der möglichen Werte von $t(n)$ umzusortieren. Während in der Definition von $t(n)$ eine Reihe zum Einsatz kommt, ist diese nur für beliebige Eingabegrößen notwendig. Durch die Eingrenzung auf den Bereich $2^r \leq n \leq 2^{r+1} - 1$ bestehen nur endlich viele mögliche Werte für $t(n)$ und es genügt die Betrachtung einer geeigneten Partialsumme.

Durch geschickte Betrachtung der Bedeutung von $t(n)$ (die Abbildung von folgenden Nullen), kann beobachtet werden, dass $t(n)$ maximal wird, wenn n selbst eine Zweierpotenz ist. n kann für den Bereich $2^r \leq n \leq 2^{r+1} - 1$ nur eine Zweierpotenz als Wert annehmen: 2^r mit $t(2^r) = r$. Diese kommt in der doppelten Summe insgesamt 2^r mal vor, sodass diese in jedem Fall einen Summanden $2^r \cdot 4^{t(2^r)} = 2^r \cdot 4^r$ enthält.

Die verbleibenden Werte t von $t(n)$ ergeben sich weniger offensichtlich. Zur Bestimmung der umsortierten Summanden bietet es sich an, die Summe in die Bereiche vor 2^r und nach 2^r zu zerlegen. Dies liegt darin begründet, dass die davorliegenden Summanden im Gegensatz zu denen zwischen 2^r und $2^{r+1} - 1$ jeweils gleich oft in der Summe vorkommen, nämlich 2^r mal (da die vordere Summe 2^r Summanden aufweist).

$$\sum_{i=2^r}^{2^{r+1}-1} \left(\sum_{j=1}^{i} 4^{t(j)} \right) = \left(2^r \sum_{j=1}^{2^r-1} 4^{t(j)} \right) + \sum_{i=2^r}^{2^{r+1}-1} \left(\sum_{j=2^r}^{i} 4^{t(j)} \right)$$

Bei Betrachtung des hinteren Ausdrucks fällt auf, dass der Wert $j = 2^r$ in der gesamten Summe genau 2^r mal vorkommt, der Wert $j = 2^r + 1$ genau $2^r - 1$ mal, usw. Schlussendlich hat der Wert $j = 2^r + (2^r - 1) = 2^{r+1} - 1$ nur $2^r - (2^r - 1) = 1$ Vorkommen. Werden nun die Werte für $t(j)$ betrachtet, so ergibt sich, dass $t(j) = 0$ insgesamt für $j = 2^{r+1} - 1$, $j = 2^{r+1} - 3$, $j = 2^{r+1} - 5$, ..., $j = 2^r + 1$ jeweils $1x$, $3x$, $5x$, ..., $(2^r - 1)x$ vorkommt. Analog verhält es sich mit $t(j) = 1$, nur dass hierbei das erste Vorkommen bei $j = 2^{r+1} - 2$ liegt und die die Distanz zwischen den Vorkommen bei 4. $t(j) = 2$ tritt zum ersten mal bei $j = 2^{r+1} - 4$ auf und die Abstände zwischen den Vorkommen betragen 8 Werte. Auf diese Weise kann bis $t(j) = r$ verfahren werden, welches wie oben beschrieben die Sondersituation aufweist, dass es nur bei $j = 2^r$ selbst auftritt.

Werden die Häufigkeiten der Vorkommen betrachtet (beispielsweise $1x$, $3x$, $5x$, ..., $(2^r-3)x$, $(2^r-1)x$), fällt auf, dass die Häufigkeiten von außen nach innen paarweise zusammenaddiert stets 2^r ergeben. Insgesamt tritt der entsprechende Summand im Falle von $t(j)=0$ nur bei jedem zweiten Wert für j auf und durch die paarweise Zusammenführung tritt der Wert $t(j)=0$ je 2^r mal und damit insgesamt $\frac{1}{2}\frac{1}{2}2^r = 2^{r-2}$ mal auf. Bei $t(j)=1$ bleibt die Halbierung durch die paarweise Addition bestehen, lediglich die Häufigkeit halbiert sich erneut und beträgt nun ein Viertel der Gesamtwerte. Es ergibt sich somit für $t(j)=1$, dass dieser Summand $\frac{1}{2}\frac{1}{4}2^r = 2^{r-3}$ mal vorkommt. Da $t(j)$ in dem Ausdruck $4^{t(j)}$ angewandt wird, muss dieser ebenfalls noch in die Multiplikation einfließen. Insgesamt ergibt sich somit (auch mit der obigen Betrachtung für den Fall $j=2^r$):

$$\sum_{i=2^r}^{2^{r+1}-1}\left(\sum_{j=2^r}^{i} 4^{t(j)}\right) = 2^r\cdot 4^r + \sum_{t=0}^{r-1} 2^r\cdot 2^{r-(t+2)}\cdot 4^t$$

Nach der gleichen Prozedur lassen sich die Summanden des Ausdrucks $2^r\sum_{j=1}^{2^r-1}4^{t(j)}$ umsortieren. Die Häufigkeit ist mit 2^r bereits bestimmt, sodass nur noch die Häufigkeit des Vorkommens des jeweiligen $t(j)$ Werts ausschlaggebend ist. Analog zum obigen Fall ergeben sich als Häufigkeiten für $t(j)=0$ insgesamt $\frac{1}{2}2^r = 2^{r-1}$ Vorkommen, für $t(j)=1$ ergeben sich $\frac{1}{4}2^r = 2^{r-2}$ Vorkommen, usw. Daraus folgt:

$$2^r\sum_{j=1}^{2^r-1}4^{t(j)} = \sum_{t=0}^{r-1} 2^r\cdot 2^{r-(t+1)}\cdot 4^t$$

Zusammengeführt folgt somit insgesamt für die Umsortierung:

$$\begin{aligned}
\sum_{i=2^r}^{2^{r+1}-1}\left(\sum_{j=1}^{i}4^{t(j)}\right) &= \left(2^r\sum_{j=1}^{2^r-1}4^{t(j)}\right) + \sum_{i=2^r}^{2^{r+1}-1}\left(\sum_{j=2^r}^{i}4^{t(j)}\right)\\
&= \left(\sum_{t=0}^{r-1}2^r\cdot 2^{r-(t+1)}\cdot 4^t\right) + 2^r\cdot 4^r + \sum_{t=0}^{r-1}\left(2^r\cdot 2^{r-(t+2)}\cdot 4^t\right)\\
&= \left(\sum_{t=0}^{r-1}2^r\cdot 2^{r-(t+1)}\cdot 4^t\right) + 2^r\cdot 4^r + \frac{1}{2}\sum_{t=0}^{r-1}\left(2^r\cdot 2^{r-(t+1)}\cdot 4^t\right)\\
&= \left(\frac{3}{2}\sum_{t=0}^{r-1}2^r\cdot 2^{r-(t+1)}\cdot 4^t\right) + 2^r\cdot 4^r\\
&= \left(\sum_{t=0}^{r-1}3\cdot 2^{r-1}\cdot 2^{r-(t+1)}\cdot 4^t\right) + 2^r\cdot 4^r
\end{aligned}$$

Nach der Umsortierung ergibt sich schlussendlich für $s_2(r)$:

$$\begin{aligned}
s_2(r) &= \frac{2}{3}\left(\sum_{i=2^r}^{2^{r+1}-1}\left(\sum_{j=1}^{i} 4^{t(j)}\right) - 2^r\right) \\
&= \frac{2}{3}\left(\left(\sum_{t=0}^{r-1} 3\cdot 2^{r-1}\cdot 2^{r-(t+1)}\cdot 4^t\right) + (2^r\cdot 4^r) - 2^r\right) \\
&= \frac{2}{3}\left(3\cdot 2^{r-1}\left(\sum_{t=0}^{r-1} 2^{r-(t+1)}\cdot 4^t\right) + (2^r\cdot 2^{2r}) - 2^r\right) \\
&= \frac{2}{3}\left(3\cdot 2^{r-1}\left(\sum_{t=0}^{r-1} \frac{2^r}{2}\frac{1}{2^t}\cdot 4^t\right) + (2^r\cdot 2^{2r}) - 2^r\right) \\
&= \frac{2}{3}\left(3\cdot 2^{r-1}2^{r-1}\left(\sum_{t=0}^{r-1} \left(\frac{4}{2}\right)^t\right) + (2^r\cdot 2^{2r}) - 2^r\right) \\
&= \frac{2}{3}\left(3\cdot 2^{2r-2}\left(\sum_{t=0}^{r-1} 2^t\right) + (2^r\cdot 2^{2r}) - 2^r\right) \\
&= \frac{2}{3}\left(3\cdot 2^{2r-2}\cdot\frac{1-2^r}{1-2} + 2^r\cdot 2^{2r} - 2^r\right) \\
&= \frac{2}{3}\left(-1\cdot 3\cdot 2^{2r-2}\left(1-2^r\right) + 2^r\cdot 2^{2r} - 2^r\right) \\
&= \frac{2}{3}\left(-\left(3\cdot 2^{2r-2}\right) + \left(3\cdot 2^{2r-2}\cdot 2^r\right) + 2^r\cdot 2^{2r} - 2^r\right) \\
&= \frac{2}{3}\left(3\cdot 2^{2r-2}\cdot 2^r - 3\cdot 2^{2r-2} + 2^r\cdot 2^{2r} - 2^r\right) \\
&= \frac{2}{3}\left(\frac{3}{4}\cdot 2^{3r} - \frac{3}{4}\cdot 2^{2r} + 2^{3r} - 2^r\right) \\
&= \frac{2}{3}\left(\frac{7}{4}\cdot 2^{3r} - \frac{3}{4}\cdot 2^{2r} - 2^r\right) \\
&= \frac{7}{6}\cdot 2^{3r} - \frac{1}{2}\cdot 2^{2r} - \frac{2}{3}\cdot 2^r
\end{aligned}$$

Nach der Bestimmung von $s_1(r)$ und $s_2(r)$ kann somit nun die gesamte Gleichung hergeleitet werden:

$$\begin{aligned}
\sum_{i=2^r}^{2^{r+1}-1} bsq(i) &= 2^r + s_1(r) + s_2(r) \\
&= 2^r + 2^{2r-1} - 2^{r-1} + \frac{7}{6}\cdot 2^{3r} - \frac{1}{2}\cdot 2^{2r} - \frac{2}{3}\cdot 2^r
\end{aligned}$$

$$= 2^r + \frac{1}{2} \cdot 2^{2r} - \frac{1}{2} \cdot 2^r + \frac{7}{6} \cdot 2^{3r} - \frac{1}{2} \cdot 2^{2r} - \frac{2}{3} \cdot 2^r$$
$$= \frac{7}{6} \cdot 2^{3r} - \frac{1}{6} \cdot 2^r$$

□

Lemma 13.8. Für $r \in \mathbb{N}_+$ gilt:

$$\sum_{i=2^r}^{2^{r+1}-1} i^2 = \frac{7}{3}2^{3r} - \frac{3}{2}2^{2r} + \frac{1}{6}2^r$$

Beweis. Der Beweis arbeitet mit Indexverschiebung und der Gaussschen Summenformel sowie ihrem quadratischen Äquivalent:

$$\begin{aligned}
\sum_{i=2^r}^{2^{r+1}-1} i^2 &= \sum_{i=0}^{2^{r+1}-1-2^r} (i+2^r)^2 \\
&= \sum_{i=0}^{2^{r+1}-1-2^r} (i^2) + \sum_{i=0}^{2^{r+1}-1-2^r} (2 \cdot i \cdot 2^r) + \sum_{i=0}^{2^{r+1}-1-2^r} (2^{2r}) \\
&= \sum_{i=0}^{2^r-1} (i^2) + 2^{r+1} \sum_{i=0}^{2^r-1} (i) + 2^{2r} \sum_{i=0}^{2^r-1} (1) \\
&= \frac{(2^r-1) \cdot 2^r \cdot (2^{r+1}-1)}{6} + 2^{r+1} \frac{(2^r-1) \cdot 2^r}{2} + 2^{2r} \cdot 2^r \\
&= \frac{1}{6}\left(2^{2r+r+1} - 2^{2r} - 2^{2r+1} + 2^r\right) + 2^r\left(2^{2r} - 2^r\right) + 2^{3r} \\
&= \frac{7}{3}2^{3r} - \frac{3}{2}2^{2r} + \frac{1}{6}2^r
\end{aligned}$$

□

Mit den zuvor beschrieben Ergebnissen kann das folgende, zentrale Theorem hergeleitet werden:

Theorem 13.9. Für $r \in \mathbb{N}_+$ gilt:

$$\lim_{r \to \infty} \frac{\sum_{i=2^r}^{2^{r+1}-1} i^2}{\sum_{i=2^r}^{2^{r+1}-1} bsq(i)} = 2$$

Beweis. Der Beweis erfolgt durch direktes Einsetzen von Satz 13.7 und Lemma 13.8

$$\lim_{r\to\infty} \frac{\sum_{i=2^r}^{2^{r+1}-1} i^2}{\sum_{i=2^r}^{2^{r+1}-1} bsq(i)} = \lim_{r\to\infty} \frac{\frac{7}{3}2^{3r} - \frac{3}{2}2^{2r} + \frac{1}{6}2^r}{\frac{7}{6}2^{3r} - \frac{1}{6}2^r}$$
$$= \lim_{r\to\infty} \frac{14\cdot 2^{3r} - 9\cdot 2^{2r} + 2^r}{7\cdot 2^{3r} - 2^r} = \lim_{r\to\infty} \frac{14 - 9\cdot\frac{1}{2^r} + \frac{1}{2^{2r}}}{7 - \frac{1}{2^{2r}}} = \frac{14}{7} = 2$$

□

13.4.2. Implikationen der theoretischen Ergebnisse

Das Ergebnis des Theorems 13.9 ist von zentraler Bedeutung. Sein Ergebnis gibt im Wesentlichen an, dass die Summe von Quadraten und Binary Squares innerhalb eines Intervalls zwischen zwei Zweierpotenzen gegen den Grenzwert von zwei geht mit steigender Zweierpotenz. In empirischen Untersuchungen kann beobachtet werden, dass der Faktor auch schon bei vergleichsweise geringen Größen wie beispielsweise 2^8 bereits sehr nah am Wert 2 liegt.

Dabei kann die einmalige Verwendung jeder Zahl zwischen zwei Grenzen als Idealfall betrachtet werden. Der entstehende Faktor ist auch bei einzelnen Doppelungen oder Auslassungen in vielen Fällen noch nah bei 2. Je mehr Werte aufsummiert werden, desto näher liegt der Faktor an der Zahl 2. Dies gilt dann, wenn die Werte zumindest einigermaßen gleich verteilt sind. Durch diese Ungenauigkeit wird das Verfahren zur Heuristik, deren Ergebnisqualität maßgeblich von der Verteilung der Werte abhängt.

Das Theorem ist für einzelne Intervalle zwischen Zweierpotenzen ausgelegt. Mit den zuvor angestellten Überlegungen bezüglich der Gleichverteilung der Werte kann leicht argumentiert werden, dass das Theorem auch für Intervalle von 1 bis zu einer Zweierpotenz Gültigkeit hat. Auf mathematische Weise entsteht dies dadurch, dass die Werte r im Theorem bei zusätzlich kleineren betrachteten Zweierpotenzen ihren Einfluss innerhalb der Grenzwertbetrachtung verlieren und nur der höchste Exponent ausschlaggebend ist.

Aus der Form des Beweises entsteht allerdings die Einschränkung auf Obergrenzen bis hin zu einer Zweierpotenz anstatt beliebiger Werte. Da gegenwärtige Prozessorarchitekturen jedoch im Normalfall genau diesen Bereich darstellen können, ist die Einschränkung für praktische Fälle weniger gravierend.

Mit diesen Überlegungen kann das Theorem eingesetzt werden, um effizienter Summen von Quadraten gegen einen Schwellenwert abzugleichen. Während auch schon bei klassischen Quadraten keine Wurzel gezogen werden muss, sofern auch

der Schwellenwert quadriert wird, ermöglicht eine nachfolgende Division des Schwellenwerts durch zwei den Einsatz von Binary Squares anstatt Quadraten. Der Geschwindigkeitsvorteil wird im Abschnitt 13.5 weiterführend erläutert.

13.5. Repräsentation des Problems

Viele Algorithmen skalieren schlecht mit der Dimensionalität von Vektoren. Um eine euklidische Distanz zu berechnen, müssen alle Differenzen in den einzelnen Dimensionen quadriert und aufsummiert werden. Eine abschließende Wurzelberechnung des Ergebnisses liefert dann die finale Distanz der Punkte zueinander. Wird nun statt herkömmlichen Quadraten mit Binary Squares und der halbierten quadrierten[3] Distanz (Konvergenz gegen Faktor 2) gearbeitet, ergibt sich eine Heuristik, die qualitativ davon abhängt, wie stark die einzelnen Dimensions-Differenzen davon abweichen jeweils nur ein mal im Vektor vorzukommen.

Ein zentraler Vorteil von Binary Squares liegt darin, dass jedes Bit im Binary Square von exakt einem Bit in der Originalzahl abhängt. Dadurch kann die Darstellung von Punkten bzw. gegebenen Vektoren so rekonstruiert werden, dass die einzelnen Bits nach Wertigkeit zusammengefasst werden können. Wird nun für einen gegebenen Vektor die Distanz bestimmt und gegen einen Schwellenwert abgeglichen, so sind die höherwertigen Bits dafür am aussagekräftigsten. Werden diese zuerst betrachtet und liegt damit die Summe bereits über dem Schwellenwert, kann die Betrachtung der restlichen Bits entfallen.

Stark wird das Verfahren beispielsweise im Kontext von SIFT dadurch, dass davon ausgegangen werden kann, dass die allermeisten Punkte bei einer Suche nach einer bestimmten »Point of interest« Umgebung (einem bestimmten Vektor) keine Übereinstimmung aufweisen und die Distanz hoch ist. Dadurch können zu hohe Distanzen schon gleich zu Beginn der Berechnung erkannt werden und die Berechnung weit früher beendet werden.

Anschaulich ist dieses Prinzip in Abbildung 13.2 an einem Beispiel (mit beliebigen Zahlen) dargestellt. Die Werte 15, 0, 39, 19, 44 und 3 sollen in ihrer Summe gegen den Schwellenwert 75 abgeglichen werden. Die direkte Addition der Werte erfordert 4 Additionen (von 5 Zahlen), während die Betrachtung entlang der Bitwertigkeit mit nur einer Addition auskommt.

Der Effekt gestaltet sich um so gravierender, je mehr Dimensionen betrachtet werden, da hierbei »nahe« Punkte nicht deutlich höhere Distanzen aufweisen,

[3] Der Einsatz eines einzelnen Binary Squares für den Schwellenwert wäre nicht zielführend, da der Effekt bezüglich Faktor 2 erst bei größeren Summen auftritt und nicht für einzelne Binary Squares funktioniert.

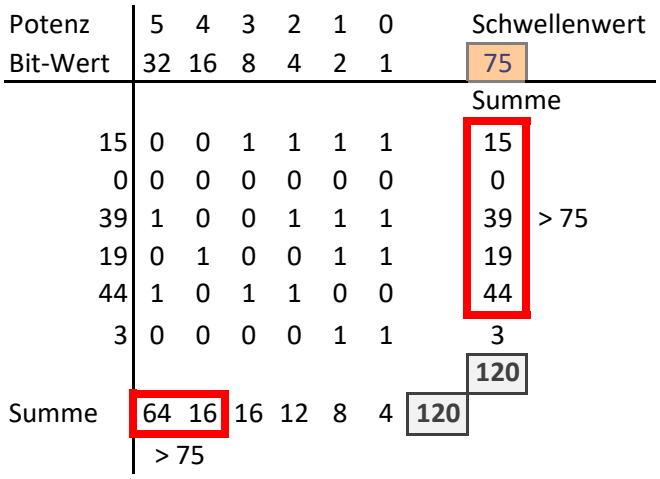

Abbildung 13.2.: Beispiel zur verkürzten Berechnung von Schwellenwerten.

entfernte Punkte jedoch durch die vielen Dimensionen um so größere Distanzen haben.

Aus diesem Grund steigt der erwartete Performancegewinn mit der Zahl der Dimensionen im Vektor. Dieses Verhalten ist begrüßenswert und kontraintuitiv, da eine Vergrößerung der Dimensionszahl meist zu schlechterer Laufzeit führt.

13.6. Aussichten und Limitationen des Ansatzes

Die Vorteile durch Binary Squaring liegen in dem beschleunigten Abgleich von Vektordistanzen zu Schwellenwerten. Beim konkreten Einsatz können Vektoren nach Bitwertigkeit gesichert werden, anstatt als herkömmliche Zahlen. Sobald ein Abgleich stattfindet, kann die Differenzberechnung ebenfalls Bitweise erfolgen, da eine Subtraktion ebenfalls als Addition gewertet werden kann und somit nach dem Kommutativgesetz zunächst mit den höchstwertigen Bit erfolgen kann. Die Summe kann somit nach Bitwertigkeit organisiert konstruiert werden und in jedem Teilschritt bereits gegen den Schwellenwert abgeglichen werden. Mit der Basis der Theorie der Binary Squares kann somit die Multiplikationsoperation gänzlich entfallen. Da pro Bit betrachtet das Binary Square nur von einem Bit der Ausgangszahl abhängt, kann die Berechnung nach der Summierung aller für die Wertigkeit gesetzten Bits einmalig stattfinden. Zusätzlich genügt hierbei eine (schnelle) Shift-Operation.

Die Laufzeitverbesserung setzt also an insgesamt drei Stellen an:

- Vermeidung von Multiplikationsoperationen auf allen Summanden

- Ersetzen von Multiplikation der Summe durch Shifts
- Vorzeitiges Abbrechen des Algorithmus durch Betrachtung der Bits nach Wertigkeit in absteigender Reihenfolge

Wie bei jeder Methode existieren hierbei jedoch auch Nachteile. Zunächst handelt es sich bei der Berechnung auf Binary Square Grundlage um eine Heuristik. Die Qualität der Ergebnisse ist, wie bereits während der Herleitung beschrieben, abhängig von der Verteilung der Werte.

Wird *nicht* der vollständige Weg implementiert (wie zu Beginn dieses Abschnittes erläutert), ergibt sich eine weitere Schwierigkeit: Auf aktuellen Prozessoren existiert keine Unterstützung für Binary Squares, sodass sie manuell berechnet werden müssen und daher Schwierigkeiten in der Konkurrenzfähigkeit gegenüber konventionellen Quadraten haben. In diesem Fall kann Vorberechnung Abhilfe schaffen, sofern die begrenzende Zweierpotenz nicht zu hoch ist. In der vollständigen Variante (welche alle drei genannten Laufzeitverbesserungen aktiv umsetzt) entfällt jedoch die Berechnung von Binary Squares, weswegen das Problem hierbei nicht relevant ist.

Mächtig ist die Methode allerdings auch dadurch, dass lediglich die Multiplikationsoperation ausgewechselt bzw. entfernt wird. Dadurch bleibt sie kompatibel mit sehr vielen anderen Algorithmen, welche ein äquivalentes Anforderungsprofil (Schwellenwertabgleich der Summe vieler Quadrate) aufweisen.

13.7. Zusammenfassung - Subprotokolle

Bei der Abbildung von realweltlichen Prozessen auf Modelle ist ein Übersetzungsprozess unerlässlich. Eine Möglichkeit der Realisierung besteht in der Szenenanalyse, bei der aus bewegten Bildern zusammen mit Wissen Hypothesen abgeleitet werden. Agenten können Subprotokolle nutzen, um Modelle bzw. Interpretationen ihrer Umgebung zu erzeugen. Um ein skalierbares System so performant wie möglich zu halten, ist es unerlässlich Abläufe zu beschleunigen, wobei dabei auch in uneindeutigen und unsicheren Domänen wie der realen Welt Heuristiken zum Einsatz kommen können. Als Novum wurde im Rahmen der Arbeit das Konzept von Binary Squares vorgestellt. Im Falle von Algorithmen, welche große Mengen an Quadraten aufsummieren und gegen einen Schwellenwert abbilden, können auf Basis der Binary Squaring Theorie Performancegewinne realisiert werden. Ein Beispiel ist die Berechnung von euklidischen Distanzen in hochdimensionalen Räumen, wie sie beispielsweise beim Algorithmus SIFT zum Einsatz kommt.

Teil IV.

Prototypen und Evaluation

14. Prototypen

Dieses Kapitel dient zwei Aspekten. Zum einen erfolgt die Dokumentation der erfolgten Implementationen der abstrakten Mushu-Architektur (vgl. Kapitel 5 bis Kapitel 8) und dem zugehörigen erstellen Realisierungskonzept (vgl. Kapitel 9 bis Kapitel 12). Dabei wird insbesondere Bezug genommen auf konkrete Ausgestaltung der Konzepte, eingesetzte Technologien, Programmiersprachen, Frameworks, usw.

Der zweite Aspekt dieses Kapitels adressiert die Dokumentation früherer Prototypen, aus welchen Erkenntnisse gewonnen wurden, die jedoch so nicht fortgesetzt wurden. In diesen Fällen wird zunächst kurz die konzeptuelle Idee erläutert und im Anschluss die Implementation umrissen, jedoch nicht im Detail ausgeführt. Allen Abschnitten dieses Kapitels ist jedoch gemein, dass sie Implementationen beschreiben, welche so erfolgt sind und neben der Arbeit selbst vorliegen. Etwaige Unterstützungen Dritter bei den jeweiligen Implementationen sind im jeweilig zugehörigen Abschnitt angegeben.

Viele, aber nicht alle Prototypen sind mit Unterstützung weiterer Autoren neben dem Autor dieser Arbeit umgesetzt worden. Die jeweiligen Unterstützungen sind bei den entsprechenden Prototypen angegeben; alle weiteren Anteile entsprechen Beiträgen durch den Autor dieser Arbeit.

14.1. Prototypen des Plattformmanagements

Diese Prototypen entstanden während der Entwicklung des Plattformmanagements und dienten sowohl der Überprüfung der Eignung einzelner Systemaspekte als auch dem Ausschluss anderer Ansätze. Zunächst werden zwei Vorläufer des Skalierungssystems vorgestellt und mit Beispielen untermauert. Nach den beiden Beispielen folgt die Implementation der Skalierungskontrolle, wie sie im Realisierungskapitel umrissen wurde: RenewKube.

Das erste Beispiel beschreibt eine Implementation der nebenläufigen Berechnung von Primfaktorzerlegung von natürlichen Zahlen. Darauf folgt im zweiten Beispiel eine Berechnung von Fraktalen durch eine Referenznetzsimulation. Die Beispiele wurden so auch in den Veröffentlichungen des Autors (Moldt, Röwekamp und M. Simon, 2017) sowie (Röwekamp, Moldt und Feldmann, 2018) eingesetzt.

In den Beispielnetzen wird eine Farbcodierung für die einzelnen Bereiche der Netze in ihrem Hintergrund verwendet. Weiße Boxen dienen der Initialisierung und Vorbereitung, während orange Boxen Fachlichkeit umsetzen. Hellblaue Boxen beschreiben Unterabschnitte von Fachlichkeit und magentafarbene Boxen den Abschluss bzw. das Herunterfahren des Systems oder der Komponente. Darüber hinaus verwenden Transitionen die Farbe Weiß für Initialisierungsaufgaben sowie das Entgegennehmen neuer Arbeitspakete und die Farbe Magenta für das Zurückschicken von Ergebnissen.

14.1.1. Virtualisierte Referenznetzsimulation

Zur Illustration des *Einsatzes von virtuellen Maschinen* zum Deployment kann es interessant sein, ein Beispiel für eine konkrete Simulation zu betrachten. Dabei existieren die Komponenten WorkDistributor, WorkLocalParallelizer und WorkerLocal und sind mit konkreten Netzen in den Abbildungen 14.1, 14.2 und 14.3 abgebildet.

Die Netze berechnen eine beispielhafte Arbeitslast, in dem die natürlichen Zahlen von 1 bis 1000 in ihre Primfaktoren zerlegt werden. Der dabei eingesetzte Algorithmus ist naiv und hat nicht den Anspruch eine konkurrenzfähige Alternative für existierende effiziente Algorithmen zur Lösung dieses Problems darzustellen. Vielmehr ist der beabsichtigte Nutzen eine exemplarische Darstellung der Verwendbarkeit des Aufbaus.

Der hier beschriebene Prototyp setzt wie bereits beschrieben als Deploymentform virtuelle Maschinen ein. Das Netz übernimmt hierbei die Rolle vom Management-System in einer sehr rudimentären Form. Dabei war die Generierung von weiteren Berechnungsinstanzen jedoch auf das Innere eines Simulators beschränkt. Es wurde ebenfalls keine Form von Continuous Integration eingesetzt. So wurden physikalische Rechner mit Images der virtuellen Maschinen per Hand bestückt. Als Protokoll kommt an jeder Stelle Java RMI zum Einsatz. Durch die Realisierung als native Referenznetzsimulation war ebenfalls kein Plugin für die Simulationsanbindung notwendig.

In Abbildung 14.1 ist das Netz *WorkDistributor* dargestellt. Seine intendierte Nutzung besteht darin eine einzelne Instanz von ihm auf einer primären[1] GUI Komponente zu erzeugen. Plätze und Transitionen mit weißer Farbe, sowie der Inhalt des Kastens mit der Beschriftung »Initialisierung« dienen dem Aufbau und Start der Simulation sowie der Steuerung und Initialisierung des Distribute Plugins. Über den orangenen Platz können die entfernten Referenzen auf Instanzen des Netzes *WorkLocalParallelizer* erzeugt werden. Die Hauptaufgaben werden durch die

[1] Vgl. den initialen Entwurf der Skalierungskontrolle mit *primärer* Komponente in Abschnitt 10.5.1

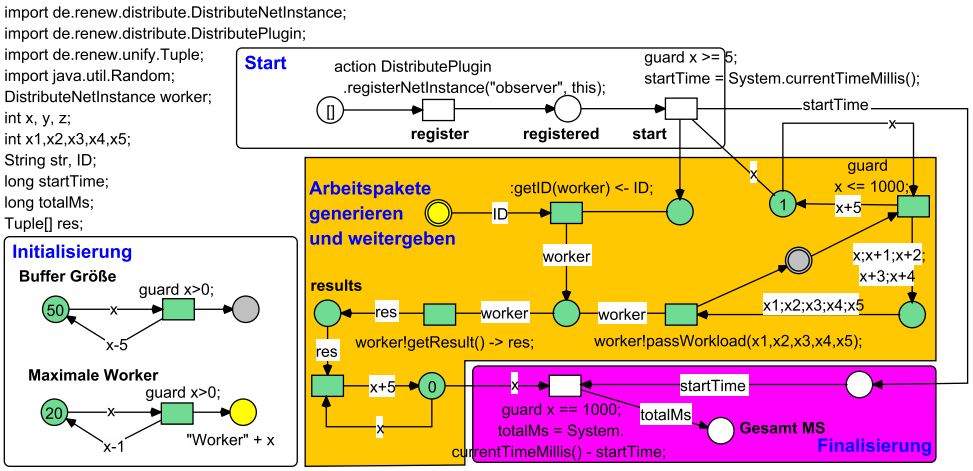

Abbildung 14.1.: WorkDistributor: Sichtbar auf der GUI-Instanz

Transitionen mit der Inschrift `worker!passWorkload` sowie `worker!getResult` ausgeführt. Dabei wird stets ein Tupel von fünf individuellen Arbeitsaufträgen weitergegeben bzw. fünf Ergebnisse vom entfernten Worker entgegengenommen.

Nachdem alle Aufträge berechnet wurden, kann die untere weiße Transitionen schalten und zeichnet die Gesamtdauer der Berechnung auf. Ein Speichern oder Weiterverarbeiten der Ergebnisse kann in der Transition im Nachbereich des mit »results« beschrifteten Platzes hinzugefügt werden.

Anschließend daran findet sich in Abbildung 14.2 eine Darstellung des Netzes *WorkLocalParallelizer*. Der *WorkLocalParallelizer* soll mit jeweils einer einzelnen Netzinstanz pro virtueller Maschine – sprich daher auch pro laufendem Renew Simulator bzw. pro Plattform – in der Simulation existieren. Auch hier dient der Kasten mit der Beschriftung »Initialisierung« der Vorbereitung von Variablen und der Initialisierung des Distribute Plugins.

Der Ablauf der Berechnungen kann darunter von links nach rechts nachvollzogen werden. In dem magentafarbenen Platz ist eine Reihe Referenzen auf *WorkerLocal* Netzinstanzen gesichert. Nach Entgegennahme eines Fünf-Tupels für die weitere Verarbeitung werden die einzelnen Arbeitspakete in einzelne *WorkerLocal* übergeben. Je eine *WorkerLocal*-Instanz kann zu einer Zeit nur ein Arbeitspaket entgegennehmen, während bis zu 15 entpackte Arbeitspakete im Pufferspeicher des *WorkLocalParallelizer* vorgehalten werden können. Hierzu dient der graue Platz als komplementärer Platz.

Sobald eine Berechnung abgeschlossen wurde, wird das errechnete Ergebnis in ein Ergebnisarray übertragen. Ein Zähler, welcher vom Wert 4 dekrementiert wird,

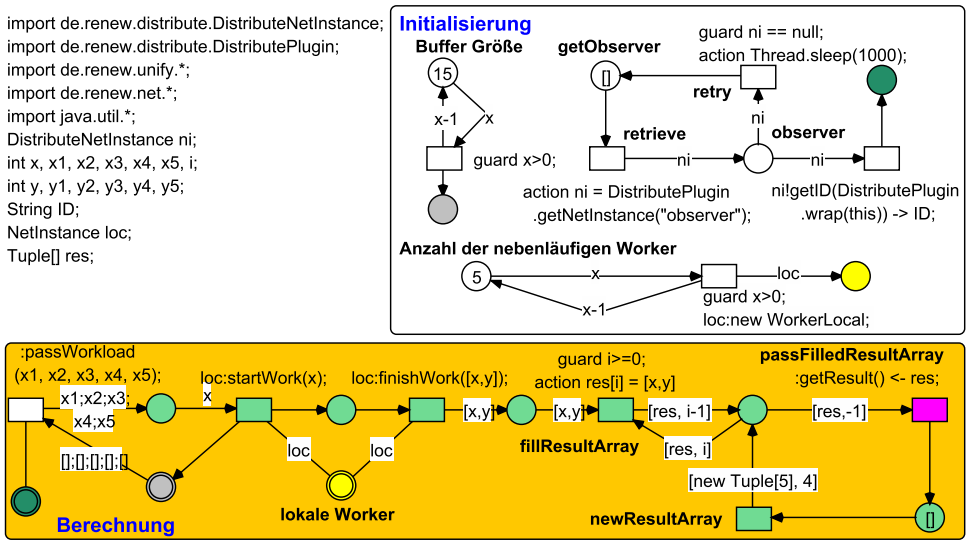

Abbildung 14.2.: WorkLocalParallelizer: Lokales antriggern von Aufgabenbearbeitung

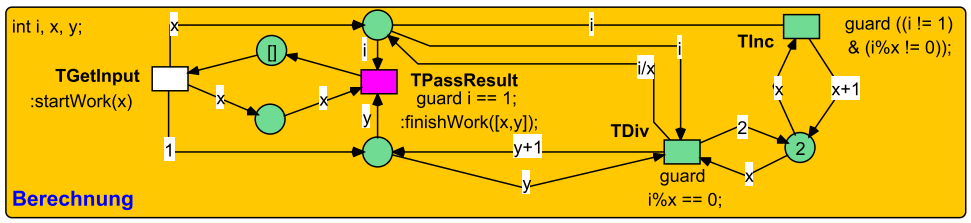

Abbildung 14.3.: WorkerLocal: Arbeitsausführung von Aufträgen

gewährleistet, dass eine Ergebnismenge erst mit genau fünf Einträgen zurück an den *WorkDistributor* übermittelt wird.

Abschließend findet sich in Abbildung 14.3 eine schematische Darstellung des *WorkerLocal* Netzes. Von *WorkerLocal* Netzen werden während der Simulation viele Instanzen erstellt und von *WorkLocalParallelizer* Netzinstanzen referenziert. Im Falle des Beispiels der Berechnung von Primfaktorzerlegung einer natürlichen Zahl stellen die *WorkerLocal* Netze einen einfachen Algorithmus dar. Aufsteigend von der Zahl 2 wird indirekt durch die guard-Inschriften für jede natürliche Zahl geprüft, ob der aktuell betrachtete Wert durch sie ganzzahlig teilbar ist. Die Erhöhung wird durch die Transitionen mit der Beschriftung *TInc* ausgeführt. Falls dies der Fall ist, wird durch die Transition mit der Inschrift *TDiv* die Zahl in die Ergebnismenge übernommen und die zu überprüfende Zahl zurück auf 2

gesetzt. Der aktuell zu betrachtende Wert wird durch die gefundene Zahl geteilt. Sobald der betrachtete Wert die Zahl 1 erreicht, kann davon ausgegangen werden, dass alle Primfaktoren gefunden wurden. Die Auftragsbearbeitung endet mit dem Feuern der Transition *TPassResult*.

Die zentralen Erkenntnisse aus diesem Prototyp umfassen die Machbarkeit einer verteilten Referenznetzsimulation durch die Abstraktion von virtuellen Maschinen. Die Notwendigkeit einer automatisierten Bereitstellung der Startparameter für die einzelnen Instanzen wurde deutlich. Zusätzlich wurde das Konzept des LocalParallelizers erörtert und letztendlich verworfen, da ein einzelner Renew Simulator bereits ausreichend Nebenläufigkeit bereitstellt, um direkt als nebenläufig konzipierte Netze auszuführen. Auch die Aufwände beim Verschieben und Kopieren der virtuellen Maschinen führten dazu, dass daraufhin Container als die nächste betrachtete Deploymentform ausgewählt wurden.

14.1.2. Von Virtualisierung zu Containerisierung

Die Untersuchungen erfolgten mit der Containertechnologie *Docker*. Zum besseren Verständnis werden die Docker-spezifischen Details, welche für den Kontext der Arbeit relevant sind, zunächst eingeführt.

Docker

Docker[2] ist eine Linux[3]-Containertechnologie und wird von Docker Inc. gepflegt. Die Schlüsselkonzepte von Docker sind *Images* und *Container*.

Ein *Image* ist eine Sammlung von Binärdaten und wird normalerweise von anderen Images abgeleitet bzw. erweitert. In einem typischen Anwendungsfall umfasst das Image die minimal erforderliche Laufzeitumgebung, um eine bestimmte Anwendung auszuführen. Images werden in einer *Container Registry* gespeichert und können von einem Docker-Daemon auf einem Zielrechner heruntergeladen werden. Images können lediglich durch ihren Erzeugungsprozess geschrieben werden und weisen ab diesem Zeitpunkt fortan einen Schreibschutz auf.

Der Docker-Daemon kann Images als isolierte Prozesse auf einem System starten. Diesen Prozessen wird dann eine (anfangs leere) schreibbare Ebene hinzugefügt. Image, Prozess und schreibbare Ebene zusammen bilden einen *Container*. Die schreibbare Ebene ist als flüchtig anzusehen und wird gelöscht, sobald der Container gestoppt wird. Für persistente Daten können einzelne Verzeichnisse mit dem Host-System in Form von *Volumes* geteilt werden. Docker Container exis-

[2] https://www.docker.com/ - Zuletzt abgerufen am 15.10.2021
[3] In neueren Implementierungen wird auch Windows unterstützt.

tieren nur für die Zeitspanne bis der zugehörige Hauptprozess terminiert. Nach Termination werden sie automatisch vom Docker-Daemon gestoppt und ggf. entfernt.

Fraktalberechnung

Dieses Beispiel behandelt als zentralen Bestandteil die Berechnung einer Visualisierung der Mandelbrotmenge. Auch hierbei besteht erneut nicht der Anspruch eine in Hinblick auf Performance konkurrenzfähige Alternative zu klassischen Berechnungen der Mandelbrotmenge zu bieten, sondern die Evaluation verschiedener Umsetzungsmöglichkeiten für die Skalierungskontrolle eines Plattformmanagements. Die zentrale Neuerung und Verbesserung gegenüber dem Beispiel zum virtualisierten Deployment liegt in der dynamischen Startbarkeit weiterer Berechnungsinstanzen. Dieser Prototyp setzt als Deploymentform Container mittels Docker ein. Ein Rohentwurf des Management-Systems wurde geschaffen, basierte jedoch noch auf in die Simulation integrierten Referenznetzen. Die Kommunikation erfolgt an allen Stellen des Systems durch Java RMI. Während Berechnungsinstanzen nun dynamisch gestartet werden konnten, musste der entsprechende Systemteil nach wie vor manuell auf den physikalischen Maschinen installiert werden.

Die Mandelbrotmenge erfüllt einen interessanten Satz an Eigenschaften, durch welche sie gut als Grundlage für das Beispiel geeignet ist. Um auf diese Eigenschaften detaillierter eingehen zu können, ist es hilfreich die Mandelbrotmenge zunächst zu definieren.

Sei $z_0 \in \mathbb{C}$ und $z_{i+1} = z_i^2 + z_0$. Die Mandelbrotmenge ist definiert als:

$$M = \{z_0 \in \mathbb{C} | lim_{n \to \infty} z_n < \infty\} \tag{14.1}$$

Sie wurde nach Benoît Mandelbrot benannt und ist erstmalig von Adrien Douady (Douady u. a., 1984) beschrieben worden.

Das Netzsystem, welches im Rahmen dieses Beispiels vorgestellt wird, berechnet eine Visualisierung der Mandelbrotmenge, wie sie an vielen Stellen auch in nicht wissenschaftlichen Beiträgen betrachtet werden kann. Streng genommen zeigt die Visualisierung die inverse Mandelbrotmenge in Farbe, während die Mandelbrotmenge selbst in schwarz dargestellt wird. Intuitiv berechnet sich, ob eine komplexen Zahl z_0 der Mandelbrotmenge angehört dadurch, dass diese Zahl (komplex) quadriert wird und danach die Ursprungszahl z_0 hinzuaddiert wird. Das Ergebnis wird erneut quadriert und es wird wieder die Ursprungszahl z_0 aufaddiert. Wird dieser Vorgang unendlich oft wiederholt und bleibt dabei das Ergebnis endlich, so ist z_0 Teil der Mandelbrotmenge.

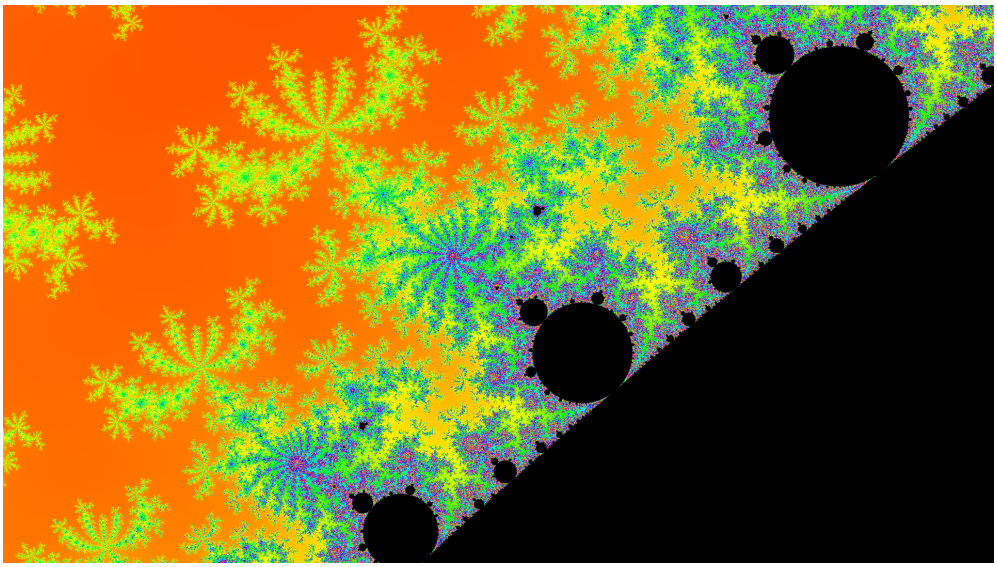

Abbildung 14.4.: Ausschnitt aus der Visualisierung der Mandelbrotmenge. Berechnet durch verteiltes und containerisiertes Renew

Während manche Elemente sehr eindeutig nicht der Mandelbrotmenge angehören - wie beispielsweise die Zahl $5 + 0i$ - und manche Elemente sehr deutlich der Menge angehören - wie beispielsweise die Zahl $0 + 0i$ - ist diese Entscheidung für viele andere Zahlen nicht trivial. Die Visualisierung der Menge entsteht dadurch, dass geprüft wird, ab welcher Anzahl an Iterationen eine Zahlenfolge zu einer Ausgangszahl sicher divergiert. Von einer Divergenz wird ab einer Distanz von 2 vom Ursprung der komplexen Ebene ausgegangen. Die Anzahl der Iterationen, die benötigt werden um die Distanz von 2 zu erreichen, kann so an ein Farbspektrum angelegt werden. In einem digitalen Bild entspricht bei gegebenem Vergrößerungsfaktor und gegebener Position des Bildausschnitts jeder Pixel des Bildes einer komplexen Zahl. Durch Berechnung der Iterationen kann so entweder ein Farbwert ermittelt oder der Pixel schwarz belassen werden, falls nach dem Erreichen einer vorgegebenen maximalen Interaktionsmenge noch immer keine Distanz von 2 vom Ursprung erreicht wurde. Eine entsprechende Visualisierung findet sich in Abbildung 14.4.

Dadurch wird auf anschauliche Weise ersichtlich, dass der Berechnungsaufwand für schwarze Pixel stets am größten ist, da die Maximalzahl an Iterationen durchlaufen worden sein muss. In der Mandelbrotmenge ballen sich für einen gegebenen Ausschnitt häufig Punkte innerhalb der Menge. Daher kann davon ausgegangen werden, dass ein Abschnitt mit einem schwarzen Pixel häufig (aber nicht immer) viele weitere schwarze Pixel in der direkten Umgebung besitzt. Darüber hinaus kann festgehalten werden, dass mit der intuitiven Berechnungsmethode kein Pixel eine (algorithmische) Abhängigkeit zu einem benachbarten Pixel aufweist und

das Beispiel dadurch exzellent parallelisierbar ist. Wird die Berechnung nun so aufgeteilt, dass in jedem Arbeitspaket ein rechteckiger Subausschnitt des Bildes berechnet wird, kann leicht nachvollzogen werden, dass durch Anhäufung von schwarzen Pixeln wesentliche Unterschiede in der Berechnungsintensität der Arbeitspakete entstehen.

Das Beispiel erfüllt damit den Effekt bei Fixierung der Grundparameter eine deterministische, jedoch in sich unterschiedliche Arbeitslast zu produzieren, ohne dabei auf umfangreiche mitgelieferte Daten zurückgreifen zu müssen. Das Ziel bei der Berechnung des Beispiels lag darin eine maximal hohe Gesamtauslastung aller beteiligten Systeme zu erzielen. Dies ist so realisiert, dass die Gesamtaufgabe nicht zerlegt und an die jeweiligen Worker verteilt wird, sondern jeder Worker eigens für die Bearbeitung eines einzelnen Pakets ins Leben gerufen wird. Ein lokaler Teil des Netzsystems, welcher in seinen Aufgaben denen des späteren Management-Systems und Clustermanagers ähnelt, startet dynamisch einen weiteren Teil der Referenznetzsimulation, welcher sich wiederum in die bestehende Simulation einkoppelt. Während die hier gewählte Lösung noch etwas instabil war und ein nicht unerheblicher Teil der technischen Umsetzung nicht transparent für das Modell umgesetzt ist, legt die hier vorgestellte Umsetzung jedoch einen wertvollen Grundstein zur Konstruktion des Management-Systems.

Im Zentrum der Betrachtung steht eine Netzinstanz des Netzes *MandelbrotDistributor*, wie sie in Abbildung 14.5 dargestellt ist. Ohne auf alle Details des Netzes einzugehen, sollen an dieser Stelle nur die Aufgaben der markierten Netzbestandteile erläutert werden. Im Bereich »Initialisierung« können Startparameter wie Bildausschnitt, Bildgröße, Vergrößerungsfaktor, maximale Iterationsanzahlen und Fragmentierungsgrad der Berechnung angepasst werden. Der Bereich »Start« startet die Berechnung und nimmt einen aktuellen Zeitstempel auf. Der orangefarbene Block »Generierung von Arbeitspaketen« erzeugt auf Basis der eingegebenen Startparameter einzelne zu berechnende Bildfragmente und legt diese im Platz »Arbeitspakete« ab. Zusätzlich wird die Anzahl der insgesamt zu erwartenden Ergebnisse berechnet und im Platz »Anzahl noch erwarteter Fragmente« abgelegt. Im Block »Aufträge abschicken« werden Arbeitspakete an Workerinstanzen weitergegeben, während sie darüber im Block »Ergebnisse sammeln« zurückgegeben werden. Links vom Block »Ergebnisse sammeln« steht eine komplementäre Stelle bereit, damit das Distribute Plugin in seinem Protokoll nicht überlastet wird. Im Block »Zusammenfügen« werden die einzelnen zuvor berechneten Arbeitspakete in ein fertiges Bild integriert. Nachdem alle erwarteten Fragmente eingegangen sind, wird hier das Bild auf den Datenträger gesichert und das Netz verklemmt.

Eine weitere wichtige Komponente der Simulation ist in Abbildung 14.6 beschrieben: der *MandelbrotLocalLauncher*. Das Netz ist so intendiert, dass auf jedem physikalischen Knoten je eine Instanz davon existiert. Der vorgelagerte Initialisierungsblock dient der Initialisierung des Distribute Plugins sowie einer Verzö-

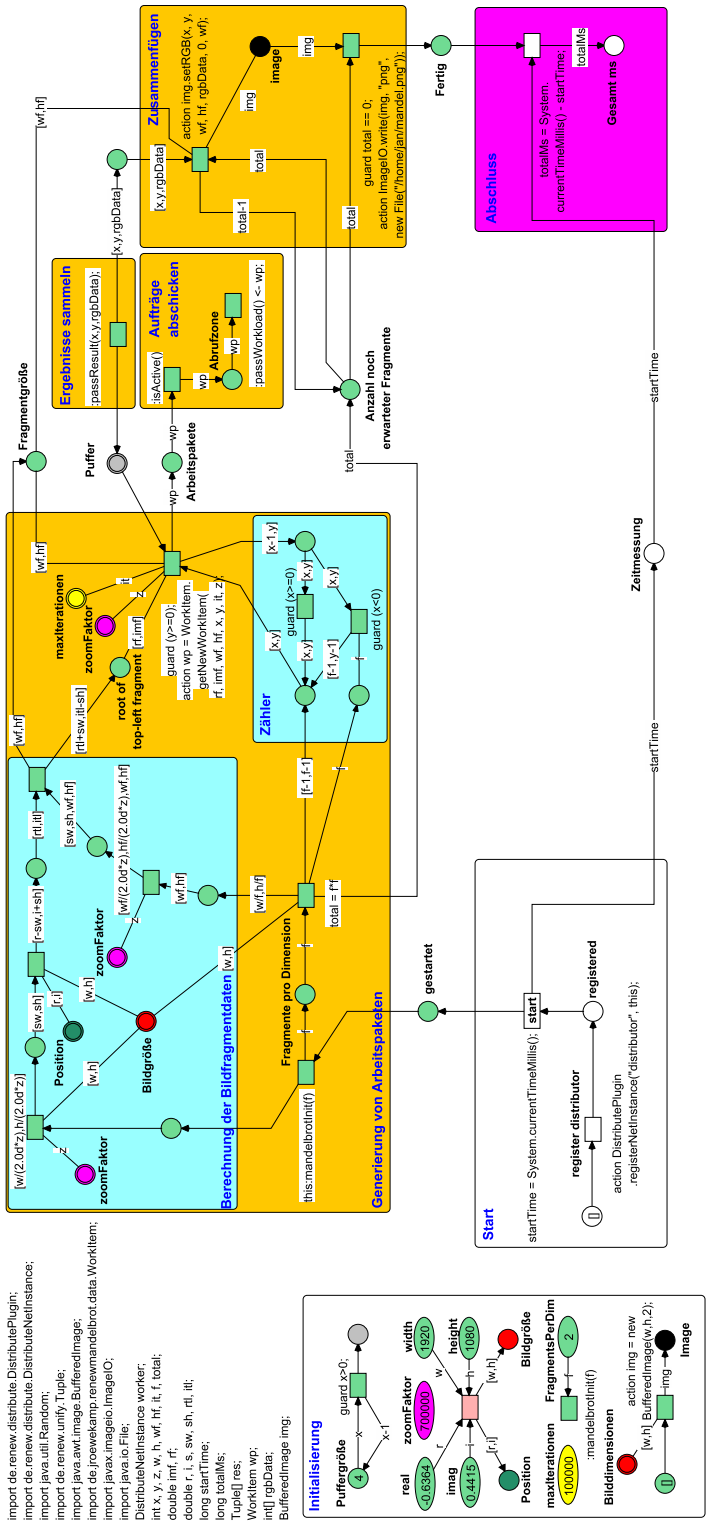

Abbildung 14.5.: MandelbrotDistributor: Generierung von Arbeitspaketen

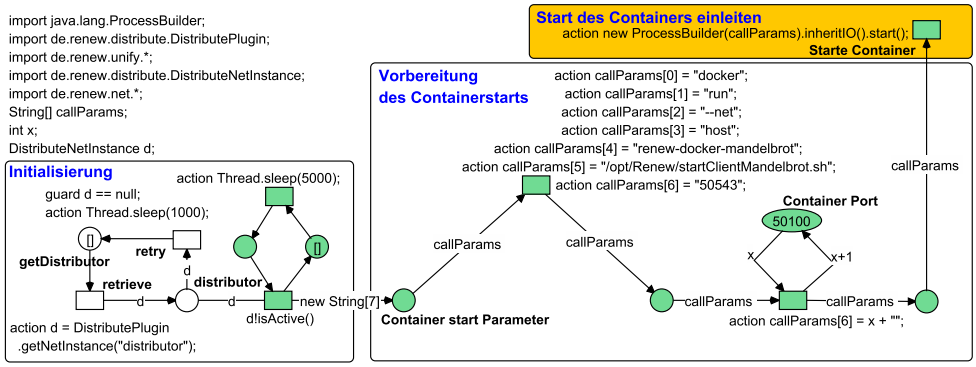

Abbildung 14.6.: MandelbrotLocalLauncher: Konstruieren der Startparameter und starten von Docker Containern

gerung, um das lokale System nicht zu überlasten. Im nachfolgenden Block wird eine Reihe an Kommandozeilenstartparametern generiert, die dazu dienen einen Docker Container mit einer vorbereiteten RENEW Instanz zu starten. Der Start dieses Containers erfolgt im darüberliegenden orangenen Block.

Auf den ersten Blick fällt durch den überwältigenden Teil des Netzes in weißen Boxen auf, dass sehr viele Komponenten des Modells technische Konfigurationen und Vorgänge abbilden. Selbst der einzige orangene Block ist in seiner Ausrichtung auch eher technisch. Diese Bestandteile gaben damals unter anderem den Anstoß, die Steuerung an ein separates Management-System auszulagern. Die entsprechenden Abwägungen wurden in Kapitel 10 festgehalten.

Ferner kann festgehalten werden, dass die zugehörigen Docker-Images auf den physikalischen Maschinen vorliegen müssen, um gestartet werden zu können. Ein Konstrukt, welches die Plattformdefinition, wie sie in MUSHU beschrieben wird, abbildet oder eine Continuous Integration Umgebung bestand zum damaligen Zeitpunkt noch nicht.

Zuletzt findet sich in Abbildung 14.7 eine Abbildung des *MandelbrotWorker* Netzes. Dabei handelt es sich um ein Netz dessen Instanzen im Inneren der durch *MandelbrotLocalLauncher* gestarteten Container existieren. Analog zu den anderen Netzen wird im Block »Initialisierung« das Distribute Plugin initialisiert. Im darauf folgenden Block »Berechnung« wird für ein gegebenes Arbeitspaket das entsprechende Ergebnis berechnet und sodann an den *MandelbrotDistributor* zurückgegeben. Nachdem dieser Vorgang erfolgreich abgeschlossen wurde, wird das System durch ein Feuern der Transition innerhalb des »Herunterfahren« Blocks beendet. Der Aufruf dieses Befehls hat die Beendigung der Java JVM zur Folge, wodurch wiederum eine Termination des gesamten Containers angestoßen wird.

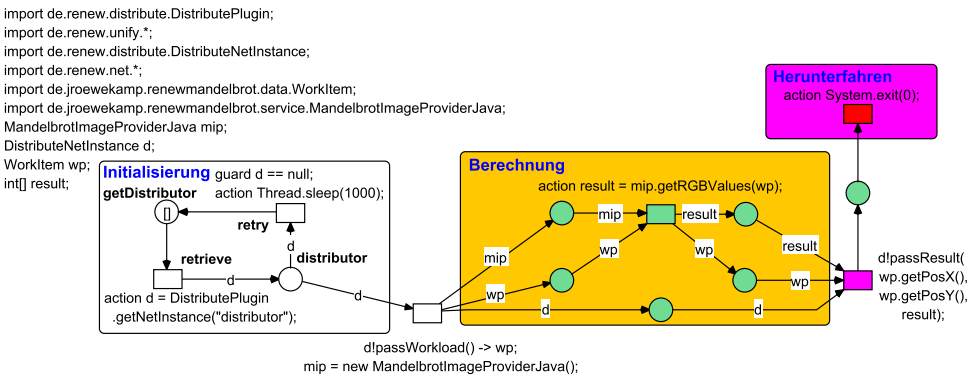

Abbildung 14.7.: MandelbrotWorker: Bearbeiten des Arbeitspakets und anschließendes Herunterfahren

Die wesentlichen Beiträge durch das Beispiel waren eine Proof-of-Concept Implementation der dynamischen Skalierung mittels Referenznetzsystem und Containertechnologie. Darüber hinaus bietet es exzellente Evaluationsmöglichkeiten in Bezug auf die Notwendigkeit der verschiedenen Systemkomponenten des Mushu-Plattformmanagements. Ein Problem entstand durch die einseitige Kommunikation mit der Containertechnologie in Bezug auf den Startvorgang. Nicht immer konnten die Container zuverlässig gestartet werden, sodass teilweise Probleme mit erwarteten Berechnungen entstanden. Darüber hinaus war es mühselig einen speziellen Teil der konkreten Referenznetzsimulation nach wie vor manuell auf einem physikalischen System vorhalten zu müssen. Da bereits Containertechnologie eingesetzt wurde, wäre es wünschenswert, wenn das Vorhandensein eines Containerhosts auf physikalischen Knoten ausreichend wäre. Dieser Umstand gepaart mit der Unzuverlässigkeit der Containerstarts machte den Einsatz eines Clustermanagers wünschenswert.

14.1.3. RenewKube

RenewKube ist die umfangreichste und weitreichendste Implementation aller in Kapitel 10 vorgestellten Aspekte des Realisierungskonzepts des Mushu-Plattformmanagements. Der Name ergibt sich aus der Kombination des Simulators Renew sowie des Clustermanagers Kubernetes. RenewKube implementiert damit alle vorgesehenen Komponenten des Mushu-Plattformmanagements mit Ausnahme der plattformspezifischen und der kommunikationsspezifischen Komponenten.

Neben einer kurzen Einführung in Kubernetes und das Spring Framework, welche beide im Kontext von RenewKube eingesetzt werden, beschreibt dieser Ab-

schnitt wie die einzelnen Komponenten der RenewKube Implementation das Realisierungskonzept mit konkreten Technologien umsetzen.

Kubernetes

Kubernetes ist ein Clustermanagement- und Containerorchestrierungswerkzeug, welches ursprünglich von Google entwickelt wurde und später der Cloud Native Computing Foundation zur Verfügung gestellt wurde. Die Hauptfunktionalität, welche von RenewKube eingesetzt wird, ist die Fähigkeit, Container aus der Ferne zu starten und herunterzufahren. Dabei bietet Kubernetes einfache Möglichkeiten die Anzahl der Replikationen von Containern gezielt zu steuern. Es bietet eine Vielzahl von Komponenten und Konzepten, welche zu vielfältig sind, um sie hier in ihrer Gesamtheit zu behandeln. Kubernetes fügt oberhalb der Containerisierung diverse weitere Abstraktionsebenen ein. Eine gute generelle Einführung findet sich auf der Website des Projekts[4]. Im Folgenden werden nun die für diese Arbeit relevanten Konzepte vorgestellt:

- Ein *Pod* kann einen oder mehrere Container beherbergen und ist die atomare Einheit von Kubernetes. Pods sind eine flüchtige Struktur und können jederzeit heruntergefahren oder gestartet werden.
- Ein *ReplicaSet* definiert eine Gesamtzahl von Pods, welche der gesamte Cluster ausführen soll. Wenn die tatsächliche Anzahl der Pods von der gewünschten Anzahl abweicht, startet oder stoppt das ReplicaSet die Pods, um diese Anzahl anzupassen. Die Implementation eines ReplicaSets entspricht daher im Wesentlichen dem Reconciler Pattern. ReplicaSets sind für den Einsatz mit zustandslosen Komponenten konzipiert, während die analogen *StatefulSets* für zustandsbehaftete Komponenten konzipiert sind. ReplicaSets sind jedoch an vielen Stellen besser integriert, weswegen ein Einsatz für zustandsbehaftete Komponenten in Ausnahmefällen sinnvoll sein kann.
- Ein *Deployment* enthält in der Regel ein zugehöriges ReplicaSet und bietet fortgeschrittenere und umfangreiche Funktionen wie rolling Updates und dergleichen.
- Ein *Service* bietet eine Art Zugriffsschicht, indem bestimmte Teile des Clusters entweder anderen Teilen de Clusters oder aber der Außenwelt zugänglich gemacht werden. Sie können auf einen oder mehrere Pods oder Deployments gerichtet sein. Da Pods flüchtig sind, erfordert die Zuordnung von Services zu Pods besondere Aufmerksamkeit und ist mit den integrierten Kubernetes-Funktionen nicht ohne weiteres möglich.

[4] https://kubernetes.io/ - Zuletzt abgerufen am 15.10.2021

Spring Framework

Das Spring Framework bietet eine Reihe an Hilfsmitteln für die schnelle Entwicklung webbasierter Anwendungen mit Sprachen der Java Familie wie Java, Groovy oder Kotlin. Der Fokus dabei liegt auf Dependency Injection[5], Entkopplung der einzelnen Komponenten sowie auf aspektorientierter Programmierung. Spring selbst bietet viele Unterprojekte für spezifische Anwendungsbereiche von denen insbesondere Spring Boot für die schnelle Konfiguration einer Spring-Anwendung heraussticht. Spring ist verfügbar unter der Apache-Lizenz und wird von Pivotal Software (welches seit Ende 2019 ein Teil von VMware ist) gewartet.

Im Kontext des Cloud-Native RENEW-Plugins kann Spring eingesetzt werden, um viele der notwendigen Handwerksarbeiten effizient umzusetzen. Dabei handelt es sich im Wesentlichen um gelöste Probleme, deren Implementation zwar zeitaufwendig, aber wenig erkenntnisbringend wäre. Spring bietet eine einfache Integration mit Webservern, um Webservices zu realisieren, aber auch mit zahlreichen anderen verbreiteten Technologien. Es bietet daher eine solide Ausgangsbasis sowohl für die beabsichtigte Implementation der Plattformkomponenten als auch für nachfolgende Untersuchungen unter der Einbeziehung gänzlich anderer Technologien. Für diese besteht eine hohe Chance, dass bereits eine Integration mit Spring existiert, sodass auch dort Handwerksarbeit entfallen können wird. Eine konkrete Technologie ist dabei Apache Kafka, wie noch im Prototyp zu Resilient Distribute in Abschnitt 14.3.2 Erwähnung finden wird.

Eine zentrale Entität im Spring-Ökosystem ist der sog. *Applikationskontext*, welcher über eine Vielzahl von Eigenschaften und Funktionalitäten verfügt. Die in diesem Zusammenhang wichtigste Funktionalität ist hierbei ein zentrales Register aller von Spring erzeugten Objekte. In der Spring-Terminologie werden diese dem Kontext bekannten Objekte als *Beans*[6] bezeichnet.

RenewKube Manager

Beim RENEWKUBE Manager handelt es sich um eine alleinstehende Spring-Anwendung. Sie ist ohne weitere Abhängigkeiten in Java implementiert und kann daher hervorragend in einem eigenen Container bereitgestellt werden. Der RENEWKUBE Manager setzt alle in Abschnitt 10.2 formulierten Konzepte um und

[5] Bei der Dependency Injection erzeugen Algorithmen selbst keine Instanzen von Klassen mehr. Diese werden viel mehr von der Umgebung bereitgestellt. Der Begriff beschreibt somit die Praxis externe Abhängigkeiten in Objekte zu injizieren. Dies erlaubt es Algorithmen nur mit Spezifikationen zu arbeiten ohne das Wissen über die konkrete Implementation und dient somit der Entkoppelung. Für weitere Informationen siehe beispielsweise (FOWLER, 2004).

[6] Die genaue Definition einer *Bean* ist etwas spezifischer, für den Kontext der Arbeit genügt jedoch die aufgezeigte Analogie.

bildet damit ein vollwertiges Management-System im Sinne des Realisierungskonzepts des Mushu-Plattformmanagements ab. Der RenewKube Manager ist direkt an einen Kubernetes Cluster angebunden. Beim Start muss ihm daher eine entsprechende Legitimation, um auf den Cluster zugreifen zu können, mitgegeben werden.

RenewKube Plugin

Beim RenewKube Plugin handelt es sich entsprechend der Ausführung um die Anbindung an die Referenznetzsimulation. Es ist als Renew-Plugin implementiert und verwendet als Grundlage das Distribute Plugin, benötigt sonst aber keine weiteren Abhängigkeiten. Das RenewKube Plugin stellt als Funktionen im Wesentlichen drei Funktionen bereit:

- Agenten der Simulation können eine neue gewünschte Anzahl an Plattformen an das Plugin senden.
- Die aktuell verfügbare Anzahl an Plattformen kann abgerufen werden.
- Die aktuell gewünschte Anzahl an Plattformen kann abgerufen werden.

Durch den Einsatz des Reconciler-Patterns können die Zahlen der letzten beiden Punkte temporär abweichen.

CI/CD mit GitLab

GitLab ist eine integrierte Lösung für Code Repositories, Projektverwaltung, Continuous Integration Pipelines, Docker Container Registries, und vieles Weiteres. Es deckt den gesamten technischen Anteil an Agilität und auf einer konzeptuellen Ebene das Update der Plattformdefinition ab. Bei RenewKube wird es für die automatische Kompilation der einzelnen Komponenten eingesetzt, welche in diesem Abschnitt beschrieben werden, sowie für die allgemeine Bereitstellung von Docker Images für besagte Komponenten.

Frühere Versionen von RenewKube setzten eine generische (private) Docker Registry ein. Diese erfordert bei jedem Aufsetzen des RenewKube Deployments einen erneuten manuellen Aufwand und bietet weit weniger Funktionen, als die integrierte Lösung mit GitLab. In Umgebungen, in denen GitLab jedoch nicht verfügbar ist, muss im Zweifel dennoch darauf zurückgegriffen werden.

Kubernetes und Java RMI

Eine weitere Problematik besteht in der Integration von Kubernetes und Java RMI. Und die vollumfängliche Implementation komplexer Kommunikation auf der Ebene der Agenten bleibt nur der Einsatz der klassischen Implementation des Distribute Plugins. Dieses setzt bekanntermaßen auf Java RMI. Java RMI ist als zustandsbehaftetes Protokoll implementiert, sodass eine gezielte Instanz eines laufenden Simulationsprozesses angesprochen werden muss, um sinnvolle Ergebnisse erzielen zu können.

In der normalen Bereitstellung durch Kubernetes können Services eingesetzt werden, um Bereichen außerhalb des Clusters Zugriff auf die Elemente innerhalb des Clusters gewähren zu können. In diesem Fall wäre dies die primäre Plattform der Simulation, an der auch eine GUI Komponente angehängt ist. Diese Services werden im Normalfall an Deployments oder ReplicaSets angehängt und die Auswahl des konkreten Pods erfolgt durch überdies auf automatisierte Art und Weise. Leider ist diese Praxis nicht mit den Anforderungen von Java RMI vereinbar.

Aus diesem Grund erstellt der RenewKube Manager für jede neu gestartete Plattform eine entsprechende Service-Exposition, welche auf den zugehörigen Pod verweist. Wird eine entsprechende Plattform wieder entfernt, so wird der zugehörige Service ebenfalls entfernt. Jedem dieser Services wird ein spezifischer Port zugewiesen, auf dem der Service erreichbar ist. Dieser Port kann an jedem beliebigen physikalischen Rechner des Clusters angesprochen werden. Das interne Routing zum korrekten Container übernimmt Kubernetes.

Ablauf einer Simulationsinitialisierung in RenewKube

Zur Illustration des Ablaufes stellt Abbildung 14.8 den Start einer Simulation in der RenewKube Umgebung dar. Hierbei wird der Einsatz des klassischen Distribute Plugins auf Java RMI Basis angenommen. Dabei existieren die Rollen der (primären) Plattform, der Distribute (RMI) Registry, dem RenewKube Manager, dem Clustermanager Kubernetes und einem entfernten physikalischen Knoten, welcher eine weitere Plattform startet.

Nach der Initialisierung der Simulation wird zunächst ein Archiv mit dem Schattennetzsystem aller im Simulator geöffneten Netze erzeugt und an den RenewKube Manager versandt. Dieses Archiv dient der initialen Versorgung von Netztemplates, sodass der entfernte Simulator die Simulation starten kann. Der RenewKube Manager verbucht auch lokal die Simulation als gestartet und informiert die primäre Plattform über diesen Zustand. Gleichzeitig erzeugt er in Kubernetes ein entsprechend Deployment für die Simulation, bei der in diesem Beispiel direkt eine weitere entfernte Plattform in die Simulation eingegliedert werden soll.

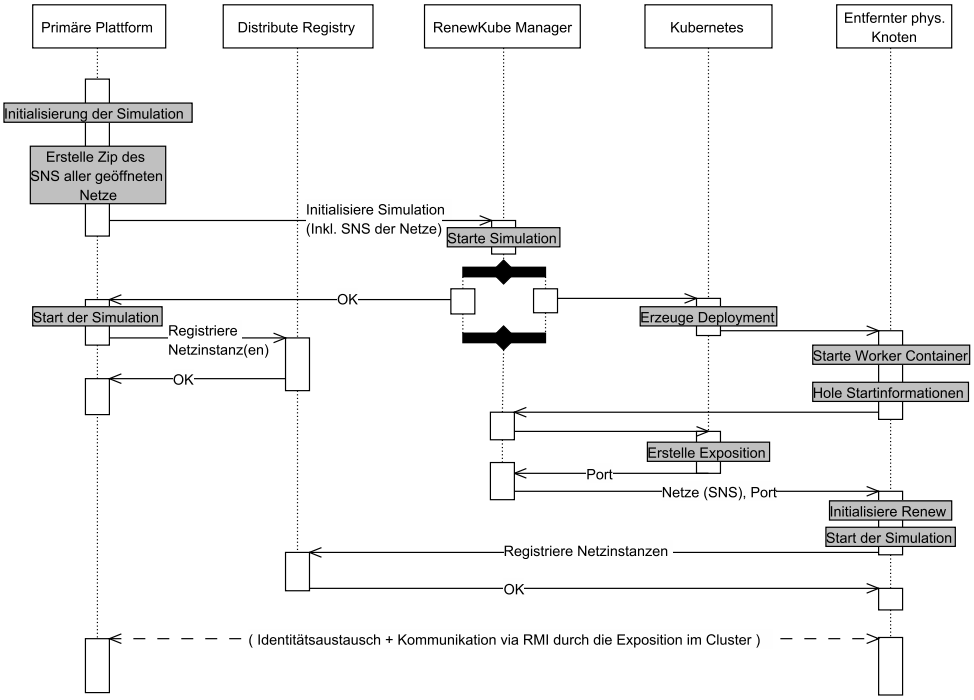

Abbildung 14.8.: Schematische Darstellung vom Ablauf eines Simulationsstarts

Auf dem entfernten physikalischen Knoten wird der entsprechende Container gestartet und in Vorbereitung auf den Start des Simulators die entsprechende Startinformation vom RenewKube Manager abgerufen. Dabei liefert der entfernte Knoten seine eigenen lokalen Netzwerkkoordinaten an den RenewKube Manager, welcher dann wiederum mit den Koordinaten eine entsprechende Service-Exposition beim Clustermanager Kubernetes beantragen kann. Nach erfolgter Exposition ist der externe Port, auf welchem der neue Simulator erreichbar ist, bekannt und der RenewKube Manager kann diese Information zusammen mit dem Schattennetzsystem an die neu zu starten Plattform übergeben. Diese kann das Wissen über den externen Port nutzen, um potenziellen Gesprächspartnern ihre Adresse mitteilen zu können. Die Notwendigkeit hierzu entsteht daraus, dass die initiale Kommunikation über Java RMI über andere Wege als fortwährende Kommunikation abläuft. Für Letztere muss jede Anwendung ihre eigene Adresse aktiv zur Verfügung stellen.

Nach dem Start registrieren beide Plattformen Netzinstanz-Informationen in der Distribute Registry. Ab diesem Zeitpunkt kann über die Protokolle des Distribute Plugins eine gemeinsame, verteilte Simulation stattfinden. Da die genauen Abläufe abhängig vom simulierten Inhalt sind, ist die Kommunikation in der Abbildung mit einem gestrichelten Doppelpfeil abgebildet.

14.1.4. Beispiel Erreichbarkeitsgraph

Ein letzter Prototyp entstand im Rahmen der Abschlussarbeit (Engelhardt, 2020) durch Henri Engelhardt mit dem Namen Distributed Analysis, welche auf dem Plugin MoMoC (Willrodt, 2019; Willrodt, Moldt und M. Simon, 2020) aufsetzt und die dortigen Ergebnisse mit RenewKube integriert. Ziel dabei ist die verteilte Berechnung von Erreichbarkeitsgraphen von gegebenen Netzen. Dabei kann die Menge an für die Berechnung zur Verfügung stehenden Renew Simulatoren dynamisch durch die Simulation angepasst werden.

14.2. Prototypen der Plattform

Neben den Implementationen der Prototypen des Plattformmanagements stellt die Umsetzung der Plattform einen weiteren wichtigen Stützpfeiler der Mushu-Architektur dar. Zunächst wird der Ansatz der *replizierten Netzdatenbank* vorgestellt, welcher letztendlich nicht weiter verfolgt wurde. Anschließend wird die erfolgte Umsetzung der Aspekte der Mushu-Plattform in der Form des *Cloud-Native* Renew *Plugin* beschrieben.

14.2.1. Replizierte Netzdatenbank

Die konzeptuellen Grundlagen für diesen Prototypen wurden innerhalb der vergangenen Kapitel nicht beschrieben. Dies liegt primär darin begründet, dass die Idee aufgrund von absehbarer Performanceproblematik nicht mehr weiterverfolgt wurde. Da sie allerdings interessante Aspekte beinhaltet, die für weitere Untersuchungen jenseits der Arbeit von Interesse sein können, soll der zugehörige Prototyp an dieser Stelle dennoch umrissen werden. Die Konzeptualisierung und einzelne, vorbereitende Implementationen erfolgten in Kooperation mit Matthias *Feldmann*.

Die Ideen zur Realisierung mittels replizierter Netzdatenbank wurden ebenfalls auf dem PNSE Workshop im Rahmen der Veröffentlichung (Röwekamp, Feldmann u. a., 2019) vorgestellt. Damals bezogen sich die Überlegungen allerdings noch auf die Simulation von P/T-Netzen. Die Kernidee bestand darin, eine zentrale, mit Replikation und Sharding versehene Datenbank für alle Operation der Simulation einzusetzen. Auf diese Weise könnte die Skalierbarkeit auf Ergebnisse der Datenbanktheorie übertragen werden. Streng genommen handelt es sich dadurch nicht ausschließlich um eine Umsetzung der Plattform, sondern übernimmt einige andere Teile aus dem Plattformmanagement. Die zentralen Aspekte sind jedoch in der Domäne der Plattform verortet.

Ein weiterer Kernaspekt bestand darin, die Sichtweise von Marken als aktive Elemente im Gegensatz zu Transitionen im Netz einzunehmen. Marken würden dabei eine spezifische Heimatplattform besitzen, durch welche sie als einziges konsumiert werden können. Marken können jedoch auf beliebig vielen Gastplattformen zugegen sein, dort können allerdings lediglich lesende Operationen auf ihnen ausgeführt werden. Marken haben stets die Bestrebung, sich selbst durch ein Transitionsfeuern zu konsumieren. Sie suchen daher auf ihrer Heimatplattform nach Transitionsinstanzen im Nachbereich der Platzinstanz, in der sie liegen.

Werden für den Feuervorgang weitere Marken benötigt, so werden diese als Gastinstanzen angefragt, sofern sie nicht dieselbe Heimatplattform aufweisen. Jeder geplante Feuervorgang bekommt eine zufällige Priorität zugewiesen, und sofern mit Marken von Gastinstanzen eine Transition aktiviert ist, versucht die initiierende Marke eine Migration zur lokalen Plattform als neue Heimatplattform aller beteiligten Marken zu bewirken. Liegen alle Marken lokal vor, kann der Feuervorgang abgeschlossen werden. Bei Konflikten, wenn beispielsweise zwei Marken gegenseitig die jeweilige Migration auf die für sie lokale Plattform fordern, entscheidet die höhere der zufällig gewählten Prioritäten.

Der Ansatz hat den Vorteil, dass die Verteilung von Daten aus Applikationssicht zentralisiert erscheint und damit die Notwendigkeit für eine applikationsseitige Verteilung entfällt. Auch Datenbankmodelle wie bei beispielsweise Graph-

datenbanken könnten hohe Synergien mit dem gespeicherten Modell aufweisen, da Petrinetze Graphen sind. Auch die hohe Komplexität, Checkpoints in True-Concurrency Simulationen zu implementieren, könnte vollständig auf bereits bestehende Lösungen zu Checkpoints aus der Datenbanktheorie reduziert werden. Damit einhergehend würde eine Form von Simulationsfeed nicht mehr schwer zu implementieren sein.

Durch den ständigen Durchgriff auf eine potenziell entfernt liegende Datenbank würde jede einzelne Berechnung an jeder Stelle der Simulation mit einem verteilten Aufruf einhergehen. Allein dies stellt einen gewaltigen Nachrichtenoverhead dar, welcher in keiner Situation zu einer zum großen Teil lokalen Berechnung konkurrenzfähig wäre. Auch die Berechnung von Plattform-lokalen Operationen im Sinne der Mushu-Architektur wäre nicht ohne Weiteres möglich.

Darüber hinaus besteht keine direkte Kontrolle über die Verteilung der Daten aus der Simulation heraus, ohne sich dabei tief in die Interaktion mit der jeweiligen Datenbank zu begeben. Dies führt zu der Gefahr einer technologieabhängigen Lösung, welche im Allgemeinen für wissenschaftliche Arbeiten nicht angestrebt werden sollte. Darüber hinaus kann es Schwierigkeiten geben, alle geführten Beweise im Kontext der Referenz- und Petrinetze auf ein Simulationsmodell mit aktiven Marken zu übertragen.

Auch detaillierte Fragen zum Simulationsalgorithmus, beispielsweise wie und wann erneut die Aktiviertheit einer Transition geprüft werden könnte, ohne dabei erneut Transitionen oder Plätze als aktive Elemente (z.B. durch Trigger auf Transitionen) einzuführen, bleiben im Modell der aktiven Marken fraglich. Aus diesen Gründen wurde von der umfänglichen Umsetzung dieses Ansatzes abgesehen.

14.2.2. Cloud-Native Renew-Plugin

In der Form des Cloud-Native Renew-Plugins erfolgte die Implementation (eines Großteils) der im Realisierungskonzept der Plattform beschriebenen Umsetzungen. Dabei handelt es sich im Wesentlichen um die Komponenten, welche aus den Cloud-Native-Hintergründen der Plattformarchitektur (vgl. Abschnitt 6.1) entstehen. Dieser Zusammenhang führt auch zu dem Namen des Plugins für den Simulator Renew.

Wesentliche Teile des Plugins bauen auf dem Spring Framework auf, welches bereits vorgestellt wurde. Die Integration mit Renew (4.0) war einigen Schwierigkeiten unterworfen und wird daher auch noch einmal gesondert adressiert. Abschließend werden die Abbildungen der einzelnen Anteile des Realisierungskonzepts auf konkrete technische Lösungen vorgestellt.

Die technische Implementation des Cloud-Native Renew-Plugins erfolgte mit Unterstützung von Marvin Taube, Alexander Senger, Patrick Mohr und Laszlo Korte. Die Implementation wurde ebenfalls als ein Anteil in der Publikation (Röwekamp, Taube u. a., 2021) veröffentlicht.

Einbindung von Spring in das bestehende Renew Ökosystem

Sowohl Spring als auch Renew bieten sehr unterschiedliche eigene komplexe Ökosysteme. Die Integration beider Ansätze gestaltet sich somit nicht trivial und bedarf einiger Vorüberlegungen. Zunächst sollte geklärt werden, ob die Komponenten gleichberechtigt nebeneinander ausgeführt werden sollen oder eine in die andere Komponente integriert werden kann.

Parallelbetrieb Der parallele Betrieb beider Komponenten entkoppelt den Kompilierungsvorgang beider Systeme vollständig, da sie direkt nicht mehr voneinander abhängen. Diesem Vorteil steht jedoch der Nachteil gegenüber, dass beide Softwarekomponenten nun mittels Interprozesskommunikation kommunizieren müssen. Dieser Aufwand steht in keinem Vergleich zum Overhead des Nachrichtenversands über ein externes Netzwerk, birgt jedoch einen außerordentlich großen Overhead an Berechnungs- und Threadsynchronisationsaufwand für überschaubare Vorteile. Zusätzlich entfällt die direkte Integration auf Java-Ebene der Komponenten, sodass Kompatibilität schwieriger herzustellen ist. Darüber hinaus wird das Testen erschwert, da bereits eine einfache Komponente des Systems aus einer Integration zweier Bestandteile besteht.

Integration von Renew in Spring Wie eingehend erläutert umfasst das Renew Ökosystem zahlreiche Bestandteile, Abläufe und ist durch viele und umfangreiche Architekturentscheidung konstruiert worden. Diese entsprechen nicht unmittelbar der Architektur von Spring, sodass bei dieser Variante die Struktur von Renew aufgebrochen werden müsste. Weder von der Sinnhaftigkeit noch vom notwendigen Aufwand kann dies empfohlen werden. Eine andere Variante wäre es Renew als isolierte Einheit innerhalb der Springanwendung zu betrachten. Bei der zweiten Variante müsste das gesamte Renew-System inklusive Plugin-Loader und Plugins in eine Bibliothek verpackt werden. Um dieses Unterfangen qualitativ hochwertig zu gestalten, wäre die Entwicklung einer entsprechenden Schnittstelle der Bibliothek notwendig. Unabhängig von den technischen Hürden, würde jedoch auch der zentrale Gedanke der Autonomie und Optionalität von Renew durch eine derartige Lösung untergraben werden, welche fortan stets das Spring Framework voraussetzen würde. Ein weiteres Problem ergibt sich durch das JPMS (Java Plattform Module System), welches von Spring so nicht unterstützt wird, jedoch in Renew ein integraler Bestandteil ist. Die

allgegenwärtige Reflexion in verschiedenen Implementationen innerhalb des Spring Frameworks macht einen Einsatz schwierig.

Integration der Spring-Anwendung in Renew Die Integration der Spring-Anwendung in Renew liefert die Vorteile, dass das bestehende Renew Plugin-System genutzt werden kann, um die Integration einfach zu gestalten. Zusätzlich stehen effiziente Wege vonseiten der Spring-Applikation zur Verfügung, um diese selbst als Bibliothek zu konfigurieren. Auf diese Weise sind die Objekte des Renew Simulators jedoch nicht als Beans im Spring-Applikationskontext verfügbar. Dies erfordert somit den Einsatz erweiterter Softwareentwicklungs-Entwurfsmuster, wie dem Beobachter-Muster, da sonst aus Spring keine direkten Aufrufe in den Simulator erfolgen könnten. Eine weitere Schwierigkeit besteht im Kompilationsprozess, da Renew bis Version 2.6 auf dem Buildtool *Ant* basiert, welches für Spring nicht verfügbar ist. Mit Version 4.0 (Daschkewitsch, 2019; Janneck, 2021) setzt Renew jedoch auf *Gradle*, welches ebenfalls für Spring verfügbar ist.

Abschließend lässt sich festhalten, dass die einzig mögliche Lösung, welche die grundlegenden Strukturen der Mushu-Architektur erhält, aus der Integration von Spring in Renew besteht. Dies erfordert wie bereits beschrieben, den Einsatz von Renew 4.0, um einen umfassend integrierten Kompilationsprozess zu gewährleisten. Es wären zwar Kompatibilitätkonstruktionen für den Einsatz von Renew 2.6 denkbar, jedoch ist die Variante direkt die neue Version zu unterstützen zukunftsorientierter. Die wesentliche Neuerung von Renew 4.0 besteht jedoch in der Einführung von Modulen und Modullayern in Renew (Janneck, 2021). Spring integriert das Modulkonzept nicht, sodass die gesamte Springrealisierung innerhalb eines einzigen Moduls erfolgen muss.

Umsetzung von Plattformkomponenten

Die konkrete Realisierung der einzelnen Komponenten der Plattform orientiert sich sehr nah an den Ausführungen von Kapitel 11. Für alle Komponenten wurde eine Webschnittstelle in Form eines REST-Controllers erzeugt und ein dahinter gelagerter Interoperationsservice, welcher die entsprechenden Events in den Simulationskern bzw. in die -umgebung von Renew überträgt.

Auf technischer Ebene können somit (Schatten-)Netzdefinitionen und Plugins zur Laufzeit in die Springanwendung geladen werden und mit einem anderen Endpunkt diese hochgeladenen Daten entweder instanziiert oder aber geladen werden. Zusätzlich können verschiedene Systemdaten und Logs abgerufen werden. Ferner kann die Simulation mit entsprechenden Endpunkten gesteuert werden und es kann ebenfalls eine Übersicht über alle möglichen Operation ausgegeben werden.

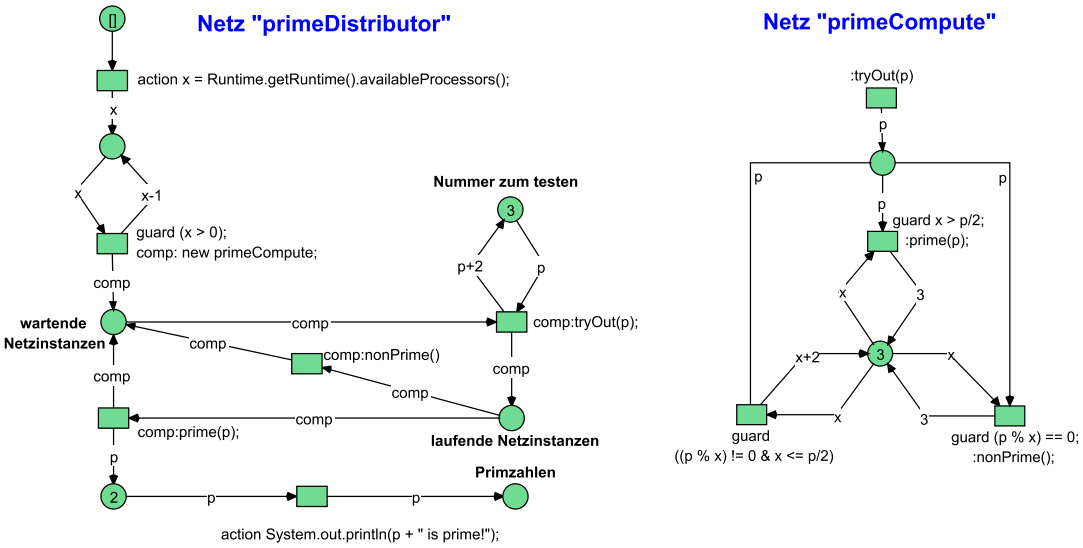

Abbildung 14.9.: Primzahlberechnung ohne Plattformfunktionalität

Eine Besonderheit besteht hierbei in der Umsetzung der Status-Monitor Komponente, welche durch fertige Komponenten für das Spring Ökosystem abgedeckt werden kann. Zu diesem Zweck kann sowohl das Spring-Unterprojekt Spring Actuator eingesetzt werden als auch das Drittanbieter-Frontend Spring Boot Admin der Firma codecentric[7]. Entsprechende Frontend Lösungen unterstützen im Normalfall auch die Überwachung mehrerer Instanzen von Springapplikation. Auf diese Art und Weise können anschaulich Statusinformationen der Plattform ausgelesen werden.

Anwendungsbeispiel: Primzahlberechnung

Die Berechnung von Primzahlen kann auf vielen unterschiedlichen Arten und Weisen erfolgen. Es ist möglich entsprechende Algorithmen zum Test einzelner Zahlen direkt in der Form von Referenznetzen zu implementieren. Da Modellierungswerkzeuge wie Petrinetze im Allgemeinen und Referenznetzen im Speziellen vor allem in komplexen Umgebungen ihre Stärken ausspielen können, ist die Modellierung eines Primzahltests ein hervorragender Kandidat für die Auslagerung in die Region von Plattformfunktionalitäten.

Als Beispiel ist daher in Abbildung 14.9 die Berechnung von Primzahlen durch Referenznetze dargestellt. Diese Netzdaten können durch das Cloud-Native Renew-Plugin den Simulator zur Laufzeit zur Verfügung gestellt werden, um eine entsprechende Berechnung durchzuführen.

[7] https://github.com/codecentric/spring-boot-admin (Zuletzt abgerufen am 14.10.2021)

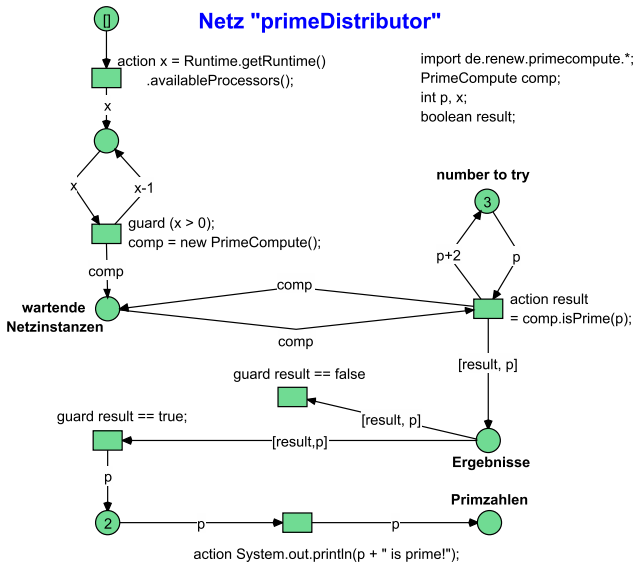

Abbildung 14.10.: Primzahlberechnung mit Plattformfunktionalität

Wie eingehend erläutert, ist jedoch die Auslagerung in eine entsprechende Bibliothek meist zielführender. Aus diesem Grund ist eine abgewandelte Fassung des Netzes in Abbildung 14.10 dargestellt. Sie erwartet die Existenz der Java-Klasse »PrimeCompute«, welche nicht standardmäßig auf einer Plattform bzw. einem Renew Simulator vertreten ist. Bei der webbasierten Anfrage an den entsprechenden Simulator dieses Netz zu simulieren wird es zu einem Fehler kommen, da die Funktionalität auf der Plattform nicht vorhanden ist.

Durch das Cloud-Native Renew-Plugin kann jedoch Abhilfe geschaffen werden. Die fehlende Funktionalität kann in Form eines Plugins in den Simulator hochgeladen werden und dort geladen werden. Nach dem Laden der spezifischen Funktionalität und insbesondere der Java-Klasse »PrimeCompute« kann auch dieses Netz simuliert werden.

Im größeren Kontext ist es nun denkbar, dass automatisiert fehlende Funktionalitäten nachgeliefert werden können, wenn es bei der Übertragung von Aufgaben an Plattformen zu Problemen kommt. Auf diese Weise setzt das Cloud-Native Renew-Plugin die von der Architektur geforderte Agilität und die Einsatzmöglichkeit in heterogenen Umgebungen um.

14.3. Prototypen zur Kommunikation

Die Prototypen der Kommunikationsinfrastruktur umfassen im Wesentlichen zwei Ansätze. Zum einen eine volle Implementation des Realisierungskonzepts bezüglich Petrinetz-Sagas, welche einen allgemeinen Ansatz verfolgt, welcher nicht spezifisch für agentenorientierte Softwareentwicklung ist. Das Ziel dabei ist der Einsatz in *generellen Microservice-Deployments* und als Spezialfall auch der mögliche Einsatz in der komplexen Kommunikation, wie sie von der Mushu-Architektur beschrieben wird.

In einem generellen Deployment kann jedoch nicht die Existenz eines Referenznetzsimulators als übergeordnete Einheit angenommen werden. Aus diesem Grund wird entgegen der Ausführungen in Abschnitt 14.2.2 die Integration von Renew in Spring angenommen. Auf diese Weise kann Renew als Bibliothek verpackt mitgeliefert werden. Die Argumentationen aus Abschnitt 14.2.2 bleiben jedoch für die Umsetzung von Mushu weiterhin gültig, sodass dort die Integration von Spring in Renew angestrebt werden sollte. Die grundlegenden Konzepte dieses Abschnitts sind hierdurch unberührt.

Der zweite Prototyp beschreibt die Ansätze von Resilient Distribute und dabei insbesondere die Realisierung einfacher Kommunikation mit der Hilfe eines Nachrichtenbrokers.

14.3.1. Petrinetz-Sagas auf Basis von Spring und Eventuate

Bei diesem Prototypen handelt es sich um die Umsetzung der Ideen bezüglich petrinetzbasierter Sagas. Das Ziel dabei ist die Bereitstellung einer Umgebung, bei der mithilfe vom Spring Framework und Renew (welches hierbei wie eingehend erläutert entgegen der Ausführungen in Abschnitt 14.2.2 als Bibliothek behandelt wird) eine Anwendung implementiert werden kann, die auf Microservice Prinzipien basiert und über ihre Knotengrenzen hinweg petrinetzbasierte Eventual Consistency realisiert. Zum Einsatz kommen dabei die Technologien Renew, Spring, das Eventuate Framework, Kafka, MySQL und PNML. Die noch nicht eingeführten Technologien Kafka und Eventuate werden im Folgenden kurz erläutert. Die Implementation erfolgte mit Unterstützung durch Manuela Buchholz und wurde ebenfalls in der zugehörigen Veröffentlichung (Röwekamp, Buchholz und Moldt, 2021) beschrieben.

Apache Kafka

Apache Kafka ist ein verteiltes System, welches zur Verarbeitung von Datenströmen eingesetzt wird. Es ist skalierbar und fehlertolerant, freie Software und

wurde ursprünglich bei der Firma LinkedIn entwickelt, ist nunmehr jedoch unter Verwaltung der Apache Software Foundation.

Das Herzstück der Anwendung ist ein verteiltes Transaktionslog. Kafka verfolgt den Ansatz der direkten persistenten Speicherung und hält selbst keine Daten im Hauptspeicher vor. Der Hintergrund dazu besteht darin, dass moderne Betriebssysteme leistungsstarke Cachingalgorithmen aufweisen und häufig abgefragte Datenbestände von persistenten Speichermedien unabhängig von Anwendung im Hauptspeicher vorhalten.

Kafka kann somit als Event-basierter Nachrichtenbroker verstanden und eingesetzt werden. Kafka wird ebenso in sehr vielen großen Unternehmen und Software Deployments eingesetzt und ist damit mehr oder weniger eine Standardlösung der Industrie für die genannten Anwendungsfälle.

Kafka basiert auf Apache Zookeeper, welches in gewissem Maße ein älteres Konkurrenzprodukt zu Kubernetes ist. Da sich jedoch zusehends Kubernetes gegen Apache Zookeeper in industriellen Anwendungen durchsetzt, bestehen Bestrebungen bei Kafka eine native Unterstützung von Kubernetes zu gewährleisten. Übergangslösungen existieren jedoch bereits heute und können im Kontext dieser Arbeit eingesetzt werden.

Eventuate Framework

Das Eventuate Framework bietet eine fortgeschrittene Implementation des klassischen Saga-Patterns. Speziell bei dem Unterprojekt Eventuate Tram Sagas wird eine direkte Integration mit bekannten Microservice Frameworks angestrebt. Entwickelt wird es federführend von Chris Richardson und während Eventuate selbst eine All Right Reserved Lizenz aufweist, ist der Code der Eventuate Projekte unter der Apache 2.0 Lizenz veröffentlicht.

Eventuate setzt auf Apache Kafka und MySQL auf. Im Kern implementiert es die für klassische Sagas typische State Machine. Eventuate setzt einen entsprechenden Weiterleitungsservice für Saga Events ein, welcher eine eigene Datenbank besitzt und dort gespeicherte Events an den Nachrichtenbroker Kafka überträgt. Services, welche durch die jeweiligen Events angesprochen werden, erhalten ihre Notifikation über die jeweilige Kafka Implementation.

Umsetzung

Für die Implementation von Sagas stehen bereits einige öffentlich verfügbare Frameworks bereit. Namentlich wären hierbei Axon, Eventuate, microprofile-lra und Narayana zu nennen. Der Beitrag (Štefanko, Chaloupka und Rossi,

2019) gibt in der Form eines Surveys eine gute Übersicht über die technischen Implementationen. Ohne in die Details der jeweiligen technischen Implementation einzutauchen, wurde aus den Möglichkeiten das Eventuate Framework ausgewählt, da es sehr speziell auf den Saga Anwendungsfall zugeschnitten ist und gleichzeitig direkte Integration mit dem Spring Framework bietet. Der verwendete Code unterliegt ebenso der Apache 2.0 Lizenz.

Die Abläufe innerhalb der Implementation entsprechen sehr genau den Vorüberlegungen aus den beiden Abbildungen 12.9 und 12.10. Zunächst wird die als P/T Netz codierte Saga im PNML-Format geparsed und in eine interne Datenrepräsentation überführt, welche die nachfolgenden Operationen wesentlich einfacher gestalten. Die Datenrepräsentation entspricht im Wesentlichen einer doppelt verlinkten Abbildung des Graphen, bestehend aus Plätzen, Transitionen und Kanten. Die Struktur wird danach exakt nach dem Ablauf aus Abbildung 12.9 in ein entsprechendes Referenznetzen transformiert und als PNML zurückgeliefert. An diesem Punkt geht der Kontrollfluss an die Anwendung zurück, welche mit dem generierten PNML nach Belieben verfahren kann.

Mit dem generierten Referenznetz-PNML oder einem anderen, bereits vorbereiteten Referenznetz-PNML, kann nun eine entsprechende Saga registriert werden. Geht nun eine entsprechende Anfrage beim System ein, wird eine neue Netzinstanz der registrierten Referenznetz-PNML Saga von der Implementation erzeugt und zur Simulation in Renew gegeben. Bei der Simulation werden nun nacheinander die einzelnen Serviceaufrufe in Form von Transitionen gefeuert und die Abarbeitung dieser Vorgänge wird entsprechend Abbildung 12.10 ausgeführt. Schlussendlich wird die Saga entweder komplett ausgeführt und ist im Zustand »fertig« angekommen, oder aber es erfolgt ein Fehlschlag vor dem Pivotelement und die Saga wird insgesamt zurückgerollt.

Die bereitgestellte Umgebung für die Entwicklung einer Anwendung inklusive petrinetzbasierten Sagas umfasst den Einsatz des Buildtools Gradle, Renew, Spring, Eventuate und eine Anbindung an Kafka. Bei der Ausführung muss darauf geachtet werden, die entsprechenden externen Systeme für die Abarbeitung von Eventuate Tram Sagas ebenfalls hochzufahren. Dabei handelt es sich um Apache Kafka (mit Apache Zookeeper), MySQL, dem verteilten Tracing-Tool Zipkin[8] sowie dem Eventuate Tram CDC Service[9].

14.3.2. Resilient Distribute

Während der Gesamtentwurf von Resilient Distribute sowohl die einfache als auch die komplexe Kommunikation abdeckt, umfasst der gegenwärtig implementierte

[8] https://zipkin.io/ - Alle URLs der Seite zuletzt abgerufen am 17.10.2021
[9] https://eventuate.io/docs/manual/eventuate-tram/latest/cdc-configuration.html

Prototyp bisher nur die einfache Kommunikation zwischen Agenten. Die Umsetzung erfolgte mit der Unterstützung von Alexander Senger im Rahmen der Bachelorarbeit (Senger, 2021). Die Realisierung fokussiert dabei den in Abschnitt 12.7.4 vorgestellten Ansatz entsprechende Adapternetze einzusetzen. Die Semantik des Aufrufs entspricht der Semantik der einfachen Kommunikation, wie sie im Rahmen der Arbeit definiert ist. Vom Aufrufer wird eine Nachricht abgeschickt und diese ohne Zustellungsgarantie unilateral an einen oder mehrere Empfänger gesendet.

Intern setzt die bestehende Implementation auf dem Spring Framework und Apache Kafka auf. Dabei ist neben dem schlichten Versand von Nachrichten die gesamte Identifikation und globale Eindeutigkeit von Simulationen, Plattform und Agent/Netzinstanz umgesetzt. Die Kommunikation zwischen Kafka und der Referenznetzsimulation ist durch besagte Adapternetze realisiert, zu deren Instanzen mittels Netzstubs Daten aus Kafka übermittelt werden können. Bei Netzstubs handelt es sich um spezialisierte Netzinstanzen, bei welchen einzelne ihrer synchronen Kanäle auch mittels Java Code an der Instanz aufgerufen werden können. Weitere Informationen zu Netzstubs finden sich im Renew User Guide (Kummer, Wienberg, Duvigneau, Cabac u. a., 2020b) in Abschnitt 3.12.6.

14.4. Subprotokoll: Binary Squaring und OpenSIFT

Die Theorie bezüglich Binary Squares wurde im Rahmen der Arbeit exemplarisch in einen entsprechenden Mustererkennungsalgorithmus eingefügt. Wie in Kapitel 13 zur Umsetzung bereits erwähnt wurde, ist Scale Invariant Feature Transform (SIFT) ein solcher Algorithmus. Die Bibliothek OpenSIFT (Hess, 2010) basiert direkt auf den Erkenntnissen zu SIFT selbst und ist in der Programmiersprache C verfasst. Die ursprüngliche Implementation der Bibliothek verwendet eine Methode, die auf sogenannten *kd*-Trees[10] basiert, um eine nächste Nachbar-Suche auszuführen. Asymptotisch besitzt der *kd*-Tree Algorithmus eine Laufzeit in Landau-Notation von $O(n \log n)$.

Die innerhalb der Bibliothek vom Autor der Arbeit implementierte Variante setzt hingegen eine naive Variante des Vergleichsalgorithmus ein, welche lediglich alle Punktpaare miteinander vergleicht. Die asymptotische Laufzeit dieses Ansatzes beträgt bekanntermaßen $O(n^2)$.

Da die Theorie von Binary Squares an einer anderen Stelle als der Suchraumreduktion der zu vergleichenden Punkte ansetzt, ist dies jedoch von geringerem Belangen. Einer vollwertigen Implementation eines $O(n \log n)$ basierten Algorithmus inklusive Binary Squares steht prinzipiell nichts entgegen, er ist nur weitaus

[10] Für weitere Informationen zu *kd*-Trees siehe die Originalveröffentlichung (Bentley, 1975).

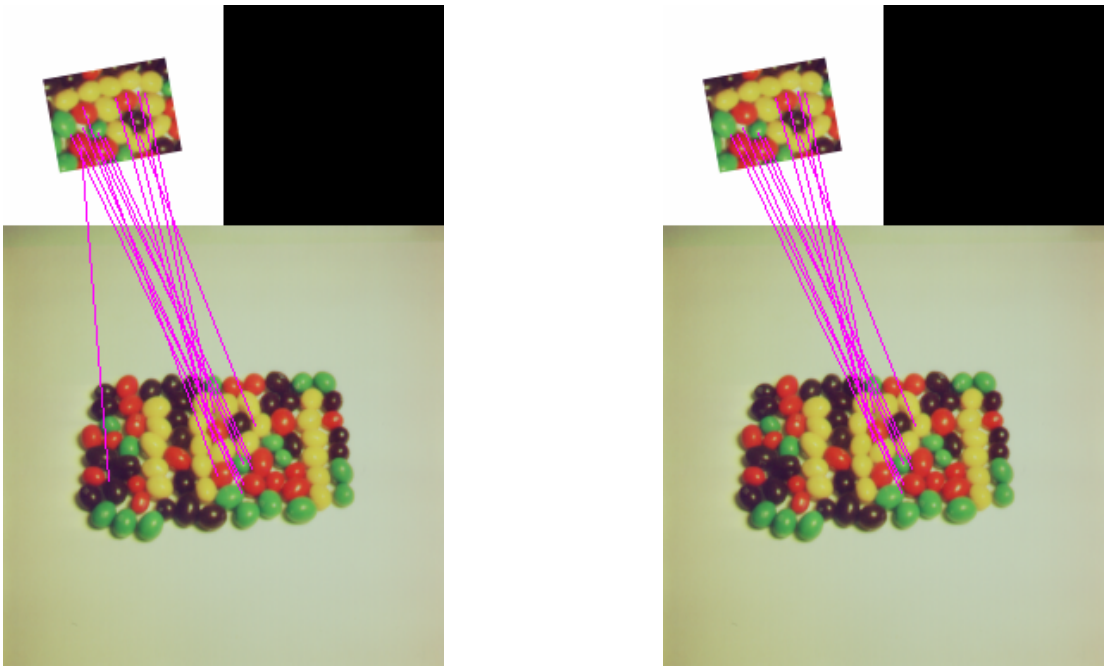

Abbildung 14.11.: Vergleich Original OpenSIFT vs. Binary Square OpenSIFT. Das Bild ist Eigentum von USC-SIPI.

aufwendiger zu implementieren als der naive Vergleich. Hilfreich an dieser Stelle ist auch, dass praktisch relevante Beispieldaten häufig vergleichsweise klein ausfallen und mitunter konstante Faktoren in den Laufzeiten größere Auswirkungen haben als der schwerwiegendste Faktor der Landau-Notation.

Anstatt konventionelle Quadrate einzusetzen, gleicht der abgewandelte Algorithmus Summen von Binary Squares gegen den halbierten Schwellenwert ab. Trotz des asymptotisch schlechteren Algorithmus ergaben sich in einigen Beispiel Anwendungen Laufzeitverkürzung von 50-75%. Dabei wurde noch nicht einmal die Restrukturierung nach höchstwertigen Bits mit umgesetzt. Weitere Untersuchungen könnten daher die geschickte Integration von Binary Squares in umfassendere Algorithmen beinhalten.

Abbildung 14.11 zeigt exemplarisch die Ergebnisse auf einem Beispielbild. Auf der linken Seite ist die originale Berechnung durch OpenSIFT zu erkennen und auf der rechten Seite die Variante durch Binary Squares. Auf den Bildern ist jeweils linksoben ein rotierter Ausschnitt (Input) aus dem unteren Bild (Referenzdatei) gezeigt, während die magentafarbenen Linien von den jeweiligen Algorithmen berechnete Übereinstimmungen von Bereichen darstellen. Im Originalalgorithmus ist ganz links ein falsch-positives Ergebnis abgebildet. Das schwarze Quadrat rechtsoben zeigt nur die Abwesenheit von Bilddaten an dieser Stelle an und hat keine weitere Bedeutung. In diesem Beispiel benötigte die Berechnung auf der

rechten Seite ca. 25% der Zeit, welche für die Berechnung der linken Seite nötig waren.

Im Rahmen der rudimentären Implementation wurde auf eine vollwertige Evaluation bezüglich Genauigkeit und Laufzeitverhalten verzichtet, da wie eben beschrieben viele offensichtliche Optimierungsmöglichkeiten des Algorithmus bestehen und eine Laufzeitevaluation keinen langen Bestand hätte. Es soll damit lediglich die potenzielle Möglichkeit bzw. Mächtigkeit der Methode motiviert werden.

14.5. Zusammenfassung - Prototypen

In diesem Kapitel wurden alle implementierten Prototypen im Rahmen der Arbeit vorgestellt. Als Umsetzungen und Vorläufer des Plattformmanagements wurden virtualisierte und containerisierte Lösungen betrachtet sowie die Umsetzung RenewKube, welche die wesentlichen Merkmale zur Skalierungssteuerung abbildet. Distributed Analysis setzt auf RenewKube und MoMoC auf und implementiert eine skalierbare, verteilte Berechnung von Erreichbarkeitsgraphen. Ein Vorläufer der Plattform auf der Basis von replizierten Netzdatenbanken wurde vorgestellt sowie die schlussendliche Umsetzung eines Großteils der Mushu-Plattform in Form des Cloud-Native Renew-Plugins. Bezüglich der Agentenkommunikation wurde eine Implementation des Realisierungskonzepts zu Petrinetz-basierten Sagas auf der Basis von Eventuate und dem Spring Framework vorgestellt. Eine Teilimplementation von Resilient Distribute wurde ebenfalls beschrieben. Im Kontext der Subprotokolle wurde die grundlegende Performanceverbesserung in der Mustererkennung durch Binary Squares anhand eines exemplarischen Algorithmus in der Bibliothek OpenSIFT motiviert.

15. Bewertung und Diskussion

Mit der abschließenden Vorstellung der Prototypen liegt die gesamte Bearbeitung der in Kapitel 4 vorgestellten offenen Punkte vor. Auf dieser Basis kann eine Evaluation der geleisteten Arbeit auf der Basis der herausgearbeiteten Anforderung erfolgen. Dieses Kapitel dient ebendiesem Zweck und ist dafür so organisiert, dass zunächst eine Übersicht über Konzepte, Realisierungskonzepte und Prototypen gegeben wird. Anschließend werden die Anforderungen erneut aufgegriffen und jede Anforderung wird sowohl auf konzeptueller als auch auf der Umsetzungsebene im Kontext von Mushu evaluiert. Darüber hinaus werden die Forschungsfragen erneut aufgegriffen und diskutiert, inwieweit diese durch die Untersuchungen beantwortet werden konnten. Abschließend werden noch einmal Stärken der Mushu-Architektur herausgestellt sowie offene Punkte adressiert, welche sich im Kontext der Arbeit ergeben haben oder deren Betrachtung nahe liegt, diese jedoch nicht im Detail bearbeitet wurden.

15.1. Abbildung vom Konzept auf die Realisierung

Dieser Abschnitt dient der Rekapitulation und Verbindung zwischen der konzeptuellen Mushu-Architektur, wie sie in Kapitel 5 bis Kapitel 8 beschrieben wurde, mit dem Realisierungskonzept aus Kapitel 9 bis Kapitel 13 sowie mit den Prototypen aus Kapitel 14. Die zentralen Zusammenhänge sind in ihrer Gesamtheit in Tabelle 15.1 aufgeführt.

Die jeweiligen Einträge in den Spalten zum Konzept und zum Realisierungskonzept weisen keine besondere Formatierung auf. Sie entsprechen den jeweiligen Ausführungen der soeben referenzierten Kapitel. Bei den konzeptuellen Konstrukten handelt es sich im Wesentlichen entweder um eigens für diesen Kontext erdachte Konzepte oder aber für den Kontext der Arbeit abgewandelte bekannte Konzepte. Dabei gibt es nur einige Ausnahmen, wie beispielsweise die plattforminterne Kommunikation, welche auch konzeptuell von Mulan übernommen wurde.

Die Arbeitsleistung bezüglich der Elemente des Realisierungskonzepts besteht in der Abbildung der konzeptuellen Konstrukte auf konkrete Systeme, welche eine technischere Ausrichtung besitzen. Dabei wurden teilweise neue Komponenten erdacht, wie beispielsweise das Management-System, das Anbindungsplugin, Re-

Konzeptuelles Konstrukt	*Realisierungskonzept*	*Technologie im Prototyp*
Plattformskalierung (reaktiv)	Clustermanager	Kubernetes
	Management-System	**RenewKube Manager**
	Anbindungsplugin	**RenewKube Plugin**
Plattformskalierung (proaktiv)	Lastüberwachung / Autoscaling	*(z.B. Kubernetes Horizontal Pod Autoscaler)*
Plattformdefinitions-update	CI/CD Server	GitLab
Plattformumzug	Portabilität von Containern	Kubernetes
Zustand (Plattformmanagement)	Status-Monitor	Spring Boot Admin
Plattformübergreifende einfache Kommunikation	direkt bzw. Nachrichtenbroker	HTTP bzw. Apache Kafka
Plattformübergreifende (komplexe) Kommunikation (Enthält die Konzepte »Ablaufmanager« und »Nachrichtenbroker«)	**Petrinetz-Sagas**	**Petrinetz-Saga-Plugin**
		Eventuate Petrinetz-Saga Core
	Resilient Distribute	**Cloud-Native Renew-Plugin + Kafka**
		(Renew + Petrinetz-Saga-Plugin + Eventuate + Apache Kafka)
Plattformfunktionalitäten	RENEW-Plugins	**Cloud-Native Renew-Plugin**
Erweiterung von Plattformfunktionalitäten	RENEW-Plugins hochladen	
Agentendefinitionen / Agenten erzeugen	Netze instanziieren	
Erweiterung von Agentendefinitionen	Netze hochladen	
Zustand (Plattform)	Statussammlung	**Cloud-Native Status Monitor Plugin**
	True-Concurrency Checkpoints	*(Neuimplementation)*
	Simulationsfeed	*(Neuimplementation)*
Plattforminterne Kommunikation	Synchronisation	RENEW synchrone Kanäle
Agentenumzug	Netze instanziieren + Austausch	*(Cloud-Native Renew-Plugin)*
Anwendungsbeispiel	**Binary Squaring**	**In OpenSIFT**

Tabelle 15.1.: Konzepte, Realisierungskonzepte und Prototypen im Kontext der Gesamtheit der MUSHU-Architektur.

silient Distribute oder Binary Squaring. Bei vielen Komponenten wurde aber auch argumentiert, warum und wie bestimmte bestehende Realisierung in diesem Kontext eingesetzt werden können. Die entsprechend gänzlich neuen Realisierungskonzepte sind in der Tabelle **fett** hervorgehoben.

In der letzten Spalte sind die jeweils für ein bestimmtes Konzept eingesetzten Technologien in den jeweils zugehörigen Prototypen referenziert. Dabei wurden einige Komponenten von Grund auf eigens für die Realisierung von Mushu implementiert, wie beispielsweise das RenewKube Ökosystem, Petrinetz-Sagas sowie das Cloud-Native Renew-Plugin. Gänzlich neu implementierte Anteile sind **fett** hervorgehoben. Für den jeweiligen Anwendungsfall konfigurierte und eingesetzte Technologien und Werkzeuge sind in normalem Schriftsatz aufgeführt. Komponenten, bei denen lediglich das Realisierungskonzept ausgeführt wurde, sind mit einer beispielhaften Idee für die Implementation *(kursiv und eingeklammert)* dargestellt.

Die einzelnen Abbildungen zwischen Konzept, Realisierungskonzepts und Prototyp sollen hier nicht im Detail ein weiteres Mal erläutert werden. Für die jeweilige Abbildung sei auf die Ausführungen in den zu Beginn dieses Abschnitts referenzierten Kapiteln verwiesen.

15.2. Erfüllung der Anforderungen an die Architektur

In Analogie zur Einführung der betrachteten Anforderungen werden diese ebenfalls bei der Evaluation in diesem Abschnitt in Kategorien aufgeteilt. Diese Kategorien umfassen Anforderungen bezüglich der Interaktionsmöglichkeiten von Agenten, der Struktur des abgebildeten Systems und der Architektur, der Dynamik des Systems und der Anforderungsdomäne. Die Erfüllung jeder Anforderung wird je auf der konzeptuellen Ebene sowie der Realisierungsebene überprüft.

15.2.1. Interaktionsbasierte Anforderungen

Interaktionsbasierte Anforderungen wurden in Abschnitt 4.2.1 beschrieben und umfassen alle Anforderungen bezüglich den Interaktionsmöglichkeiten von Agenten innerhalb der Architektur.

Anforderung I1: »*Agenten können direkten Einfluss auf die Erzeugung und Vernichtung von Plattformen nehmen.*«

Innerhalb der Mushu-Architektur residieren Agenten auf Plattformen. Plattformen wiederum sind in der darüber gelegenen Ebene des Plattformmanagements

eingebettet. Analog zur Mulan-Architektur kann auch in Mushu jede Einheit als Agent betrachtet werden, welcher Nachrichten empfangen und senden kann.

Das Plattformmanagement bietet mit seiner Möglichkeit, eine Nachricht zum reaktiven Skalieren von Plattformen zu empfangen, die Grundlage für Skalierungsoperationen. Die Erzeugung und Zerstörung von Plattformen entspricht diesen auf Plattformen bezogenen Skalierungsoperationen. Die Möglichkeit zum Empfang der Nachrichten ist nicht auf einen bestimmten Absender begrenzt, sondern kann potenziell von einem beliebigen Absender erfolgen. Aus diesem Grund sind auch alle Agenten in der Lage, zumindest den Wunsch zur Skalierung von Plattformen an das Plattformmanagement zu äußern. Da das Plattformmanagement dennoch bezüglich der Anfrage aufgrund von internen Zuständen eine Entscheidung treffen kann, erfolgt nicht aufgrund jeder einzelnen Skalierungsanfrage immer auch eine Skalierung der Anzahl von Plattformen.

Dennoch lässt sich festhalten, dass die konzeptuellen Ausführungen der Mushu-Architektur Agenten im Allgemeinen die Möglichkeit einräumen, aktiv Einfluss auf die Erzeugung und Zerstörung von Plattformen zu nehmen.

Im Realisierungskonzept wird die reaktive Plattformskalierung auf drei Komponenten abgebildet: einen Clustermanager, ein Anbindungsplugin und ein Management-System. Auf der Realisierungsebene werden Agenten in der Form von Netzinstanzen in Renew Instanzen simuliert. Entlang der Mushu-Architektur kann von entsprechenden Instanzen mithilfe des Anbindungsplugins, welches selbst wiederum eine Plattformfunktionalität darstellt, ein Skalierungsantrag an das Management-System gestellt werden. Zusammen mit dem Clustermanager wird die Skalierungsoperation ausgeführt. Als Prototyp übernimmt ein neu implementiertes System mit dem Namen RenewKube die Rolle des Management-Systems und Kubernetes die Rolle des Clustermanagers.

Zusammenfassend wird die Anforderung **I1** sowohl auf konzeptueller als auch auf der Realisierungsebene vollständig erfüllt.

Anforderung I2: »*Agenten können mit Plattformen in Interaktion treten.*«

In den bisherigen Ausführungen wurde bereits dargelegt, dass Plattformen im Rahmen der Mushu-Architektur insbesondere auch als Agenten behandelt werden können. Dabei existieren im Rahmen der Mushu-Architektur verschiedene Möglichkeiten für Agenten, um mit Plattformen zu interagieren. Die einfachste Möglichkeit besteht darin, auf vorhandene Plattformfunktionalitäten zuzugreifen. Dabei handelt es sich meist um statische Fähigkeiten oder Algorithmen, welche von der Plattform speziell für die auf ihr ansässigen Agenten zur Verfügung gestellt werden.

Darüber hinaus können Agenten auch Nachrichten direkt an Plattformen adressieren. Diese Methode der Interaktion kommt meist dann zum Einsatz, wenn eine

Interaktion mit einer Plattform angestrebt wird, auf der sich der Agent bisher (noch) nicht befindet. Die für den Agenten lokale Plattform leitet die Nachricht entsprechend weiter. In diesem Fall übernimmt das Plattformmanagement entlang der Einheitentheorie für diese Interaktion die Rolle der Plattform. Auf diese Art können Agenten in der konzeptuellen Architektur von Mushu auf vielfältige Weise in Interaktionen mit Plattformen treten.

Auf der Ebene des Realisierungskonzepts wird ein großer Teil der Interaktion mit der Plattform durch den Simulator Renew selbst und durch das Hochladen und Aktivieren von sowohl Netzdefinitionen als auch Plugins gelöst. Im Prototyp werden diese Eigenschaften vom Cloud-Native Renew-Plugin implementiert. Zusätzlich kann bezogen auf Statusinformationen auf die Statussammlung und auf das Cloud-Native Status Monitor Plugin zurückgegriffen werden. Im Rahmen des Realisierungskonzepts bieten True-Concurrency Checkpoints und der Simulationsfeed die Möglichkeit, Informationen zum technischen und fachlichen Voranschreiten der Simulation für daran interessierte Agenten transparent zu gestalten.

Insgesamt kann somit festgehalten werden, dass die Mushu-Architektur die Interaktion von Agenten mit Plattformen unterstützt und damit Anforderung **I2** erfüllt. Das Realisierungskonzepts zeigt einen Rahmen auf, wie die abstrakten konzeptuellen Interaktionen in konkrete Systeme übersetzt werden können.

Anforderung I3: »*Agenten können über Plattformgrenzen hinweg kooperieren und kommunizieren.*«

Während die Plattform in ihrer Realweltabbildung den Raum für die Kommunikation zwischen zwei Gesprächspartnern darstellt, so muss jedoch stets ein Gesprächsteilnehmer auch über die Einladung in einen solchen Raum informiert werden können. Dafür können übergeordnete Strukturen eingesetzt werden und ebenfalls bereits existierende Plattformen. Analog dazu können Agenten auch in der Mushu-Architektur gezielt Nachrichten an Agenten auf anderen Plattformen senden. Die gemeinsame übergeordnete Plattform wird in diesem Fall durch entweder das Plattformmanagement oder im Falle mehrerer zusammengeschalteter Plattformmanagements durch die abstrakt belassene Infrastruktur realisiert.

Je nach intendierter Komplexität des Austausches sieht das Mushu-Plattformmanagement zwei Formen der Kommunikation vor: Einfache Kommunikation, welche sich durch unilateralen Informationsaustausch und die Abwesenheit von Zustellungsgarantien auszeichnet, sowie komplexe Kommunikation, welche multilateralen Informationsaustausch und Zustellungsgarantien mit einbezieht. Für beide Interaktionsformen existieren entsprechende Konstrukte im konzeptuellen Modell von Mushu. Dabei werden für komplexe Kommunikation spezialisierte Ablaufmanager instanziiert, welche den Austausch steuern, mit temporären Ausfällen einzelner Plattformen umgehen können und ebenfalls für eine lose Kopplung der Plattformen im Gesamtsystem sorgen.

Auf der Realisierungsebene steht mit dem Konzept von Petrinetz-Sagas eine starke Umsetzung bereit, welche alle Voraussetzungen erfüllt, um eine komplexe Kommunikation, wie sie in Mushu beschrieben wird, zu realisieren. Die Kommunikation selbst erfolgt auf der Basis verteilter synchroner Kanäle, wie sie bereits durch das Distribute Plugin in (M. Simon und Moldt, 2016) beschrieben wurden, und erweitert diese für eine Unterstützung von langfristig laufenden verteilten Transaktionen (Sagas), bei der einzelne Plattformen temporär nicht erreichbar sein können. Die so kombinierte Realisierung erhielt im Rahmen der Arbeit die Bezeichnung »Resilient Distribute« und wurde für einfache Kommunikation als Prototyp umgesetzt.

Plattformübergreifende Kommunikation zwischen Agenten ist nach der Annahme der Abbildung der Realwelt möglich, sollte aber im Normalfall nicht häufiger auftreten als plattformlokale Kommunikation. Auf Basis dieses Umstandes wurde die Umsetzung auf konzeptueller und Realisierungsebene in Mushu gewählt. Mushu unterstützt damit vollumfänglich plattformübergreifende Kommunikation zwischen Agenten und erfüllt damit insbesondere auch die Anforderung **I3**.

15.2.2. Strukturelle Anforderungen

Strukturelle Anforderungen umfassen alle Anforderungen an die Struktur der Architektur. Sie wurden in Abschnitt 4.2.2 vorgestellt und umfassen gesetzte Mindestvoraussetzungen bezüglich der Modellierung von Realweltstrukturen durch die Architektur.

Anforderung S1: »*Die hierarchische Struktur zwischen Agenten und Plattformen wird abgebildet.*«

Die Abbildung der hierarchischen Struktur zwischen Agenten und Plattformen erfolgt auf der konzeptuellen Ebene der Mushu-Architektur durch den Einsatz des objektnetzbasierten Petrinetzformalismus der Referenznetze. In Referenznetzen beinhalten Marken im Netz Referenzen auf andere Objekte bzw. Netzinstanzen. Da Agenten auf einer Plattform beheimatet sind, liegt es nahe, dass die Plattform in der Modellierung entsprechend Agenten als Einheiten referenzieren und handhaben kann. Die Kommunikation an dieser Stelle erfolgt durch den Einsatz synchroner Kanäle, bei denen hierarchisch höher gelegene Transitionen synchron mit niedrigeren Transitionen geschaltet werden können. Dabei kann ein multilateraler Informationsaustausch erfolgen.

An dieser Stelle setzt die Mushu-Architektur auf den bereits bestehenden Ausführungen der Mulan-Architektur auf und kann die dort bereits bestehenden Modellierungen übernehmen. In der Realisierung werden diese Zusammenhänge nach wie vor durch den Simulator Renew und seine Plugins abgebildet. Mushu erfüllt somit die Anforderung **S1**.

Anforderung S2: »*Plattformen können heterogen sein.*«

Mushu Plattformen weisen als ein zentrales Konstrukt eine Sammlung an verschiedenen Plattformfunktionalitäten auf. Dabei handelt es sich um bestimmte Ausprägung der Plattform, die sie von der grundlegenden Blaupause aller Plattformen unterscheidet. In der direkten Analogie aus der Realweltableitung entsprechen diese Plattformfunktionalitäten lokalen Gegebenheiten in Kommunikationsräumen. Plattformfunktionalitäten können daher beispielsweise Tafeln, die Aufzeichnungsfähigkeit eines Onlinekonferenzraums, die Zustellungsabsicherung auf der Transportebene einer Netzwerkverbindung, die Interaktion mit der Hardware eines speziellen Servers und weitere sein.

Durch die beliebige Menge an individuellen Plattformfunktionalitäten ist es in der konzeptuellen Betrachtung möglich, jede beliebige Form von Plattform zu realisieren. Mushu Plattformen besitzen daher die Möglichkeit, hervorragend Heterogenität abzubilden.

In der Realisierung übernimmt diese Funktion das Plugin-System von Renew. Semantisch handelt es sich dabei um ein äquivalentes Konstrukt, da Simulatoren erweitert werden können und Plugins entsprechend optional verwendbare Funktionalitäten bereitstellen. Eine Instanz des Simulators Renew kann folglich einer anderen gleichen, muss dies aber nicht. Die Auswahl der geladenen Plugins bestimmt maßgeblich die Ausprägung der konkreten instanziierten Plattform.

Zusammenfassend kann also festgehalten werden, dass Mushu sowohl konzeptuell als auch in der Realisierung Heterogenität unterstützt und damit Anforderung **S2** erfüllt.

15.2.3. Dynamische Anforderungen

Dynamische Anforderungen wurden in Abschnitt 4.2.3 vorgestellt und umfassen alle Voraussetzungen bezüglich der Dynamik des Systems. Eine Lösung erfordert eine grundlegende Unterstützung von dynamischen Prozessen zur Laufzeit. Ohne Beachtung vonseiten der Architektur ist dies nur schwerlich umsetzbar.

Anforderung D1: »*Die Funktionalitäten von Plattformen können dynamisch angepasst werden.*«

Wie bereits im vergangenen Abschnitt erläutert, setzt Mushu das Konzept von Plattformfunktionalitäten ein, um Heterogenität in der Plattformlandschaft darstellen zu können. Ebenso wie in der realen Welt können Plattformen Erweiterungen und Änderungen unterworfen sein. Diesen Umstand bildet Mushu ebenfalls explizit ab, indem einer gegebenen Plattform eine Nachricht zugestellt werden kann, auf deren Basis sie ihre Plattformfunktionalitäten erweitern bzw. anpassen kann.

Mushu gewährleistet daher insbesondere im Kontext der dynamischen Skalierung der Plattformmenge eine fortwährende Unterstützung der Heterogenität der Plattformmenge. Ohne entsprechende dynamische Anpassungsmöglichkeiten müssten für jede Ausprägung entsprechende Plattformdefinitionen vorgehalten werden. Da dies für beliebige Plattformmengen unmöglich erfüllt werden kann, muss eine dynamische Anpassbarkeit von Plattformen gewährleistet werden, wie es bei Mushu der Fall ist.

Bezogen auf die Realisierung von Mushu wird diese Fähigkeit wie bereits beschrieben durch Renew-Plugins abgebildet. Bisher konnten derartige Plugins zwar geladen werden, jedoch war dies nur lokal auf der Plattform möglich und auch nur bei bereits im Vorhinein vorliegenden Plugins. Durch die Umsetzung einer entsprechenden API für den Upload und das Laden von Renew-Plugins wurde eine entsprechende Möglichkeit geschaffen, Plattformen Nachrichten zukommen zu lassen, wie es die Mushu-Architektur vorsieht.

In der prototypischen Implementation ermöglicht das Cloud-Native Renew-Plugin auf der Basis von Java Spring den Upload und das Laden von Plugins via HTTP. Dabei handelt es sich im aktuellen Zustand nur um eine Erweiterungsmöglichkeit, die Pluginentfernung und -anpassung ist jedoch an anderer Stelle Gegenstand von Untersuchungen. Erste Entwicklungen beschreibt die Arbeit (Janneck, 2021). HTTP entspricht an dieser Stelle semantisch der einfachen Kommunikation, obwohl es streng genommen mit Zustellungsgarantien arbeitet und daher etwas mehr leistet als die einfache Kommunikation fordert.

Insgesamt kann somit festgestellt werden, dass Mushu die dynamische Anpassung von Plattformen unterstützt und damit insbesondere auch Anforderung **D1** erfüllt.

Anforderung D2: »*Agenten können Plattformen wechseln.*«

Die Mushu-Architektur sieht neben dem Erzeugen und Zerstören von Agenten explizit auch den Umzug von Agenten vor. Dabei orientiert sich die Konzeptualisierung an bereits bestehenden Arbeiten, wie in den zugehörigen Abschnitten motiviert wurde. Ein Agent kann somit umziehen, sobald er keine aktiven Protokolle oder ausstehende Nachrichten auf der lokalen Plattform mehr besitzt bzw. erwartet.

Im Realisierungskonzept kann ein Agent, welcher einen Umzug erwägt, die lokale Plattform bitten, seine entsprechende Agentendefinition an eine entfernte Plattform zu übermitteln, welche als Ziel des Umzugs fungiert. In einer speziellen Aktion kann der Agent dann seine lokale Zerstörung veranlassen und eine Nachricht an die Zielplattform mit seinem internen Zustand senden. Die Zielplattform instanziiert sodann einen neuen Agenten und übermittelt diesem den übergebenen internen Zustand.

Die Übermittlung und Instantiierung von Agentendefinitionen wird prototypisch durch das Cloud-Native RENEW-Plugin realisiert. Der Wechsel von Plattformen wird somit durch MUSHU unterstützt und somit erfüllt MUSHU ebenfalls Anforderung **D2**.

Anforderung D3: »*Der Nachrichtenversand erfolgt für die Agenten auf technischer Ebene transparent.*«

Agenten kommunizieren mittels Nachrichten. In der MUSHU-Architektur wird eine abzusendende Nachricht zunächst an die lokale Plattform des Agenten übergeben. Dabei wird der Zielagent der Nachricht vom lokalen Agenten vermerkt und die Nachrichten wird entweder lokal oder über das Plattformmanagement zugestellt. Jedoch sind all diese Aspekte in den höheren Hierarchieebenen der MUSHU-Architektur vor dem Agenten verborgen, für den diese Aktionen transparent ablaufen.

Auch auf der Ebene des Realisierungskonzepts kommen an dieser Stelle globale Identifikatoren zum Einsatz. In der einfachen Kommunikation zu Plattformen (welche ebenfalls als Agent interpretiert werden können) werden diese direkt angesprochen. Die Identifikation der Plattform kann über das Nachrichtensystem ermittelt werden. Dabei wird die absolute Position der Plattform dynamisch aufgelöst.

Analog verhält es sich bei der Kommunikation mit anderen Agenten auf anderen Plattformen. Hierbei kommt unabhängig von komplexer oder einfacher Kommunikation das Nachrichtensystem als Zwischenschritt für die Zustellung der Nachricht ins Spiel. Agenten können dort indirekt durch ihre Plattformen Nachrichten für sie entgegennehmen. An keiner Stelle muss ein Agent für die Übermittlung einer Nachricht die konkrete Position des anderen Agenten im System kennen.

Zusammengefasst kann daher festgehalten werden, dass MUSHU den technischen Versand von Nachrichten transparent den Agenten gegenüber gestaltet und somit Anforderung **D3** erfüllt.

15.2.4. Anwendungsdomäne

Die letzte Anforderung bezieht sich auf die Einsetzbarkeit der Architektur in spezifischen Anwendungsszenarien.

Anforderung A1: »*Die Architektur sollte mindestens für einen Einsatz in der Anwendungsdomäne der Szenenanalyse geeignet sein.*«

Die MUSHU-Architektur unterstützt bestimmte Anwendungen der Szenenanalyse auf verschiedenen Ebenen. Zum einen können durch die Abbildung auf Agentensysteme Interaktionen zwischen verschiedenen Teilnehmern bzw. Menschen mo-

delliert werden. Dabei können sowohl die Agenten als auch der Raum, in welchem sie sich befinden und interagieren, modelliert werden. Eigenschaften des Raums können als Plattformfunktionalitäten abgebildet werden, welche durch Agenten beeinflusst werden können.

Aus den simulierten Handlungen der Agenten können sodann Hypothesen abgeleitet werden. Speziell durch die Mushu-Architektur kann hierbei nun auch die Erzeugung bzw. Eröffnung sowie die Schließung von Räumen auf ein entsprechendes Modell der Architektur abgebildet werden. Auch der Wechsel von Räumen kann durch Agentenumzüge abgebildet werden.

Darüber hinaus kann auch auf der Ebene der Berechnung für derartige Systeme ein Beitrag geleistet werden. Die Sicht verschiedener Agenten auf eine bestimmte Szene kann durch den Einsatz einer heterogenen Plattformlandschaft modelliert werden. So kann die Plattform, auf der sich ein jeder Agent befindet, dem Agenten eine bestimmte Sicht auf die Szene durch Plattformfunktionalitäten ermöglichen. Als Realweltäquivalent könnten für derartige Plattformfunktionalitäten beispielsweise verschiedene Sensoren einer oder mehrerer Aufnahmeeinheit(en) genannt werden (Stichwort: »Sensor Fusion«).

Auf Basis der Rohdaten können über verschiedene Prozesse mit weiteren Inputs wie etwa Wissensdatenbanken Hypothesen konstruiert werden. An dieser Stelle kommen verschiedene Abläufe zusammen, welche im Rahmen der Mushu-Architektur als Protokolle von Agenten oder als deren Subprotokolle modelliert werden können. Ein Agent ist hierbei eine Einheit des Systems, welche die Algorithmen berechnet. So könnte beispielsweise eine Teilaufgabe einen Mustererkennungsalgorithmus wie beispielsweise SIFT umfassen. Zu diesem Beispiel wurde im Rahmen der Arbeit die Theorie zu Binary Squares vorgestellt, welche sich auf der Subprotokollebene von Agenten in der Mushu-Architektur einbetten lassen und eine Beschleunigung für bestimmte Klassen von Mustererkennungsalgorithmen versprechen.

15.3. Stärken der Architektur und offengelassene Fragen

Nachdem vergangenen Abschnitt eine Diskussion aus der Sicht der Anforderungen auf die konstruierte Architektur erfolgt ist, verfolgt dieser Abschnitt die entgegengesetzte Richtung. Aus der Sicht der konstruierten Architektur sollen Stärken und Limitationen der Architektur beleuchtet werden. Die Kombination dieser beiden Sichtweisen erlaubt es, möglichst viele Aspekte der Untersuchungen dieser Arbeit einzufangen. Diese beiden Ausführungen erlauben in der Folge eine entsprechende Gesamtevaluation der Forschungsfragen in Abschnitt 15.4.

Wie bei jeder Arbeit, welche ein ähnlich komplexes und weitreichendes Forschungsthema behandelt, wie es die vorliegende Arbeit tut, ist es unmöglich, alle weiteren dabei entstehenden Fragen umfassend beantworten zu können. Daher liefert der zweite Unterabschnitt eine Übersicht über naheliegende, aber nicht in der Arbeit diskutierte Fragestellungen. Die Hintergründe für den Verzicht auf die Einbeziehung der jeweiligen Frage werden entsprechend dargelegt.

15.3.1. Stärken

Dieser Abschnitt adressiert die Vorteile, welche auf der Basis und aus Sicht der Mushu-Architektur entstehen. Diese umfassen insbesondere die Eigenschaft von Mushu eine Referenzarchitektur für verteilte Referenznetzausführung zu sein, die Optimierung verteilter Aufrufe durch Mushu, die Abbildung des Deployments durch die Architektur sowie die Eignung für heterogene Anwendungsumgebungen.

Referenzarchitektur für verteilte Referenznetzausführung

Petrinetze im Allgemeinen und Referenznetze im Speziellen sind hervorragende Modellierungsmittel für die Abbildung von nebenläufigen Systemen. Verteilte Systeme weisen eine Reihe Eigenschaften auf, welche über die Eigenschaften rein lokaler Systeme hinausgehen. Eine Eigenschaft ist die inhärente Nebenläufigkeit verteilt ausgeführter Systemteile. Während andere Ansätze bemüht sind, das verteilte System einer künstlichen Taktung zu unterwerfen, verfolgen Petrinetze zumindest in der True-Concurrency-Semantik einen natürlicheren Ansatz.

Aus diesem Grund können Petrinetze konzeptuell auch problemlos für die Beschreibung verteilter Systeme eingesetzt werden. Eine Spezialität bei Referenznetzen ist ihre Eigenschaft, neben der Modellierung gleichzeitig auch eine ausführbare Implementation des Modells mitsamt Nebenläufigkeit mitzuliefern. Während das Modell selbst ein verteiltes System beschreiben kann, gestaltet sich die echte verteilte Simulation solcher Netze durch die Verschmelzung mit ausführbaren Elementen als weitaus schwieriger.

Referenznetze greifen auf synchrone Kanäle zurück, um Hierarchie, Nebenläufigkeit und insbesondere die Auflösung von Referenzen und Informationsaustausch zwischen Netzinstanzen zu realisieren. Durch ihre komplexe Natur sind sie das zentrale Konstrukt in Referenznetzen, jedoch dadurch auch das am schwersten zu verteilende. Erste Arbeiten beschäftigten sich mit der isolierten Ausführbarkeit von entfernten Referenznetzsimulationen (S. Bendoukha, 2017) oder der technischen Realisierbarkeit einer reduzierten Fassung verteilter synchrone Kanäle (M. Simon und Moldt, 2016).

Keine der bisher präsentierten Arbeiten hat jedoch die Verteilung der Simulation von Referenznetzen in einer strukturierten Fassung unternommen. In diesem Kontext ist die Mushu-Architektur die erste Referenzarchitektur, mit deren Hilfe verteilte, agentenbasierte Referenznetzsimulationen in einer strukturierten Fassung entworfen werden können. Dabei leistet sie zentrale und wegbereitende Beiträge, wie etwa die Verringerung der Häufigkeit verteilter Synchronisationen, die Adressierung des Deployments und Skalierbarkeit des Gesamtdeployments direkt aus der Architektur heraus sowie die Unterstützung potenziell heterogener Plattformen bzw. Simulatoren und Umgebungen. Diese drei Aspekte werden in den folgenden Abschnitten genauer diskutiert.

Verringerung der Häufigkeit verteilter Synchronisationen

Verteilte Synchronisation erfordert die Übermittlung von Informationen an entfernte Maschinen. Diese Art Informationsübermittlung ist immer und in jedem Fall langsamer als ein lokaler Aufruf. Dies ist insbesondere im Kontext der Berechnung synchrone Kanäle von Interesse, da diese Berechnungen ausgesprochen komplex ausfallen können.

Durch die Einführung der Agentenmetapher, wie sie aus Mulan bekannt ist und der Erweiterung auf die Modellierung von plattformübergreifendem Management durch die hier vorgestellte Mushu-Architektur, erhält die verteilte Synchronisation einen klar definierten Einsatzrahmen. Sobald Agenten mit anderen Agenten in Kommunikation treten, welche sich nicht auf der Plattform befinden, auf der sie selbst heimisch sind, ist verteilte Synchronisation erforderlich.

Dank der grundlegenden Unterstützung von Agentenumzügen können klar umrissene Anteile des Gesamtsystems – die Agenten – ihre heimischen Plattformen wechseln. Auf diese Weise müssen verteilte Synchronisation nur in wenigen Fällen ausgeführt werden und die Effizienz dynamischer lokaler Subsysteme innerhalb des globalen verteilten Systems kann effektiv ausgenutzt werden.

Robuste Kommunikation und Adressierung der Übertragungsmodalitäten

Eine der im Rahmen der Kommunikation zwischen Agenten seltener betrachteten Eigenschaften ist die Modalität der Übertragung selbst. Viele Ansätze, wie beispielsweise die der FIPA, adressieren zwar Abläufe des Kommunikationsprotokolls und Inhalte der Nachrichten, aber nicht die Frage, wie derartige Nachrichten übertragen werden.

Da die Mushu-Architektur von Grund auf die Annahme der Verteilung innerhalb des Gesamtsystems trifft, liegt es ebenfalls nahe, ein robustes Übertragungsverfahren vorzusehen. Robustheit betrifft in diesem Kontext die Wahrscheinlichkeit,

mit der Nachrichten verloren gehen können, der Zusammenbruch des Nachrichtensystems und auch die Möglichkeit einzelner Kommunikationsteilnehmer temporär nicht erreichbar zu sein.

Mushu sieht durch den Einsatz eines Ablaufmanagers und Nachrichtenbrokers für komplexe Kommunikation zwischen Agenten einen sowohl skalierbaren und entkoppelnden als auch robusten Ansatz vor. Eine weitere Stärke ist die Modellierung zweier verschiedener Qualitätsansprüche an die Übertragung und die Berücksichtigung innerhalb der Architektur. Somit kann entsprechend der benötigten Qualität gegebenenfalls auf Ansprüche verzichtet werden und somit eine niedrigere Komplexität der Übertragung erreicht werden.

Innerhalb der Realisierung kommt ein auf Petrinetze erweitertes Transaktionsmuster für eventual consistency zum Einsatz: Petrinetz-Sagas. Neben ihrem Einsatz im Rahmen der Mushu-Architektur sind Petrinetz-Sagas auch unabhängig davon und für das Deployment klassischer Microservice-Strukturen beschrieben und ausdefiniert.

Deployment ist Teil der Architektur

Eine weitere Stärke der Mushu-Architektur ergibt sich daraus, dass das Deployment und seine Umgebung ebenfalls durch die Architektur betrachtet werden. Mushu beschreibt Deployment und Implementation als eine integrierte Gesamtarchitektur und abstrahiert unter der Prämisse Container als Deploymentformat einzusetzen von der konkreten Ausprägung physikalischer Maschinen. Auf der Basis der Nutzung portabler Deploymentformate wird eine Spezifikation der benötigten Deploymentumgebung mittels der Konfiguration des Systems ermöglicht und darüber hinaus eine dynamische Manipulation des Deployments durch das System zur Laufzeit ermöglicht.

Dadurch ergeben sich die Möglichkeiten, Elemente des Deployments in gleichem Maße anzusprechen wie Softwarekomponenten. Mushu beschreibt damit indirekt auch eine Erweiterung der Agentenmetapher auf Deploymentlandschaften.

Szenarien mit heterogenen Plattformen

Die Mushu-Architektur unterstützt durch die Modellierung verschiedener dynamisch anpassbarer Plattformfunktionalitäten die Abbildung heterogener Systemlandschaften. Während andere Systeme diverse Vorbereitung benötigen, um heterogene Systemlandschaften in ein einheitliches Schema anzugleichen, wird die Heterogenität von Mushu inhärent unterstützt.

Während die Mushu-Architektur auch in homogenen Umgebung eingesetzt werden kann, gehen eine Reihe der Stärken in diesem Fall verloren. Für Fälle von homogenen Umgebungen existieren viele weitreichend untersuchte Lösung, welche auf diesen Anwendungsfall zugeschnitten sind. An dieser Stelle unterscheidet sich Mushu auch von vielen anderen Skalierungslösungen, da hierbei meist nur identische Kopien der Anwendung in Zusammenhang mit einem Load Balancer (dt.: Lastbalancierer) eingesetzt werden. Durch die integrierte Abbildung von Deploymentlandschaft und Anwendung selbst sowie der Realweltmetapher der Agenten, welche für sich neue Plattformen kreieren, können Skalierungsoperationen in weit größerer Detailtiefe und spezifischer in die konkrete Anwendung integriert werden.

15.3.2. Offene Fragen und Limitationen

Neben den vielen angesprochenen Stärken, existieren wie bei jedem Ansatz auch Limitationen, insbesondere bezogen auf die möglichen Einsatzbereiche von Mushu. Darüber hinaus wurden viele noch weitere mögliche Untersuchungen nicht mehr im Rahmen der Arbeit unternommen. Die entsprechenden Gründe hierzu sowie Ideen zur Umsetzung jenseits des Umfangs der Arbeit werden an dieser Stelle ebenfalls diskutiert.

Integration in MULAN-Umsetzungen und CAPA

Mulan wurde neben der abstrakten Architektur ebenfalls im Kontext von Referenznetzen als Implementation umgesetzt. Darauf aufbauend bietet Capa eine Plattform, welche sich an den Vorgaben der FIPA orientiert. Während Mushu auf der konzeptuellen Ebene eine unmittelbare Weiterentwicklung von Mulan darstellt, erfolgte die Implementation der Prototypen nicht direkt in den Mulan Umsetzungen, wie es unter Umständen auf den ersten Blick logisch erschiene.

Dies hat mehrere Gründe. Die hauptsächliche Entwicklungsarbeit bezüglich der Mushu-Komponenten adressiert die Ebenen oberhalb der Mulan-Architektur. Die Umsetzungen setzen direkt auf Renew und dem Referenznetzformalismus auf. Somit sind sie in dieser Form noch allgemeiner gehalten, als wenn sie spezifisch für Mulan umgesetzt worden wären. An einzelnen Stellen der Prototypen wurde bereits erläutert, dass diese auch außerhalb der Agentenmetapher als einzelne Systemkomponenten eingesetzt werden können.

Die Implementationsarbeit zu Mushu erweitert somit die Basis, auf der Mulan arbeitet. Da eine Integration konzeptuell keine weiteren Elemente einführen würde, verbleibt für eine Integration reine Handwerksarbeit auf Implementationsebene.

Die Capa-Plattform steht in gewissem Maße in Konkurrenz mit der Plattform, wie Mushu sie beschreibt. Dies wurde zu Beginn der Arbeit erläutert und liegt darin begründet, dass die Capa-Plattform nicht ausreichende Modellierungsmöglichkeiten im Kontext eines Plattformmanagements bietet. Eine Integration wäre hier denkbar, wurde aber für den Kontext der Arbeit nicht als ausreichend interessant befunden, um dafür auf andere grundlegende Konzeptrealisierungen wie den Nachrichtenversand zu verzichten.

FIPA-Konformität von Agentennachrichten

Die Umsetzung beispielsweise in Capa zeichnet sich unter anderem durch eine FIPA-Konformität der Nachrichten aus. Eine naheliegende Frage wäre daher, warum eine Kompatibilität zu FIPA im Rahmen der Modellierung von Kommunikation im Kontext der Mushu-Architektur nicht diskutiert wurde.

Die Antwort auf diese Frage lässt sich mit der Ebene der Betrachtung der Kommunikation geben. FIPA-Nachrichten beschreiben zwar Inhalt der Nachrichten und in gewissem Maße das Kommunikationsprotokoll, lassen jedoch die konkrete Übertragung insbesondere über Plattformgrenzen hinweg als »Message Transport System« abstrakt. Da mit den bestehenden Untersuchungen bezüglich Mulan bereits ausführliche Ergebnisse für diese Aspekte von Nachrichten vorliegen, bezieht sich die Untersuchung im Rahmen von Mushu auf die höher angelegte Ebene der Frage, wie Nachrichten übermittelt werden und nicht, welche Inhalte die Nachrichten aufweisen.

In diesem Sinne sind die Untersuchungen im Rahmen von Mushu orthogonal zu den Inhalten der Nachrichten und etwaige versandte Nachrichten können zusätzlich FIPA-konform sein, wenn die Agenten entsprechende Protokolle einsetzen.

Sicherheitsaspekte

Bei jedem Deployment in einer produktiven Umgebung müssen eine gewisse Menge an Sicherheitsaspekten beachtet werden. Da Mushu insbesondere auch das Deployment modelliert, sind für ein produktives Deployment auch derartige Fragen interessant.

Mit Blick auf die Forschungsfragen zeigt sich jedoch, dass die zentrale Beantwortung der Fragen auf die Modellierung und generelle Machbarkeit einer entsprechenden Architektur hinführt. Einige Sicherheitsaspekte in der Form von Vertrauensnetzwerken (engl.: *net of trust*) sind im Rahmen von Referenznetzen auch schon beispielsweise durch die Arbeit (Jürgensen, 2021) umgesetzt, welche auch sehr gute Anwendungen im Rahmen der Mushu-Architektur finden können. Aus

diesen Gründen liegen Sicherheitsbetrachtungen jenseits des Umfangs der Arbeit und verbleiben für weitere Untersuchungen.

Prototypen zu Simulationsfeed und weiteren Komponenten

Während der Beschreibung der Mushu Plattform wurde die Konstruktion eines plattformübergreifenden Simulationsfeeds diskutiert. Als konzeptueller Hintergrund wurde hierbei sowohl das holonische Agentenmodell aus der Betrachtungsseite der Agenten angeführt als auch die Beobachtbarkeit in Cloud-Native-Architekturen aus der Perspektive der Web- und Microservices. Während beide Ansätze wertvolle Beiträge in der Arbeit zur Diskussion der Forschungsfrage lieferten, steckt die Realisierung eines entsprechenden Simulationsfeeds nicht in den Kernfragen. Die Konstruktion ist für die Administration eines entsprechenden Systems jedoch hilfreich, sodass zumindest die theoretische Fundierung in der Arbeit diskutiert wurde. Die Implementationsarbeit wurde aber aufgrund der eben genannten Gründe für Untersuchungen jenseits der Arbeit belassen.

Plattformerweiterung vs. Plattformanpassung im Prototypen

Der vorgestellte Prototyp des »Cloud-Native Renew-Plugin« umfasst im aktuellen Zustand nur die Erweiterung von Plattformfunktionalitäten. Der Hintergrund liegt in der Java-basierten Implementation und der Eigenschaft, dass Java klassisch kein Entladen oder Anpassen von bereits geladenem Code vorsieht. Im Rahmen der bereits angesprochenen Arbeit (Janneck, 2021) wird die Weiterentwicklung von Renew auf Renew 4.0 beschrieben. Dabei kommt das Java Modulsystem zum Einsatz, welches eine genauere Trennung in der Entwicklung zwischen den einzelnen Komponenten ermöglicht und Plugins weiter entkoppelt. Als weitere Neuerung wird in Renew 4.0 ebenfalls das Java Modullayersystem eingesetzt, welches diese Trennung auch in die Laufzeit der Anwendung propagiert. Auf der Basis der Trennung zwischen Modullayern in Renew 4.0 wird es in Zukunft denkbar sein, Funktionalität auch wieder entladen zu können.

Aus diesem Grund ist exakt der Punkt der Plattformanpassung bereits Gegenstand laufender Untersuchungen und muss daher ruhigen Gewissens an dieser Stelle nicht weiter adressiert werden. Sobald dort eine entsprechende Implementation vorliegt, ist diese effektiv auch für das Cloud-Native Renew-Plugin verfügbar.

Einzelne Agenten auf mehreren Plattformen

Bei genauer Betrachtung der Ableitung der Mushu-Architektur aus der realen Welt kann auffallen, dass zu Anfang argumentiert wurde, dass Entitäten auf meh-

reren Plattformen gleichzeitig mit anderen Entitäten in Verbindung treten können. Bei der Umsetzung von sowohl MULAN als auch der Weiterentwicklung durch diese Arbeit, MUSHU, sind Agenten jedoch stets immer nur einer Plattform zugeordnet. Diese Problematik und weitere führten jedoch in MUSHU dazu, auch plattformübergreifende Kommunikation zu unterstützen. Agenten können unabhängig davon, ob sie auf der gleichen Plattform heimisch sind, miteinander Interaktionen ausführen. Um eine intensivere Zusammenarbeit zu ermöglichen, sollten sich die Agenten auf der identischen Plattform einfinden, um die Beschleunigung einer lokalen Ausführung ausnutzen zu können.

Bezogen auf die beschriebene Realweltsituation entspricht dies dem fokussierten Arbeiten an einer Hauptaufgabe. Es können zwar mehrere Tätigkeiten gleichzeitig erfolgen, jedoch erhält im Normalfall jeweils eine Tätigkeit die hauptsächliche Aufmerksamkeit. Auf dieser Basis ist die Kommunikation über Plattformgrenzen und die Kommunikation auf Plattformen in MUSHU umgesetzt.

Einsatz in klein umrissenen Problemdomänen

Der Einsatz von Referenznetzen, MUSHU, RENEW und das gesamte Ökosystem eines MUSHU Deployments erzeugt einen großen Overhead. Kleine Problemdomänen mit einem klar umrissenen Anforderungsprofil und insbesondere homogenen Anforderungen können daher durch den Overhead negativ beeinflusst werden. Für derartige Probleme sind spezialisierte Lösungen mit weit mehr Annahmen meist der performantere Weg. Um die Stärke wirklich ausspielen zu können, benötigt MUSHU eine umfangreiche und im Idealfall heterogene Umgebung.

Große Fallstudien

Im Rahmen der Arbeit wurden Konzept, Realisierungskonzepts und die prototypische Implementation von MUSHU beschrieben. Dabei wurde anschließend auf die explizite Darstellung eines großen Anwendungsbeispiels bzw. einer großen Fallstudie auf einer noch niedrigeren vierten Ebene verzichtet. Dies liegt darin begründet, dass dieses Beispiel entlang der Argumentation des vorangegangenen Punktes eine große Größe aufweisen müsste und daher entsprechend viel Zeit zur Modellierung in Anspruch nehmen würde. Um den Gesamtrahmen dieser Arbeit nicht weiter übermäßig auszudehnen, hätte eine derartige Konstruktion den Verzicht auf die Diskussion anderer Aspekte der Arbeit bedeutet. Da allerdings alle betrachteten Aspekte zentrale Bestandteile der Gesamtarchitektur insbesondere auf der Plattformmanagementebene von MUSHU sind, wäre ein Verzicht auf einzelne Bestandteile unglücklich.

Darüber hinaus kann analog zur Argumentationsstruktur der Arbeit (Duvigneau, 2009) argumentiert werden, dass die Anwendung der Mushu-Architektur auf den Referenznetzsimulator Renew, welcher bereits die Simulation nebenläufiger Systeme erlaubt, eine Fallstudie auf der Ebene des Realisierungskonzepts darstellt. Aus den genannten Gründen wurde daher von der Konstruktion einer umfangreichen Implementation innerhalb einer beispielhaften Anwendungsdomäne abgesehen.

Formale Beweise

Formale Beweise sind mit der Ausnahme der Binary Squaring Theorie nicht Teil dieser Arbeit. Die Hintergründe hierbei ergeben sich so, dass durch die vielen betrachteten Aspekte von Nebenläufigkeit, Autonomie, Plattformerschaffung und -manipulation, etc. ein umfassender Problemrahmen übergreifend adressiert wird. Dieser ist vonseiten des Umfangs zu komplex, als dass er sich gut formalisiert im Rahmen einer einzelnen Arbeit darstellen lassen würde. Darüber hinaus existieren keine Formalisierungen der Ausgangsarbeiten wie beispielsweise von Mulan. Für Mulan ist diese aufgrund der Komplexität der Modelle bisher nicht erfolgt. Dies wäre grundlegend kein Problem, müsste jedoch nachgezogen werden und hätte den Fokus dieser Arbeit deutlich verlagert. Durch den Umfang der technischen Realisierung zu Mushu, welche in dieser Arbeit vorgestellt wurde, ist eine Formalisierung der technischen Seite unrealistisch. Die Formalisierung auf der konzeptuellen Ebene zu den Anteilen zur komplexen Agentenkommunikation ist auf der Basis der Arbeiten zu Workflow-Netzen jedoch denkbar. Eine mögliche Basis für die Abbildung von Agentensystemen auf Workflows zur Verifikation findet sich auch in (Markwardt, 2013). Die zugehörigen Ideen spiegeln sich bei den Agentenmodellen in den Agent Interaction Protocols (AIPs) wider, welche für die Generierung von Protokollnetzen verwendet und dann verifiziert werden können (Cabac, 2010).

15.4. Diskussion der Forschungsfragen im Kontext der Ergebnisse

Als abschließenden Punkt in der Evaluation der Ergebnisse der Arbeit liegen nun alle Voraussetzungen vor, um die zu Beginn der Arbeit formulierten Forschungsfragen zu erörtern. Diese werden dabei jeweils in ihrem Wortlaut wiederholt und es folgt eine entsprechende Diskussion.

Forschungsfrage *»Wie können, entsprechend realweltlichen Interaktionen, nebenläufige, interagierende Agenten mit Einfluss auf die Plattformen, auf denen*

ihre Interaktion stattfindet, sowie auf deren Gesamtmenge als Modell bzw. Architektur eines verteilten Systems, beschrieben werden?«

Hierarchische Zusammenhänge können in direkter Weise durch den Referenznetzformalismus, auf welchem die Mushu-Architektur aufbaut, modelliert werden. Zu den Konzepten der objektorientierten Programmierung bieten Referenznetze zusätzlich inhärente Nebenläufigkeit, insbesondere mit der True-Concurrency-Semantik, sowie die Modellierung von Synchronisationen. Die Möglichkeit zum multilateralen Informationsaustausch während der Synchronisation ist dabei ebenfalls gegeben.

Durch die neu eingeführte Mushu-Architektur erfolgt im Kontext der Simulation nebenläufiger, interagierender Agenten eine integrierte Beschreibung von Anwendung und Deploymentumgebung. Auf der konzeptuellen Ebene werden diese technischen Unterscheidungen vereinigt beschrieben. Da eine Plattform in einer Simulation als natürliches Äquivalent eine physikalische Maschine eines verteilten Systems abbildet, wird durch diese Verbindung eine Brücke für die Interaktion von Agenten mit ihren Plattformen geschlagen.

Das durch Mushu eingeführte Plattformmanagement liegt entlang der Einheitentheorie oberhalb der Plattform, welche wiederum oberhalb der Agenten angesetzt ist. Wie auch bereits die Mulan-Architektur auf konzeptueller Ebene in ihrer Anwendung auf Realweltzusammenhänge verschoben werden konnte und damit Plattformen auch als Agenten agieren konnten, ist dies in Mushu weiterhin vorgesehen. Plattform und auch Plattformmanagement können daher als Agenten fungieren. Somit können Agenten über ihre Plattform Nachrichten an das übergeordnete Plattformmanagement senden und somit Einfluss auf die Plattformmenge nehmen. Durch Kommunikation mit der Plattform kann ebenfalls Einfluss auf die Plattform selbst genommen werden.

Während die Mushu-Architektur theoretisch auch auf ein lokales System angewendet werden könnte, wird ausführlich in der vorliegenden Arbeit beschrieben, wie sie für die Realisierung eines echt verteilten Systems eingesetzt werden kann. Die Forschungsfrage kann somit insoweit beantwortet werden, dass eine derartige Beschreibung eines verteilten Systems möglich ist und durch die Mushu-Architektur erfolgt.

Forschungsfrage *»Wie kann eine nebenläufige bilaterale Kommunikation zwischen Agenten gewährleistet werden, wenn ihre Plattformen zur Interaktion potentiell flüchtige Entitäten sind?«*

Agenten in der Mushu-Architektur sind innerhalb von Plattformen eingebettet, welche wiederum innerhalb von Plattformmanagements eingebettet sind. Kommunizieren Agenten lokal miteinander, so können die Eigenschaft eines lokalen Systems ausgenutzt werden und Kommunikationen erfolgen schnell und unkompliziert. Stellt eine gegebene Plattform ihren Dienst ein, so kann auch die lokale

Kommunikation zunächst abgeschlossen werden und die Agenten umgezogen werden, bevor die Plattform abgebaut wird.

Bei der Kommunikation zweier Agenten über entfernte Plattformen hingegen sieht die MUSHU-Architektur den Einsatz eines Nachrichtenbrokers und eines Ablaufmanagers vor. Der Nachrichtenbroker ist eine verteilte und replizierte Komponente, welche Nachrichten zwischen Plattformen und damit ultimativ auch zwischen Agenten auf verschiedenen Plattformen vorhält. Der Ablaufmanager sorgt für die korrekte Zustellung der einzelnen Nachrichten. Dabei kennen nur der Initiator und der Ablaufmanager die konkreten Teilnehmer der Kommunikation und dabei insbesondere auch nur ihre logische (und damit positionsunabhängige) Adressierung. Daraus ergibt sich eine Entkoppelung der Komponenten und der Abbau statischer Abhängigkeiten.

Durch die Verankerung des Ansatzes eventual consistency im System können während einer Kommunikation Plattformen unerreichbar sein und dennoch komplexe Kommunikationen fortgeführt werden, sobald die Plattform oder eine ihr äquivalente Plattform zur Verfügung stehen. Ablaufmanager können auf der Basis des für Petrinetze erweiterten Saga-Patterns als zueinander und auch inhärent nebenläufige Konstrukte realisiert werden.

Die Forschungsfrage kann daher dahingehend beantwortet werden, dass das Zusammenspiel von Nachrichtenbroker, Ablaufmanager und Petrinetz-Saga-basierte Abarbeitung für eine nebenläufige bilaterale Kommunikation zwischen Agenten eingesetzt werden kann, auch dann, wenn ihre Plattformen flüchtige Entitäten sind.

Forschungsfrage *»Ist es möglich, für einige Bildverarbeitungsalgorithmen grundlegende und beschleunigende Heuristiken zu definieren, deren Leistung zwischen reiner Umsetzbarkeit der Aufgabe und den Anforderungen einer Echtzeitanwendung liegt?«*

Auf der Ebene der Subprotokolle innerhalb der MUSHU-Architektur wurde ein derartiges Beispiel aus dem Bereich der Mustererkennung genannt. Während dies als Beispiel für ein Subprotokoll in der MUSHU-Architektur dient, sind die dort gemachten Erkenntnisse wesentlich allgemeiner. Betrachtet wurde die Unterkategorie von Mustererkennungsalgorithmen, welche in hochdimensionalen euklidischen Räumen darauf angewiesen sind, Distanzen zu berechnen. Als ein beispielhafter Vertreter wurde der Algorithmus SIFT vorgestellt.

In diesem Kontext wurde die Theorie von Binary Squares eingeführt und motiviert, dass damit die Darstellung des Suchraums für eine Schwellenwertberechnung derartiger Distanzen deutlich optimiert werden kann. Die Ergebnisse versprechen interessante Anwendungsmöglichkeiten und einfache Untersuchungen zeigen bereits deutliche Beschleunigungen.

Aus diesem Grund kann die Forschungsfrage dahingehend beantwortet werden, dass es durch Binary Squaring eine beschleunigende Heuristik zu definieren, ohne dabei Qualitätseinbußen einer Echtzeitanforderung einzugehen.

15.5. Zusammenfassung - Diskussion

Die Mushu-Architektur ist das zentrale Ergebnis der vorliegenden Arbeit. Sie erfüllt damit alle der an die zu konstruieren Architektur gestellten Anforderungen wie Skalierung von Plattformen durch Agenten abzubilden, Agenten die Möglichkeit einzuräumen, Plattformen zu manipulieren und eine Modellierung von Nachrichtenübertragung, welche die Unwägbarkeiten eines verteilten Systems berücksichtigt. In der Evaluation wurden die Stärken der Architektur wie beispielsweise die Bereitstellung einer Referenzarchitektur für verteilte Referenznetzausführung oder aber die Integration des Deployments in die Architektur und die Unterstützung von heterogenen Plattformen beleuchtet. Dagegen wurden aber auch offene Fragen und Limitationen, wie beispielsweise Sicherheitsaspekte und die Integration mit Mulan und Capa thematisiert. Abschließend wurden erneut die zu Beginn der Arbeit gestellten Forschungsfragen aufgegriffen und präsentiert, dass Mushu, die in Mushu konstruierte Kommunikationsform sowie die gezeigten Verbesserungen an einer Klasse von Bildverarbeitungsalgorithmen insgesamt alle Forschungsfragen in positiver Weise beantworten konnten.

Teil V.

Abschluss

16. Gesamtzusammenfassung

Dieses Kapitel gibt eine umfassende Zusammenfassung aller behandelten Themen und aller Ergebnisse dieser Arbeit. Die Gliederung orientiert sich hierbei an den Teilen der Arbeit.

16.1. Teil I - Ausgangspunkt

In der realen Welt ereignen sich täglich eine Vielzahl an Interaktionen zwischen verschiedensten Entitäten. Dabei interagieren diese zum Teil nebenläufig auf verschiedensten Plattformen sowohl für sich als auch miteinander und befolgen dabei bestimmte intrinsische Protokolle und Ziele. Die agentenorientierte Softwareentwicklung kann derartige Zusammenhänge abbilden. Solche Entitäten tragen hier den Namen »Agent«.

Je nach Granularität der Betrachtung können Protokolle auch selbst wiederum als Agenten modelliert werden, während der ursprüngliche Agent so selbst zur Plattform wird. Ein Beispiel hierzu wäre der Weltmarkt, auf dem Unternehmen konkurrieren; ein Unternehmen, welches kooperierende Abteilungen besitzt; die Abteilungen, welche Mitarbeiter besitzen, die nach bestimmten Grundsätzen handeln, welche sich wiederum durch eine Vielzahl biochemischer Prozesse ergeben. Je nach Granularität kann die Hierarchie »Plattform – Agent – Protokoll« an jeder Stelle dieses Beispiels angebracht und betrachtet werden.

Diese Zusammenhänge können durch die MULAN-Architektur beschrieben werden. Entitäten können jedoch auch Einfluss auf Plattformen nehmen, weitere Plattformen erzeugen, diese ändern und auch wieder zerstören bzw. abbauen/entfernen. Simulationen können dabei helfen, Abläufe vorauszuberechnen und Aussagen aus Modellen abzuleiten. Die Szenenanalyse kann dabei helfen, aus Realweltszenen Abbildungen auf die Wirklichkeit zu konstruieren, indem auf der Basis von Wissen und Beobachtungen Hypothesen aufgestellt werden.

Auf der Basis all dieser Umstände wurden die folgenden Forschungsfragen formuliert:

»Wie können, entsprechend realweltlichen Interaktionen, nebenläufige, interagierende Agenten mit Einfluss auf die Plattformen, auf denen ihre Interaktion statt-

findet, sowie auf deren Gesamtmenge als Modell bzw. Architektur eines verteilten Systems, beschrieben werden?«

»Wie kann eine nebenläufige bilaterale Kommunikation zwischen Agenten gewährleistet werden, wenn ihre Plattformen zur Interaktion potentiell flüchtige Entitäten sind?«

»Ist es möglich, für einige Bildverarbeitungsalgorithmen grundlegende und beschleunigende Heuristiken zu definieren, deren Leistung zwischen reiner Umsetzbarkeit der Aufgabe und den Anforderungen einer Echtzeitanwendung liegt?«

Für Abbildungen dieser Art existieren vielfältige Möglichkeiten der Modellierung. Dabei sind Petrinetze bzw. insbesondere Referenznetze erwähnenswert, da sie im Gegensatz zur UML eine formale Fundierung und eine operationale Semantik besitzen sowie interaktive Simulationen auf ihnen möglich sind. Referenznetze und die dazu existierenden Vorarbeiten überzeugen mit der direkten Vereinigung von Modell und ausführbarer Software. Die Strukturen »Plattform - Agent - Protokoll« können hervorragend auf Referenznetzbasis formuliert werden, wie die Mulan-Architektur zeigt.

Mit dem Simulator Renew existiert ein mächtiges Werkzeug, welches eine Simulation von Referenznetzen ermöglicht. Bisher waren Simulationen in Renew allerdings nur schwerlich verteilt ausführbar. Es existieren zwar Verteilungsansätze, jedoch waren diese nur mit viel manuellem Aufwand in größerem Umfang ausführbar.

In einem zweistufigen Prozess wurden andere Architekturen und Ansätze zunächst mit der Ausrichtung des Themas der Forschungsfragen verglichen. Die Themeneingrenzung der Arbeit umfasst dabei die Punkte der echten Nebenläufigkeit (im Gegensatz zur getakteten Parallelität), die Interaktion von Agenten, die Beschreibung als verteiltes System, die explizite Unterscheidung zwischen lokaler und globaler Verarbeitung und die mögliche Interaktion der Agenten mit Plattformen. Dabei wurden die Architekturen JADE, Elastic JADE, cloneMAP, DSE-JAMON, MMAS2L, LightJason, Jadex, Orestes, MARS und Mulan sowie die beiden unbenannten Architekturen durch je Hsieh und Pawlaszczyk betrachtet.

Darüber hinaus wurden fünf weitere generelle Ansätze und Architekturgedanken aufgegriffen und mit den Themen abgeglichen. Dabei handelte es sich konkret um die Themenfelder der sogenannten GALS (Globally asynchronous, locally synchronous)-Systeme, des Cloud-Fog-Computings, genereller Arbeiten zum Infrastrukturmanagement, Arbeiten zur verteilten diskreten Eventsimulation und abschließend der agentenbasierten Szenenanalyse.

Von den betrachteten Themen passten nur Mulan, cloneMAP und MMAS2L ausreichend zu den Themen der angestrebten Architektur. Alle anderen vorge-

stellten Lösungen unterstützten zu viele der genannten Aspekte nicht, sodass sie im Kontext der Forschungsfragen nicht von tiefergehendem Interesse sind.

Um eine weitere Eingrenzung und Entscheidung für eine Grundlage für die zu konstruierende Architektur schaffen zu können, wurde auf Basis einer Vision an das Gesamtsystem eine Reihe an Anforderungen formuliert. Diese unterteilten sich in interaktionsbasierte, strukturelle, dynamische und anwendungsdomänenbezogene Anforderungen. Beim Abgleich der zuvor genannten, untersuchten Architekturen und Lösungen mit den Anforderungen stellte sich heraus, dass keine der bestehenden Architekturen die Anforderungen zufriedenstellend erfüllen kann.

Die größte Übereinstimmung mit den Anforderungen existiert bei der Mulan-Architektur, sodass diese als konzeptioneller Ausgangspunkt für weitere Untersuchungen herangezogen wurde. Die Kernidee bei diesen Untersuchungen bestand darin, die Granularitätsverschiebung der Mulan-Architektur einzusetzen, um ein Plattformmanagement einzuführen, welches sich bezogen auf Plattformen verhält wie die Plattformen in Bezug auf Agenten.

Auf der Basis der konzeptionellen Vision wurde nach einem angemessenen Ansatz für prototypische Umsetzungen und Erprobungen gesucht und dieser in Form der Webservices identifiziert. Ein besonders wichtiges Kriterium ist dabei die Skalierbarkeit.

Ausgehend von den betrachteten Anforderungen, der Erfüllung dieser durch Mulan sowie der Erweiterungen von Mulan wurde ferner abgeleitet, dass auch eine geeignete Plattform konzipiert werden muss und die Abstraktion der Agentenkommunikation betrachtet werden sollte.

16.2. Teil II - Konzeption einer skalierenden Referenznetzsimulation

Zunächst erfolgte eine konzeptuelle Aufarbeitung der Agentenkommunikation. Das Hauptergebnis dabei war die Unterscheidung zwischen einfacher Kommunikation, bei der ohne Zustellungsgarantien und nur mit unilateralem Informationsaustausch gearbeitet wird, und der komplexen Kommunikation, bei der diese Einschränkung entfällt. Es wurde hergeleitet, dass komplexe Kommunikation auf Basis der Task-Transition entworfen werden kann, deren Konstruktion im verteilten Fall aus dem Zwei-Phasen-Commit-Protokoll hergeleitet wurde.

Aufgrund der Konsistenzorientiertheit des Zwei-Phasen-Commits ist das Protokoll selbst jedoch eher ungeeignet im beabsichtigten Anwendungsbereich. Der Einsatz eines Verfahrens mit eventual consistency wurde somit angestrebt. Auf

die Struktur der Kommunikation passte das Saga-Pattern mit der Ausnahme der fehlenden Nebenläufigkeit, welche in späteren Teilen der Arbeit folgt.

Zusammen mit den Anforderungen durch die Kommunikation konnte nun eine geeignete Plattform konzipiert werden, welche eine automatische und semiautomatische Skalierung vonseiten der Applikation unterstützt. Während das Konzept abstrakt belassen wird, umfasst der intendierte Rahmen der Plattform einzelne Anwendungen innerhalb der Umsetzung. Unter dieser Maßgabe wurde die Cloud-Native-Architektur als möglicher Ausgangspunkt und als mögliches Paradigma für die Konstruktion der Plattform ausgewählt.

Die Plattform soll zu Folgendem in der Lage sein: Sie soll auf unbekannten Umgebungen lauffähig sein, den Start und Stopp von Agenten unterstützen, Agentendefinitionen erweitern und eine Reihe plattformspezifische Funktionalitäten für die Agenten zur Verwendung bereitstellen können. Funktionalitäten beschreiben die konkrete Ausprägung einer jeden Plattform, um so eine heterogene Plattformlandschaft abbilden zu können. Die in diesem Absatz beschriebenen Aspekte wurden als Interaktionen modelliert und ein Petrinetz-Modell der gesamten Plattformarchitektur konstruiert.

Zusammen mit den Anforderungen aus den Ausführungen zur Kommunikation und mit den Anforderungen zur Plattform konnte das Konzept zum Plattformmanagement entworfen werden. Die Hauptaufgaben des Plattformmanagements umfassen die proaktive und reaktive Erzeugung und Zerstörung von Plattformen, die Aggregation von Zustandsinformationen der einzelnen Plattformen sowie die Bereitstellung eines Identifikationssystems, eines Kommunikationsmediums und eines Ablaufmanagers für komplexe Kommunikation. Für die Erschaffung neuer Plattformen ist darüber hinaus die Verwaltung einer Basis-Plattformdefinition nötig, an der auch Anpassungen erfolgen können, die sich auf alle (neu erzeugten) Plattformen gleichermaßen beziehen. Analog zur Konzeption der Plattform erfolgte eine Beschreibung der Interaktionen sowie die Darstellung in einem Referenznetzmodell.

Aus den Konzepten zu Agentenkommunikation, Plattform, Plattformmanagement und den Ausgangskomponenten der Mulan-Architektur konnte nun die Mushu-Architektur konstruiert werden, welche in ihrer Gesamtheit eines der zentralen Ergebnisse dieser Arbeit darstellt.

Es wurde begründet, dass sich die Mushu-Architektur nicht als Spezialfall der Mulan-Architektur ausdrücken lässt. Insbesondere können Plattformmanagements und Plattformen nicht einfach ineinander überführt werden. Dies liegt im Wesentlichen daran, dass Plattformmanagements als föderierte Einheiten konzipiert sind, wohingegen Plattformen bestimmte Ausprägungen aufweisen, jedoch ansonsten einen monolithischen Charakter besitzen. Abschließend wurde Mushu im Kontext der Organ-Architektur beschrieben und motiviert, dass Mushu als

Referenzarchitektur für Agentensimulationen eingesetzt werden kann, bei denen Agenten Einfluss auf die physikalische Ausdehnung des Systems besitzen.

16.3. Teil III - Realisierungskonzepte zur Mushu-Architektur

Im zweiten Hauptteil der Arbeit wurde eine mögliche Realisierung der Mushu-Architektur diskutiert. Dazu sollte eine mögliche Umsetzung im Kontext des Simulators Renew und auf der Basis des Distribute Plugins erfolgen. Um die Umsetzung möglichst allgemein zu halten, bestanden neben diesen beiden Einschränkungen keine weiteren konkreten Vorgaben zur Technik. Stattdessen wurden die eingesetzten Techniken auf konzeptueller Ebene diskutiert. Zunächst wurde für die einzelnen Bestandteile des Konzeptes von Mushu erörtert, inwiefern bereits bestehende Lösungen im Kontext von Renew diese abdecken.

Dabei ergab sich die Notwendigkeit für die Realisierung von Beobachtbarkeit und Operabilität sowie Abhärtung auf der Anwendungsebene. Für die (komplexe) Kommunikation existierten mit dem Distribute Plugin und einer Untersuchung von ActiveMQ Vorarbeiten, welche als Grundlage dienen konnten. Eine Realisierung des Plattformmanagements hingegen musste von Grund auf konstruiert werden.

In der Betrachtung der Realisierung des Plattformmanagements nahm die Plattformskalierung eine zentrale Position ein. Es wurden diverse mögliche Lösungen der Umsetzung diskutiert. Schlussendlich ergab sich die Realisierung eines Management-Systems in Anbindung an einen Clustermanager als am zielführendsten. Das Management-System arbeitet nach dem Reconciler-Pattern und ist nach der Maßgabe eines Webservices konstruiert. Netzinstanzen in Renew können über spezielle Funktionen durch ein Anbindungsplugin mit dem System interagieren. Das Deployment der überwiegenden Anzahl der Komponenten sollte auf der Basis von Containerisierung erfolgen, um so eine weitere Unabhängigkeit von der zugrunde liegenden Plattform sowie die Portabilität und die dynamische Aktualisierung der Plattformdefinition zu ermöglichen.

Bei der Umsetzung der Plattform wurde aufgezeigt, dass sich Plattformfunktionalitäten durch den Einsatz von Renew-Plugins realisieren lassen. Entsprechend dazu werden Agentendefinitionen durch Netze abgebildet. Sowohl das dynamische Nachladen von Funktionalitäten als auch das reaktive Laden von Agentendefinitionen musste umgesetzt werden und sollte analog zur Realisierung des Plattformmanagements als Webschnittstelle erfolgen. Mit derartigen Schnittstellen war eine Integration in die Skalierungsumgebung durch die Realisierung des Plattformmanagements wesentlich leichter.

Es wurde weiter erörtert, dass der Umzug von Agenten einer Kombination von Nachrichtenversand und dem reaktiven Hochfahren von Agenten entspricht. Auch die Umsetzung eines Statusmonitors als eigenständiges Plugin mit optionaler Teilnahme durch einzelne Bestandteile der Plattform wurde vorgestellt. Als spezielle Umsetzung von Beobachtbarkeit wurde ebenfalls die Realisierung eines Simulationsfeeds diskutiert. Diese erfordert den Einsatz von Checkpoints unter True-Concurrency-Semantik, für welche ebenfalls eine mögliche Lösung vorgeschlagen wurde.

Für die Umsetzung der komplexen Kommunikation zwischen Agenten wurde die Darstellungsmöglichkeit von Sagas durch Petrinetze erörtert. Zusammengefasst ergab sich dabei, dass Petrinetz-Sagas als P/T-Netze definiert werden sollten, welche für die Ausführung sodann in Referenznetze überführt werden, um die Eigenschaften beider Formalismen optimal auszunutzen. Bei der Ausführung wird die Existenz eines Orchestrators angenommen, welcher dem Mushu-Ablaufmanager entspricht. Für die komplexe Kommunikation wurde als Gesamtkonstrukt Resilient Distribute entworfen, welches den Ablaufmanager repliziert einsetzt und eine robuste Neuimplementation des Distribute Plugins anstrebt. Dabei sollen die Kernalgorithmen des Distribute Plugins erhalten bleiben und der Feuervorgang eines verteilten synchronen Kanals auf der Basis von Petrinetz-Sagas erfolgen. Die Umsetzung einfacher Kommunikation auf der Basis von Resilient Distribute wurde ebenfalls präsentiert.

Mit allen zentralen Komponenten konnte so nun eine Darstellung einer möglichen Realisierung von Mushu im Kontext von Renew erfolgen. Auf dieser Basis konnte der Prozess der Szenenanalyse diskutiert werden. Dabei wurde exemplarisch als ein Teilschritt einer Analyse die Berechnung eines Mustererkennungsalgorithmus angenommen. Innerhalb der Mushu-Architektur wird dieser auf der Ebene der Subprotokolle verortet.

Für die Beantwortung der dritten Forschungsfrage wurde dann in diesem Kontext die Theorie des Binary Squaring eingeführt. Binary Squares können heuristisch eingesetzt werden, um effizient Schwellenwerte in hochdimensionalen euklidischen Räumen abzugleichen oder in allen Anwendungsfällen, bei denen eine große Menge an Quadraten aufsummiert und gegen einen Schwellenwert abgeglichen wird. Diese Anforderung tritt in einigen Mustererkennungsalgorithmen auf. Eine mögliche algorithmische Umsetzung der Binary Squaring-Theorie wurde ebenfalls diskutiert.

16.4. Teil IV - Prototypen und Evaluation

Mit dem Abschluss des Realisierungskonzepts konnten nunmehr konkrete Implementationen erfolgen. Dabei wurden verschiedene Prototypen im Rahmen der Arbeit vorgestellt. Als wesentliche Ergebnisse sind dabei zu nennen: Renew-Kube, welches die Skalierungsaspekte betrifft, das Cloud-Native Renew-Plugin, welches einen Großteil der Plattformaspekte implementiert, die Implementation von Petrinetz-Sagas auf der Basis vom Eventuate-Framework und Apache Kafka, die Implementation einfacher Kommunikation im Kontext von Resilient Distribute sowie eine beispielhafte Implementation von Binary Squares in der Bibliothek OpenSIFT. Darüber hinaus erfolgten einige für die Inhalte dieser Arbeit vorbereitende Implementationen sowie einige weitere Implementationen, welche auf den genannten Prototypen aufsetzen.

Abschließend widmete sich die Arbeit im Rahmen der Evaluation den Anforderungen, wie sie innerhalb der Anforderungsanalyse dargestellt wurden. Es ergab sich, dass die Mushu-Architektur alle im Rahmen der Anforderungsanalyse gestellten Anforderungen erfüllen kann. Dies betrifft insbesondere die Möglichkeit für Agenten, selbst Plattformen zu schaffen und diese auch zu manipulieren, während das Gesamtsystem als verteiltes System angenommen wird.

Mushu bietet dadurch eine Referenzarchitektur für die verteilte Referenznetzausführung und kombiniert sowohl die Deploymentebene als auch die Applikationsebene in einer integrierten Architektur. Darüber hinaus wurden nicht von dieser Arbeit behandelte Aspekte diskutiert.

17. Fazit und Ausblick

Dieses Kapitel zieht ein Fazit aus den Untersuchungen dieser Arbeit und zeigt mögliche Anknüpfungspunkte für folgende Arbeiten auf. Der betrachtete Gegenstand dieser Arbeit umfasst miteinander interagierende Entitäten wie beispielsweise Menschen, welche in der Lage sind, für die Interaktion miteinander dynamisch Räume bzw. Plattformen zu eröffnen, anzupassen und schließen zu können. Insbesondere wurde dabei Augenmerk auf die Simulation derartiger Systeme gelegt, bei der analog zur Realwelt die echte Nebenläufigkeit und umfangreiche Größe solcher Interaktionen durch viele Entitäten Beachtung findet.

Die agentenorientierte Softwareentwicklung sowie der Anwendungskontext der Szenenanalyse lieferte hierbei eine hervorragende Ausgangslage, sodass die folgenden zentralen Forschungsfragen formuliert und im Rahmen der Arbeit durch die Mushu-Architektur beantwortet werden konnten:

»Wie können, entsprechend realweltlichen Interaktionen, nebenläufige, interagierende Agenten mit Einfluss auf die Plattformen, auf denen ihre Interaktion stattfindet, sowie auf deren Gesamtmenge als Modell bzw. Architektur eines verteilten Systems, beschrieben werden?«

»Wie kann eine nebenläufige bilaterale Kommunikation zwischen Agenten gewährleistet werden, wenn ihre Plattformen zur Interaktion potentiell flüchtige Entitäten sind?«

Ferner wurde die folgende anwendungsbezogene Frage formuliert, deren Beantwortung sich in eine mögliche Realisierung der Mushu-Architektur einbettet:

»Ist es möglich, für einige Bildverarbeitungsalgorithmen grundlegende und beschleunigende Heuristiken zu definieren, deren Leistung zwischen reiner Umsetzbarkeit der Aufgabe und den Anforderungen einer Echtzeitanwendung liegt?«

Ausgehend von den Forschungsfragen konzentrierte sich die Untersuchung auf den Querschnitt einer Reihe von Forschungsfeldern: Den ersten Rahmen umfasste die *agentenorientierte Softwareentwicklung*, da Agenten die Annahme der grundlegenden Entitäten ideal abbilden. Dabei sollte insbesondere die *Interaktion dieser*

Agenten Berücksichtigung finden. Des Weiteren sollte ein *modellbasierter Ansatz* verfolgt werden, bei dem zunächst ein Modell des Systems entworfen wird anstatt einer direkten Umsetzung. Analog zur realen Welt sollte eine etwaige Taktung der Agenten keine Rolle spielen, weswegen eine explizite Modellierung von *echter Nebenläufigkeit* erforderlich ist. Aufgrund der potenziellen Größe der abgebildeten Situationen sollte die Untersuchung unter dem Vorzeichen der Konzeption eines *verteilten Systems* ablaufen. Dabei sollte insbesondere die *Unterscheidung zwischen lokalen und verteilten Abläufen* im System abgebildet sein. Darüber hinaus sollte eine Art von *Plattformverwaltung* adressiert werden, beispielsweise im Kontext automatisierter Skalierung und ähnlichen Forschungsfeldern.

Zur Beantwortung der zuvor genannten Forschungsfragen wurde in der vorliegenden Arbeit als zentrales Ergebnis die Mushu-Architektur beschrieben. Mushu umfasst dabei in seiner Konzeption alle Elemente von der einzelnen Softwarekomponente bis hin zur vollständigen Deploymentlandschaft und integriert sie in ein gemeinsames System. Diverse umfangreiche, implementierte Prototypen belegen die Realisierbarkeit des beschriebenen Systems. Wie bereits im Rahmen der Evaluation in Abschnitt 15.4 dargelegt, konnten alle Forschungsfragen in positiver Art und Weise beantwortet werden.

Zusammenfassend wurden die folgenden Ergebnisse und Nova durch den Autor dieser Arbeit beigetragen:

- **Eine integrierte Architektur für Agentensysteme, bei denen Agenten dynamischen Einfluss auf die Ausdehnung der Simulation und auf die Plattformen, auf denen sie interagieren, nehmen können unter Erhaltung der expliziten Modellierung von Nebenläufigkeit.** Die Architektur erhält die explizite Modellierung von Nebenläufigkeit durch den durchgehenden Einsatz von Petrinetzen. Da Plattformen eigentlich normalen Agenten übergeordnet sind, besitzen diese wiederum keinen direkten Zugriff auf die Plattformen selbst. Die Architektur löst diese Problematik dadurch, dass sowohl auf der Ebene der Plattform als auch auf der Ebene des Plattformmanagements grundlegende Bestandteile von Agenten integriert sind. Die Leitmetapher »Alles ist ein Agent« (Rölke, 2004) wird aufgegriffen, wie sie auch beim Paose-Ansatz vorliegt. Plattform und Plattformmanagement können somit auch als Agenten gesehen werden. Die Problematik wird somit auf eine Mitteilung zwischen Agenten reduziert und Agenten können indirekt über das Plattformmanagement auf die Plattform Einfluss nehmen. Für die agentenseitige Skalierung von Plattformen wurde ein umfangreiches Realisierungskonzept vorgestellt, welches ebenfalls als Prototyp implementiert wurde.
- **Eine Modellierung der Übertragungsmodalitäten der Agentenkommunikation.** Im Kontext der Beschreibung eines gesamten verteilten Systems kann und sollte hierbei auch dieser Aspekt Beachtung finden. Dabei

sind explizit nicht die konkreten Inhalte der Kommunikation von Interesse, da in diesem Bereich bereits umfangreiche und ausführliche Ergebnisse wie beispielsweise die Spezifikationen der FIPA und deren Implementation im Mulan-Kontext durch Capa vorliegen, sondern der Weg der Übertragung. Die Arbeit führt dabei die explizite Unterscheidung von einfacher und komplexer Kommunikation ein, wobei einfache Kommunikation die Einschränkung auf unilateralen Informationsaustausch und den Verzicht auf Zustellungsgarantien beinhaltet, während komplexe Kommunikation dieser Einschränkung nicht unterliegt. Im Rahmen des Realisierungskonzepts wurden verschiedene Möglichkeiten der Umsetzung dieser beiden Kommunikationsformen beschrieben.

- **Die explizite Adressierung, Handhabung und Integration von Heterogenität im verteilten System durch die Architektur.** Heterogenität wird in vielen Fällen zunächst mit einer aus der Realität stammenden Unwägbarkeit interpretiert, mit der ein System umgehen können muss. Es bestehen jedoch auch Systeme, bei denen Heterogenität gewollt oder sogar notwendig ist. Das kann beispielsweise die Integration spezialisierter Hardware wie (Bild)sensoren betreffen. Der Beitrag durch die Architektur liegt insbesondere darin, dass Heterogenität dynamisch und im laufenden Betrieb abgebildet werden kann. Zu diesem Zweck kann eine generelle einheitliche Plattformdefinition dynamisch durch Plattformfunktionalitäten angepasst werden. Diese Plattformfunktionalitäten können beispielsweise durch Agenten(nachrichten) nachgereicht werden. Auch die Gesamtheit der ansässigen Agenten inklusive ihrer Prokolle und Wissensbasen erzeugt die Möglichkeit für eine individuelle Ausprägung jeder Plattform. Auf diese Weise verfolgt die Architektur explizit nicht den Weg einer abstrahierenden Vorverarbeitung, sondern integriert Heterogenität als einen zentralen Stützpfeiler ihres Aufbaus. Im Kontext von Renew ist es durch Plugins möglich die Software auf konzeptueller Basis dynamisch anzupassen. Laufende Arbeiten erweitern die Plugins auf Module nach dem Java Platform Module System (JPMS) und unterstützen die aktuellsten Java-Versionen. In diesem Kontext wurden vom Autor dieser Arbeit und Teilnehmern des Lehrprojektes am Arbeitsbereich ART verschiedene Implementationen umgesetzt.
- **Die Konzepte zur Realisierung eines petrinetzgestützten verteilten Persistenzalgorithmus für *eventual consistency*, welcher auch für generelle Microservice Deployments eingesetzt werden kann.** Im Rahmen des Realisierungskonzepts für komplexe Agentenkommunikation wurde die Grundidee eines petrinetzgestützten Saga-Patterns eingeführt. In der Arbeit wurde der Einsatz der Petrinetz-Sagas als konzeptuelle Task-Transition in Aufbau auf die Workflow-bezogenen Arbeiten von (Reese, 2009) und (Wagner, 2018) beschrieben. Das generelle Konzept auch jedoch auch allgemeiner als im Kontext der komplexen Agentenkommunikation

einsetzbar. Das Saga-Pattern kann in seiner sequenziellen Form bereits in Microservice Deployments zum Einsatz kommen, um langlaufende, verteilte Persistenzoperationen abzubilden. Der Einsatz von Petrinetzen als Basis der Darstellung einer Saga führt als Novum die Modellierung expliziter echter Nebenläufigkeit in das Konstrukt ein. Neben dem Realisierungskonzept, aus welchem diese Neuerung entstammt, wurde ebenfalls eine vollständige Implementation als Prototyp auf der Basis des Saga-Frameworks Eventuate vorgestellt.

- **Eine Referenzarchitektur für die Modellierung und Implementation verteilter Referenznetzsimulationen.** Eine solche Referenzarchitektur ergibt sich durch eine umgekehrte, technische Sicht, bei der die Implementation einer verteilten Referenznetzsimulation angestrebt wird. Speziell bei diesem Punkt sei auf der Basis der technischen Ausgangssicht unabhängig vom konkreten Einsatzfall bereits festgelegt, dass eine Implementation mit einer verteilten Referenznetzsimulation erfolgen soll. Unter dieser Prämisse existierte bisher kein umfassendes Konzept, um eine derartige Implementation gezielt zu realisieren. Diese Lücke wird durch die Mushu-Architektur geschlossen, da diese als entsprechende Basis dienen kann. Die Ausführung verteilter synchroner Kanäle ist sehr aufwendig und sollte daher nicht über ein absolut nötiges Minimum hinausgehen. Mushu setzt dies durch die Trennung nach Plattformen und Plattformmanagements um, sodass Agenten nur über Plattformgrenzen hinweg verteilte Synchrone Kanäle einsetzen müssen. Durch Mushu besteht jetzt die Möglichkeit für eine Kommunikation eine eigene Plattform zu erzeugen und diese zu nutzen. Insbesondere bei gehäuft auftretenden Kommunikationen (Synchronisationen) zwischen zwei oder mehr festen Kommunikationspartnern liegt es nahe, die Anteile der Anwendung (die Agenten) gemeinsam auf eine (ggf. eigene) Plattform auszulagern. Mit der Modellierung von Agenten, der Plattformerschaffung und des Agentenumzugs liefert Mushu hierzu die konzeptionelle und technische Basis.

- **Ein Beitrag zur Erweiterung des Paose-Ansatzes** Mushu unterstützt durch die Fähigkeit von Agenten, spontan Plattformen erzeugen und ändern zu können, den Entwicklungsprozess eines Multiagentensystems. Als wesentlichen Beitrag bietet die Mushu-Architektur auf der technischen Ebene eine Ausführungsumgebung für Anwendungen nach dem Paose-Ansatz. Sie gliedert sich damit in den Paose -Ansatz ein, welcher aus Technik, Methode, Werkzeug, Ressourcen, Kontext, Paradigmen, Ausführungsumgebungen und weiteren Bestandteilen zusammengesetzt ist. Eine weitere Folgerung aus dem Paose -Ansatz ist die Übertragung des Paradigmas auf das Entwicklerteam. Die Fähigkeit von Entwicklern, sich in Meetings zu treffen, im Pair- und Crowd-Programming zu arbeiten und sich austauschen zu können, kann durch die Interkation mit Plattformen ausgedrückt werden.

Insbesondere in der verteilten Entwicklung kann das Konzept von Mushu überzeugen, da es eine architekturelle Fundierung der Möglichkeiten der Entwickler zum Aufbauen und Gestalten der Interaktionsmöglichkeiten auf virtueller Basis bietet.

- **Ein grundlegendes heuristisches Beschleunigungsverfahren für eine Klasse an (Mustererkennungs)algorithmen.** Als eine mögliche Abbildung der realen Welt auf ein entsprechendes Multiagentensystem wurde der Kontext der Szenenanalyse vorgestellt. Der Ablauf hierbei umfasst viele verschiedene Schritte, von denen einige beispielsweise auf der Basis von Mustererkennung erfolgen können. Um die Berechnung entsprechender Abbildungen generell zu beschleunigen, wurde als Novum die Theorie von »Binary Squares« und eine darauf basierende Heuristik vorgestellt, welche auf alle Algorithmen angewendet werden kann, welche den Abgleich von hochdimensionalen euklidischen Distanzen gegen Schwellenwerte beinhalten. Durch eine geschickte alternative Darstellungsweise des Problems können somit Beschleunigungen erzielt werden. Eine Umsetzung erfolgte prototypisch im Kontext der OpenSIFT Bibliothek. Bei der Implementation von Anwendungen auf der Basis von Mushu kann bei den genannten Algorithmen auf die Ergebnisse im Kontext von Binary Squaring dieser Arbeit zurückgegriffen werden.

Aus den verschiedenen Untersuchungen können diverse mögliche Anknüpfungspunkte abgeleitet werden, welche nicht zum zentralen Thema dieser Arbeit zählten und deswegen nicht tiefgehender betrachtet wurden, um den Untersuchungsrahmen klar umrissen zu belassen. Einige dieser Möglichkeiten zusammen mit entsprechenden Ideen der Umsetzung sollen an dieser Stelle abschließend aufgeführt werden:

- *Abbau und Update von Plattformfunktionalität.* Mit dem Abbau von einzelnen Plattformfunktionalitäten könnte eine dynamische Aktualisierung von Plattformfunktionalitäten implementiert werden. Zentrale Vorarbeiten dabei umfassen die Modularisierung von Renew, durch welche auf der Ebene der Modullayer eben diese Module, welche von Renew 4.0 für jedes Plugin vorgesehen sind, wieder entladen werden könnten. Auf diese Weise ist es nicht mehr notwendig, eine Plattform neu zu konstruieren und die gewünschten (neuen) Funktionalitäten nachzuliefern, sondern es ist möglich diese dynamisch im laufenden Betrieb zu aktualisieren.

- *Nebenläufige Sagas mit mehreren Pivotelementen.* Sagas unterteilen sich in drei distinkte Phasen, von denen die mittlere ein einzelnes Pivotelement umfasst, welches über Erfolg oder Misserfolg der Saga maßgeblich entscheidet. Durch Einführung der Nebenläufigkeit in Sagas und insbesondere auf Basis der Referenznetze wäre es denkbar, eine übergeordnete Pivottransition in mehrere untergeordnete nebenläufige Prozesse aufzuspalten. Auf diese

Weise könnten mehrere Pivotprozesse im System existieren, deren Erfolge in konjunktiver Verbindung über den Erfolg der virtuellen übergeordneten Pivottransition entscheiden. Mit Referenznetzen könnte dieses Verhalten als untergeordnete Netzinstanz und mit einem synchronen Kanal an der Pivottransition realisiert werden.

- *Verifikation von Petrinetz-Sagas.* Petrinetz-Sagas liefern einen formalisierbaren Unterbau für die Beschreibung nebenläufiger Sagas auf der Basis von Workflow-Netzen. Auf der Basis der Petrinetzstruktur könnte beispielsweise Verifikationsalgorithmen die Verifikation einzelner Eigenschaften durchgeführt werden. Für kritische Infrastruktur und kritische Kommunikation zwischen einzelnen Services kann dies eine interessante Untersuchung darstellen. Für erfolgreiche Läufe kann dies direkt auf den P/T-Netzen der Saga durchgeführt werden (da in diesen die Kompensation nicht modelliert sein muss), während die Verifikation für erfolgreiche und nicht erfolgreiche Läufe auf der Basis der (generierten) Referenznetze vollzogen werden muss. Nur die generierten Referenznetze enthalten den wirklichen Ablauf der Petrinetz-Saga mit allen kompensierenden Aktionen. Durch die Mächtigkeit des Referenznetzformalismus sollte für ein erfolgreiches Verifikationsvorhaben eine Einschränkung auf ein konkretes oder einige konkrete Modelle erfolgen.
- *IT-Sicherheit und Signaturen.* Die Integration von digitalen Signaturen in Plugins und Netze kann eine zusätzliche Sicherheit beim dynamischen Nachladen dieser Einheiten bieten. Ein möglicher Anknüpfungspunkt hierbei ist die Integration eines Vertrauensnetzwerkes in die technische Realisierung von Mushu wie es beispielsweise in (Jürgensen, 2021) beschrieben wird. Auf dieser Basis können sowohl Plattformen und Agenten als auch Modellierer und Anwender Vertrauen gegenüber anderen Agenten bzw. Funktionalitäten ausdrücken. Die Frage nach der Vertrauenswürdigkeit der Komponenten könnte so auch innerhalb des Systems von Agenten modelliert werden und wäre nicht mehr ausschließlich eine Arbeit, welche immer den Betreibern des Simulationssystems zufällt.
- *Authentifikation und Autorisation.* Im Rahmen der im letzten Punkt beschriebenen Infrastruktur wäre auch eine Erweiterung auf ein vollwertiges Authentifikations- und Autorisationssystem für Agenten denkbar. Plattformen könnten auf dieser Basis Agenten beispielsweise den Umzug verweigern, wenn diese nicht in der Lage sind, eine entsprechende Authentifizierung vorzuweisen. Analog dazu könnten bestimmte Plattformfunktionalitäten eine höhere Autorisationsstufe der Agenten erfordern. Diese Überlegungen sind problemlos ebenfalls auf das Plattformmanagement übertragbar.
- *Zustandslose Plattformen.* Wie bereits im Laufe der Arbeit angeregt, ist die Konstruktion zustandsloser Plattformen für eine Skalierbarkeit ideal. Sind

jedoch die Plattformen zustandslos, so muss der Zustand ihrer beheimateten Agenten an anderer Stelle persistiert werden. Beschriebene Ideen sind hierbei der Einsatz rein seiteneffektfreier funktionaler Abläufe und/oder der Einsatz externer Datenbanken.

- *Graphdatenbanken und ein lokal/verteilter Ansatz auf Datenbankebene.* Im Rahmen der Prototypen wurde eine verteilte Netzdatenbank als mögliche Lösung vorgestellt, jedoch wurde der Ansatz wegen antizipierter Performanceprobleme nicht weiter verfolgt. Eine mögliche Lösung für die Performanceproblematik könnte sich dadurch ergeben, dass der Ansatz, lokale und verteilte Operationen entlang der Mushu-Architektur dediziert zu trennen, auf die Ebene der unterliegenden Datenbanken übertragen wird. Eine entsprechende Lösung könnte gut mit zustandslosen Plattformen synergieren, die bei dieser Betrachtungsweise auf die reine Bereitstellung von Funktionalität und Simulationsalgorithmen reduziert werden würden. Der Einsatz von Graphdatenbanken wie beispielsweise *neo4j*[1] ist hierbei vielversprechend, zumal diese auf natürliche Weise die Struktur von Petrinetzen, welche insbesondere Graphen sind, abbilden.

- *Call-by-Name.* In ihrer aktuellen Form setzen Referenznetze in ihrer Implementation in Renew eine Referenzsemantik (»Call-by-Reference«) ein. Diese erlaubt ihnen äußerst komplexe Objekte abzubilden, die jedoch nicht zwangsläufig serialisierbar sein müssen. Für eine Übertragung im verteilten Fall ist häufig jedoch eine Serialisierung unabdingbar, sodass aktuelle Lösungen zur Verteilung dem »Call-by-Value« Ansatz entsprechen. Eine alternative Lösung könnte es sein, die Referenzierung direkter lokaler Adressen zugunsten einer (global eindeutigen) Benennung der Daten einzustellen. Auf diese Weise könnten auch systemübergreifende komplexe Objekte bestehen und einzelne, lokal nicht vorhandene Informationen könnten nachgeladen werden, ohne dabei rekursiv die gesamte Objektstruktur übertragen zu müssen. Durch diese Umstrukturierung wird ein Systemverhalten erzeugt, welches sich durch »Call-by-Name« beschreiben lässt. Bei einer Realisierung könnten die Algorithmen zur Benennung in den bereits bei der Realisierung von Mushu bestehenden Identifikationsservice integriert werden, da dieser diesen Ansatz auf den Ebenen der Simulation, Plattform und Netzinstanz bereits umsetzt.

- *Verteilung von Referenznetzsimulationen komplettieren.* In der Verteilung von Referenznetzen umfasst die beste verfügbare Lösung die Algorithmen des Distribute Plugins. Die Unifikation ist hierbei jedoch nicht vollständig umgesetzt. Dies liegt insbesondere daran, dass die Berechnung im verteilten Fall mit einer erheblichen Nachrichtenkomplexität einhergehen würde und nicht mehr effizient im allgemeinen Fall durchzuführen wäre. Ferner

[1] https://neo4j.com/ - Zuletzt abgerufen am 06.11.2021

kann nicht jedes Objekt (leicht) serialisiert und übertragen werden. Auch auf der Basis des Call-by-Name-Ansatzes müssen weiterhin verteilte Objekte koordiniert werden. Eine vollwertige Implementation könnte auf eine Weise erfolgen, dass die gesamte Berechnung des synchronen Kanals beim Initiator der Kommunikation durchgeführt wird und dieser jeweils benötigte Objekte und Daten auf Basis der zugehörigen Namen von den entfernten Knoten anfordert. Dadurch wäre die Berechnung erneut lokal und auch Unifikationen könnten effizient erfolgen. Jedoch würde dabei mit Kopien der Objekte gearbeitet werden, wodurch Inkonsistenzen entstehen können. Eine mögliche Lösung wäre aus der Sicht des Plattformmanagements zwei Mushu-Plattformen auf der selben physikalischen Maschine zu platzieren und beide Maschinen durch in-memory Synchronisation kommunizieren zu lassen. Erste Untersuchungen in diese Richtung liefert die Arbeit (Feldmann, 2019).

- *Verteiltes Undo.* Unter der Voraussetzung, dass ein verteilter Simulationsfeed, wie er im Rahmen des Realisierungskonzepts der Plattform diskutiert wurde, als Implementation zur Verfügung steht, ist die Implementation eines verteilten »Undo« denkbar. Die Überlegung bezieht sich dabei auf die letzte (verteilte) Aktion bzw. alle Aktionen bis inklusive einer bestimmten Aktion der Historie. Das Undo eines isolierten, beliebigen Schrittes der Historie ist selbst im lokalen Fall nicht eindeutig zu lösen. Dies liegt darin begründet, dass an keiner Stelle eine Disjunktivität der Daten und Strukturen, auf denen Operationen agieren, gefordert wird. Die Auswirkungen der Rücknahme einer einzelnen Operation kann durch die Nebenläufigkeit und Unabhängigkeit nicht mehr eindeutig zugeordnet werden. Beim angesprochenen verteilten Undo wäre somit das erklärte Ziel die Zurücksetzung des letzten Feuervorgangs eines verteilten synchronen Kanals. Da alle lokalen Simulatoren nach dem Feuervorgang im Zweifel entsprechende weitere Berechnung durchgeführt haben, ist auch dieses Vorhaben aufwendig und nicht trivial umzusetzen. Der integrierte Simulationsfeed und insbesondere True-Concurrency Checkpoints bieten hierbei gute Rahmenvoraussetzungen für eine entsprechende Umsetzung. Dennoch sind diverse technische und konzeptionelle Hürden zu überwinden, bevor ein verteiltes Undo realisiert werden kann, sodass dieses Unterfangen als äußerst komplex anzusehen ist. Im Kontext von etwaiger nebenläufiger Anpassung von Netzmodellen selbst erhöht sich diese Komplexität noch weiter. Eine möglicher Ansatz wäre die Realisierung als insgesamt kompensierende bzw. abmildernde Form, bei der das Gesamtsystem einen anderen aber akzeptablen Zustand einnimmt, anstatt eines strengen Wiederherstellen des Ausgangszustandes. Dies ist insbesondere bei irreversiblen Operationen eine sinnvolle Einschränkung. Unter dieser Maßgabe könnte auf einer höheren Abstraktonsebene ein ganzer Verlauf von Ereignissen als Saga interpretiert werden und mit den in der

Arbeit dargelegten Ergebnissen zu Petrinetz-Sagas beschrieben, ausgeführt und ggf. auch kompensiert werden.

- *Formalisierung von* Mushu. Im Rahmen der Vorstellung von Mushu wurde auf die Formalisierung der Architektur verzichtet. Analog zum Aufbau von Referenznetzen auf Graphersetzungssystemen könnte eine entsprechende Formalisierung zunächst für Mulan und daran anschließend ebenfalls für Mushu erfolgen. Mit einer formalen Basis könnte untersucht werden inwiefern Beweise und Verifikationsfragen im Rahmen von Mushu beantwortet werden können. In einem ersten Schritt könnten Teile der Modelle durch gezielte Modellierungskonzepte beschränkt und dann lokal analysiert werden. Details hierzu wurden bereits in Abschnitt 15.3.2 erläutert.

- *Integration von Binary Squares in weitere Algorithmen.* Die Theorie von Binary Squares wurde abstrakt vorgestellt und auf den Algorithmus Scale Invariant Feature Transform (SIFT) beispielhaft projiziert. Die Umsetzbarkeit in weiteren Algorithmen, möglicherweise auch jenseits der Mustererkennung, kann daher ein interessantes Forschungsfeld umfassen. Die Ergebnisse sind immer dann anwendbar, wenn eine große Summe von Quadraten gegen einen Schwellenwert abgeglichen wird.

- *Beschreibung des vollen Weges der Szenenanalyse im Kontext von* Mushu. Im Rahmen der Arbeit wurde der Weg mittels Szenenanalyse aus der realen Welt Modelle auf Basis der Mushu-Architektur abzuleiten, motiviert. Es wurde ebenfalls motiviert, dass die vollständige Konzeption und Umsetzung dieses Prozesses der Gegenstand mehrerer eigener Arbeiten sein kann. Konkrete Möglichkeiten sind die Erörterung notwendiger Vor- und Rahmenbedingungen wie beispielsweise betrachtete Szenen und eingesetzte Sensoren, Modellierung des entsprechenden Wissens, die Auswahl der Algorithmen zu jedem der Schritte und abschließend die Transformation der einzelnen Erkenntnisse in einzelne Agenten im Rahmen der Mushu-Architektur. Darauf aufbauende Arbeiten können dann Ableitungen aus der massiven Simulation derartiger Agentensysteme umfassen, welche wiederum durch die Skalierungsmöglichkeiten der Mushu-Architektur realisierbar sind.

- *Weiterentwicklung von* Organ *und Sozialsimulationen.* Im Rahmen der Arbeit wurde bereits die Multiorganisationsarchitektur Organ (Wester-Ebbinghaus, 2010) referenziert, welche oberhalb der Mulan-Architektur angesiedelt ist. Im Organ-Kontext lassen sich Mushu-Plattformmanagements als Bereiche in Organisationen interpretieren. Auf der Basis ist auch die Simulation und Analyse von sozialen Prozessen denkbar. Der Beitrag durch Mushu besteht zum einen in der Skalierbarkeit solcher Simulationen und die Möglichkeit auch große und dynamische Modelle zu simulieren. Zum anderen kann die konzeptionelle Erweiterung durch die Nutzung nebenläufiger Sagas zur verbesserten technischen Umsetzung beitragen. Ferner ergibt

sich eine Verbesserung dadurch, dass in ORGAN zwar Einheiten in mehreren Kontexten existieren können, aber erst auf der konzeptuellen Basis von MUSHU die Möglichkeit besteht, dass sich Einheiten neue Kommunikationsplattformen erstellen. Diese umfassen eine volle Plattform mit Agenten, Verhalten und Strutkur und basieren nicht nur auf einfachem Nachrichtenaustausch.

Insgesamt konnte durch die vorliegende Arbeit entsprechend den Vorhaben erfolgreich gezeigt werden, dass die Modellierung von explizit nebenläufigen Agentensystemen und die Abbildung auf ein reales verteiltes Simulationssystem auch unter der Maßgabe möglich sind, dass Agenten in der Lage sind ihre Kommunikationsplattformen (proaktiv) zu ändern, zu erweitern und zu skalieren.

Teil VI.

Anhang

A. Veröffentlichungen und Chronologie

In diesem Anhang finden explizit alle Veröffentlichungen des Autors Erwähnung, welche direkten Bezug zur dieser Arbeit haben und welche im Rahmen des Promotionsvorhabens entstanden sind. Dabei sind alle genannten Veröffentlichungen maßgeblich vom Autor dieser Arbeit als Erstautor verfasst worden. Die einzige Ausnahme bildet dabei die Veröffentlichung (Moldt, Röwekamp und M. Simon, 2017), bei der der Autor jedoch auch große Anteile hatte.

In der frühen Phase der Forschung gaben die Bachelor- (Röwekamp, 2011) und Master-Thesis (Röwekamp, 2013) des Autors den initialen Anstoß Konzepte der Bildverarbeitung und theoretischen Informatik zu vereinen. Daraus entstand zunächst das Konzept von Binary Squares, welches im Detail in Kapitel 13 behandelt wurde. In einer frühen Phase wurde Binary Squaring noch in direkter Anbindung an Referenznetze vorgestellt (Röwekamp und Haustermann, 2015). Die direkte Verknüpfung zu Petrinetzen entpuppte sich jedoch bald als nicht zielführend, da die Untersuchungen sich auf einer sehr niedrigen Ebene abspielten, auf der Petrinetze ihre Modellierungsfähigkeit nicht gut ausspielen können. Daher wurde in der nächsten Iteration auf den Petrinetz-Unterbau verzichtet und stattdessen der Fokus auf direkte analytische Beweise verschoben. Zusammen mit einer Proof-of-Concept Implementation in der OpenSIFT Library und einer Beispielevaluation schaffte es das Novum bis auf die International Conference on Pattern Recognition (ICPR) 2016 (Röwekamp, 2016).

Nach der Betrachtung der Abläufe, wie sie im Einleitungsteil der Arbeit geschildert wurden, wurde klar, dass eine höhere Abstraktionsebene eingenommen werden muss. Auf der Basis von Nebenläufigkeit, Agentenorientiertheit, verteilten Systemen, der Unterscheidung und Ausnutzung verteilter und lokaler Anteile in Berechnungen fiel die Wahl der weiteren Untersuchungen auf Mulan und den Renew Simulator. Eine Verteilung existierte bereits durch das Distribute Plugin, jedoch keine Skalierungsmöglichkeit und auch kein ausreichend ausgeführtes Konzept für eine solche.

In ersten Untersuchungen entstanden einige Prototypen auf der Basis von virtuellen Maschinen (Moldt, Röwekamp und M. Simon, 2017) und dem Distribute Plugin. Aufbauend auf diesen ersten Vorstößen folgte eine Implementati-

on als containerisierte Anwendung auf Basis von Docker (Röwekamp, Moldt und Feldmann, 2018). Schnell wurde jedoch klar, dass eine skalierbare Architektur auf Plattformebene entsprechende Unterstützung aus dem Simulator heraus benötigt. Dazu wurden verschiedene Ideen in direktem Bezug auf das Spring-Framework vorgestellt (Röwekamp, 2018), welches schlussendlich auch im Cloud-Native Renew-Plugin, welches im Rahmen der Arbeit implementiert wurde, Einzug gefunden hat. Schlussendlich wurde eine finale Fassung eines skalierbaren, containerisierten Referenznetzsimulators und der zugehörige Prototyp RenewKube auf der Petri Nets 2019 Hauptkonferenz vorgestellt (Röwekamp und Moldt, 2019).

Dabei wurden zusätzlich einige Überlegungen zu grundlegenden Architekturumstellungen für spätere Vorgehen vorgestellt (Röwekamp, Feldmann u. a., 2019). Diese wurden jedoch nur auf konzeptueller Ebene vorgestellt und wegen eines antizipierten Performanceproblems nicht weiter verfolgt. Stattdessen bezogen sich weitere Untersuchungen auf die Ausgestaltung der Plattform auf der Basis einer Cloud-Native-Architektur (Röwekamp, Taube u. a., 2021). Auf dieser Basis wurde ebenfalls die komplexe Agentenkommunikation konzeptualisiert und in ein Realisierungskonzept gegossen (Röwekamp, Buchholz und Moldt, 2021).

Mit den zentralen vom Autor der Arbeit beigetragenen Stützpfeilern des Plattformmanagements, der Cloud-nativen Plattform und der komplexen Agentenkommunikation war es sodann möglich die gesamte Architektur Mushu mit zugehörigem Konzept und einer möglichen Realisierung zu beschreiben, wie es in dieser Arbeit erfolgt ist.

Gesamtübersicht über alle Veröffentlichungen und Abschlussarbeiten

Röwekamp, Jan Henrik (Mai 2011). »Algorithmische Betrachtung von Switching Graphs«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 461).

Röwekamp, Jan Henrik (Dez. 2013). »Komplexitätstheoretische Betrachtungen an einer probabilistischen Abwandlung der Nearest Neighbour-Suche in kd-Trees«. Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 461).

Röwekamp, Jan Henrik und Haustermann, Michael (2015). »Applying Petri Nets to Approximation of the Euclidean Distance with the Example of SIFT«. In: *Petri Nets and Software Engineering. International Workshop, PNSE'15, Brussels, Belgium, June 22-23, 2015. Proceedings*. Hrsg. von Moldt, Daniel,

Rölke, Heiko und Störrle, Harald. Bd. 1372. CEUR Workshop Proceedings. CEUR-WS.org, S. 323–324 (siehe S. 365, 461).

Röwekamp, Jan Henrik (2016). »Fast thresholding of high dimensional Euclidean distances using binary squaring«. In: *23rd International Conference on Pattern Recognition, ICPR 2016, Cancún, Mexico, December 4-8, 2016*. IEEE, S. 3103–3108. ISBN: 978-1-5090-4847-2. DOI: 10.1109/ICPR.2016.7900111 (siehe S. 365, 367, 461).

Moldt, Daniel, Röwekamp, Jan Henrik und Simon, Michael (2017). »A Simple Prototype of Distributed Execution of Reference Nets Based on Virtual Machines«. In: *Algorithms and Tools for Petri Nets Proceedings of the Workshop AWPN 2017, Kgs. Lyngby, Denmark October 19-20, 2017*. Hrsg. von Bergenthum, Robin und Kindler, Ekkart. DTU Compute Technical Report 2017-06, S. 51–57 (siehe S. 214, 257, 387, 461).

Röwekamp, Jan Henrik (2018). »Investigating the Java Spring Framework to Simulate Reference Nets with Renew«. In: *Algorithms and Tools for Petri Nets, Proceedings of the 21th Workshop AWPN 2018, Augsburg, Germany*. Hrsg. von Lorenz, Robert und Metzger, Johannes. Reports / Technische Berichte der Fakultät für Angewandte Informatik der Universität Augsburg 2018-02, S. 41–46 (siehe S. 462).

Röwekamp, Jan Henrik, Moldt, Daniel und Feldmann, Matthias (2018). »Investigation of Containerizing Distributed Petri Net Simulations«. In: *Petri Nets and Software Engineering. International Workshop, PNSE'18, Bratislava, Slovakia, June 25-26, 2018. Proceedings*. Hrsg. von Moldt, Daniel, Kindler, Ekkart und Rölke, Heiko. Bd. 2138. CEUR Workshop Proceedings. CEUR-WS.org, S. 133–142 (siehe S. 207, 214, 220, 387, 462).

Röwekamp, Jan Henrik, Feldmann, Matthias, Moldt, Daniel und Simon, Michael (2019). »Simulating Place / Transition Nets by a Distributed, Web Based, Stateless Service«. In: *Petri Nets and Software Engineering. International Workshop, PNSE'19, Aachen, Germany, June 24, 2019. Proceedings*. Hrsg. von Moldt, Daniel, Kindler, Ekkart und Wimmer, Manuel. Bd. 2424. CEUR Workshop Proceedings. CEUR-WS.org, S. 163–164 (siehe S. 91, 404, 462).

Röwekamp, Jan Henrik und Moldt, Daniel (2019). »RenewKube: Reference Net Simulation Scaling with Renew and Kubernetes«. In: *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings*. Hrsg. von Donatelli, Susanna und Haar, Stefan. Bd. 11522. Lecture Notes in Computer Science. Springer, S. 69–79. ISBN: 978-3-030-21570-5. DOI: 10.1007/978-3-030-21571-2_4 (siehe S. 214, 462).

Röwekamp, Jan Henrik, Buchholz, Manuela und Moldt, Daniel (2021). »Petri Net Sagas«. In: *Proceedings of the International Workshop on Petri Nets*

and Software Engineering 2021 co-located with the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021), Paris, France, June 25th, 2021 (due to COVID-19: virtual conference). Hrsg. von Köhler-Bussmeier, Michael, Kindler, Ekkart und Rölke, Heiko. Bd. 2907. CEUR Workshop Proceedings. CEUR-WS.org, S. 65–84 (siehe S. 311, 410, 462).

Röwekamp, Jan Henrik, Taube, Marvin, Mohr, Patrick und Moldt, Daniel (2021). »Cloud Native Simulation of Reference Nets«. In: *Proceedings of the International Workshop on Petri Nets and Software Engineering 2021 co-located with the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021), Paris, France, June 25th, 2021 (due to COVID-19: virtual conference)*. Hrsg. von Köhler-Bussmeier, Michael, Kindler, Ekkart und Rölke, Heiko. Bd. 2907. CEUR Workshop Proceedings. CEUR-WS.org, S. 85–104 (siehe S. 277, 406, 462).

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

Aalst, Wil van der (1997). »Verification of Workflow Nets«. In: *Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings*. Hrsg. von Azéma, Pierre und Balbo, Gianfranco. Bd. 1248. Lecture Notes in Computer Science. Springer, S. 407–426. doi: 10.1007/3-540-63139-9_48 (siehe S. 23, 24).

Aalst, Wil van der, Moldt, Daniel, Valk, Rüdiger und Wienberg, Frank (1999). »Enacting Interorganizational Workflows Using Nets in Nets«. In: *Proceedings of the 1999 Workflow Management Conference Workflow-based Applications, Münster, Nov. 9th 1999*. Hrsg. von Becker, Jörg, Mühlen, Michael zur und Rosemann, Michael. Working Paper Series of the Department of Information Systems. Working Paper No. 70. Department of Information Systems, Steinfurter Str. 109, 48149 Münster: University of Münster, S. 117–136 (siehe S. 102, 127, 135, 319).

Akutsu, Tatsuya, Jansson, Jesper, Takasu, Atsuhiro und Tamura, Takeyuki (2017). »On the parameterized complexity of associative and commutative unification«. In: *Theoretical Computer Science* 660, S. 57–74. doi: 10.1016/j.tcs.2016.11.026 (siehe S. 135).

Aldossary, Mohammad und Alharbi, Hatem A. (2021). »Towards a Green Approach for Minimizing Carbon Emissions in Fog-Cloud Architecture«. In: *IEEE Access* 9, S. 131720–131732. doi: 10.1109/ACCESS.2021.3114514 (siehe S. 107).

Amdeberhan, Tewodros, Manna, Dante und Moll, Victor H. (2008). »The 2-adic Valuation of Stirling Numbers«. In: *Experimental Mathematics* 17.1, S. 69–82. doi: 10.1080/10586458.2008.10129026 (siehe S. 371).

Andrews, Greg R (1999). *Foundations of Parallel and Distributed Programming*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc. isbn: 0201357526 (siehe S. 36).

Arshed, Jawad Usman und Ahmed, Masroor (2021). »RACE: Resource Aware Cost-Efficient Scheduler for Cloud Fog Environment«. In: *IEEE Access* 9, S. 65688–65701. doi: 10.1109/ACCESS.2021.3068817 (siehe S. 107).

Aschermann, Malte, Dennisen, Sophie, Kraus, Philipp und Müller, Jörg P. (2018). »LightJason, a Highly Scalable and Concurrent Agent Framework:

Overview and Application«. In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '18. Stockholm, Sweden: International Foundation for Autonomous Agents und Multiagent Systems, S. 1794–1796 (siehe S. 92).

Balla, David, Simon, Csaba und Maliosz, Markosz (2020). »Adaptive scaling of Kubernetes pods«. In: *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, S. 1–5. doi: 10.1109/NOMS47738.2020.9110428 (siehe S. 106).

Baratloo, Arash, Chung, P. Emerald, Huang, Yennun, Rangarajan, Sampath und Yajnik, Shalini (1998). »Filterfresh: Hot Replication of Java RMI Server Objects«. In: *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), April 27-30, 1998, Eldorado Hotel, Santa Fe, New Mexico, USA*. USENIX, S. 65–78 (siehe S. 248, 349).

Bartsch, Thomas, Dreschler-Fischer, Leonie und Schröder, Carsten (1986). »Merkmalsdetektion in Farbbildern als Grundlage zur Korrespondenzanalyse in Stereo-Bildfolgen«. In: *Mustererkennung 1986, 8. DAGM-Symposium, Paderborn 30. September - 2. Oktober 1986, Proceedings*. Hrsg. von Hartmann, Georg. Bd. 125. Informatik-Fachberichte. Springer, S. 94–98. doi: 10.1007/978-3-642-71387-3_17 (siehe S. 104).

Bass, L., Clements, P., Kazman, R. und Safari (2021). *Software Architecture in Practice, 4th Edition*. Addison-Wesley Professional, an O'Reilly Media Company. isbn: 9780136885979 (siehe S. 34).

Becker, Ulrich und Moldt, Daniel (Okt. 1993a). »Object-Oriented Concepts for Coloured Petri Nets«. In: *Systems, Man and Cybernetics, 1993. 'Systems Engineering in the Service of Humans', Conference Proceedings., IEEE, International Conference on*. Bd. 3. Le Touquet, France: IEEE, S. 279–285. doi: 10.1109/ICSMC.1993.385024 (siehe S. 101).

Becker, Ulrich und Moldt, Daniel (1993b). »Objekt-orientierte Konzepte für gefärbte Petrinetze«. In: *Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen*. Hrsg. von Scheschonk, Gert und Reisig, Wolfgang. Informatik Aktuell. Gesellschaft für Informatik. Berlin, Heidelberg, New York: Springer-Verlag, S. 140–151 (siehe S. 101).

Beese, Felix (Juli 2021). »Untersuchung von Optionen bei der Erweiterung einer Multiagentenanwendung im Kontext von PAOSE am Beispiel von Settler«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 95).

Bellifemine, Fabio, Poggi, Agostino und Rimassa, Giovanni (1999). »JADE - A FIPA-compliant agent framework«. In: *Proceedings of the Practical Applications of Intelligent Agents*, S. 97–108 (siehe S. 89).

Bellifemine, Fabio, Poggi, Agostino und Rimassa, Giovanni (2000). »Developing Multi-agent Systems with JADE«. In: *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop, ATAL 2000, Boston, MA, USA, July 7-9, 2000, Proceedings*. Hrsg. von Castelfranchi, Cristiano und Lespérance, Yves. Bd. 1986. Lecture Notes in Computer Science. Springer, S. 89–103. doi: 10.1007/3-540-44631-1_7 (siehe S. 89).

Bendoukha, Sofiane (2013). »Multi-agent Approach for Managing Workflows in an Inter-Cloud Environment«. In: *Service-Oriented Computing - ICSOC 2013 Workshops - CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium, Berlin, Germany, December 2-5, 2013. Revised Selected Papers*. Hrsg. von Lomuscio, Alessio, Nepal, Surya, Patrizi, Fabio, Benatallah, Boualem und Brandić, Ivona. Bd. 8377. Lecture Notes in Computer Science. Springer, S. 535–542. isbn: 978-3-319-06858-9. doi: 10.1007/978-3-319-06859-6_48 (siehe S. 82, 103).

Bendoukha, Sofiane (2017). »Multi-Agent Approach for Managing Workflows in an Inter-Cloud Environment«. eng. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 82–84, 103, 202, 208, 427).

Bendoukha, Sofiane, Bendoukha, Hayat und Moldt, Daniel (2015a). »ICNETS: Towards Designing Inter-Cloud Workflow Management Systems by Petri Nets«. In: *Enterprise and Organizational Modeling and Simulation - 11th International Workshop, EOMAS 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8-9, 2015, Selected Papers*. Hrsg. von Barjis, Joseph, Pergl, Robert und Babkin, Eduard. Bd. 231. Lecture Notes in Business Information Processing. Springer, S. 187–198. isbn: 978-3-319-24625-3. doi: 10.1007/978-3-319-24626-0_14 (siehe S. 82, 103).

Bendoukha, Sofiane, Bendoukha, Hayat und Moldt, Daniel (2015b). »RenewGrass - A Tool for Building Scientific Workflows: Application to the Remote Sensing Domain«. In: *2015 IEEE International Conference on Information Reuse and Integration, IRI 2015, San Francisco, CA, USA, August 13-15, 2015*. Hrsg. von Rubin, Stuart H. und Chen, Shu-Ching. IEEE, S. 311–317. isbn: 978-1-4673-6656-4. doi: 10.1109/IRI.2015.67 (siehe S. 82, 103).

Bendoukha, Sofiane, Moldt, Daniel und Wagner, Thomas (Aug. 2013). »Enabling Cooperation in an Inter-Cloud Environment: An Agent-based Approach«. In: *Database and Expert Systems Applications (DEXA), 2013 24th International Workshop on*. Hrsg. von Morvan, Franck, Tjoa, A Min und Wagner, Roland. IEEE Computer Society, S. 217–221. isbn: 978-0-7695-5070-1. doi: 10.1109/DEXA.2013.27 (siehe S. 103).

Bendoukha, Sofiane und Wagner, Thomas (Juni 2012). »Cloud Transition: Integrating Cloud Calls into Workflow Petri Nets«. In: *Petri Nets and Soft-*

ware Engineering. International Workshop PNSE'12, Hamburg, Germany, June 2012. Proceedings. Hrsg. von Cabac, Lawrence, Duvigneau, Michael und Moldt, Daniel. Bd. 851. CEUR Workshop Proceedings. CEUR-WS.org, S. 215–216 (siehe S. 103).

Bentley, Jon Louis (Sep. 1975). »Multidimensional Binary Search Trees Used for Associative Searching«. In: *Communications of the ACM* 18.9, S. 509–517. issn: 0001-0782. doi: 10.1145/361002.361007 (siehe S. 413).

Bertozzi, Davide, Miorandi, Gabriele, Ghiribaldi, Alberto, Burleson, Wayne P., Sadowski, Greg, Bhardwaj, Kshitij u. a. (2021). »Cost-Effective and Flexible Asynchronous Interconnect Technology for GALS Systems«. In: *IEEE Micro* 41.1, S. 69–81. doi: 10.1109/MM.2020.3002790 (siehe S. 105).

Betz, Tobias (Sep. 2011). »Entwicklung eines Rahmenwerks für webbasierte Agentendienste – Modellierung auf Basis von Petrinetzen«. Diplomarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 98).

Bordini, Rafael H., Hübner, Jomi Fred und Vieira, Renata (2005). »Jason and the Golden Fleece of Agent-Oriented Programming«. In: *Multi-Agent Programming: Languages, Platforms and Applications*. Hrsg. von Bordini, Rafael H., Dastani, Mehdi, Dix, Jürgen und Seghrouchni, Amal El Fallah. Bd. 15. Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, S. 3–37 (siehe S. 92).

Bosse, Stefan (2021). »Parallel and Distributed Agent-based Simulation of Large-scale Socio-technical Systems with Loosely Coupled Virtual Machines«. In: *Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications, SIMULTECH 2021, Online Streaming, July 7-9, 2021*. Hrsg. von Wagner, Gerd, Werner, Frank, Ören, Tuncer I. und Rango, Floriano De. SCITEPRESS, S. 344–351. doi: 10.5220/0010553003440351 (siehe S. 92).

Bosse, Stefan und Engel, Uwe (2019). »Real-Time Human-In-The-Loop Simulation with Mobile Agents, Chat Bots, and Crowd Sensing for Smart Cities«. In: *Sensors* 19.20, S. 4356. doi: 10.3390/s19204356 (siehe S. 92).

Bratman, M. (1987). *Intention, plans, and practical reason*. Cambridge, MA: Harvard University Press. isbn: 9780674458185 (siehe S. 5).

Braubach, Lars (2007). »Architekturen und Methoden zur Entwicklung verteilter agentenorientierter Softwaresysteme«. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik. isbn: 978-3-00-023107-0 (siehe S. 99).

Braubach, Lars, Lamersdorf, Winfried und Pokahr, Alexander (Dez. 2003). »Jadex: implementing a BDI-infrastructure for JADE agents«. In: *EXP In Search of Innovation (Special Issue on JADE)* 3 (siehe S. 99).

Braubach, Lars und Pokahr, Alexander (2013). »The Jadex Project: Simulation«. In: *Multiagent Systems and Applications - Volume 1: Practice and Experience*. Hrsg. von Ganzha, Maria und Jain, Lakhmi C. Bd. 45. Intelligent Systems Reference Library. Springer, S. 107–128. doi: 10.1007/978-3-642-33323-1_5 (siehe S. 99).

Brewer, Eric A. (2000). »Towards Robust Distributed Systems (Abstract)«. In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: Association for Computing Machinery, S. 7. isbn: 1581131836. doi: 10.1145/343477.343502 (siehe S. 39).

Brügge, Bernd und Dutoit, Allen H. (2004). *Object-oriented software engineering - using UML, patterns and Java (2. ed.)* Prentice Hall. isbn: 978-0-13-191179-6 (siehe S. 35).

Buchs, Didier, Flumet, Jacques und Racloz, Pascal (1992). »Producing prototypes from CO-OPN specifications«. In: *Proceedings of the Third International Workshop on Rapid System Prototyping, RSP 1992, Research Triangle Park, North Carolina, USA, June 23-15, 1992*. IEEE Computer Society, S. 77–93. doi: 10.1109/IWRSP.1992.243915 (siehe S. 6).

Buchs, Didier und Guelfi, Nicolas (2000). »A Formal Specification Framework for Object-Oriented Distributed Systems«. In: *IEEE Trans. Software Eng.* 26.7, S. 635–652. doi: 10.1109/32.859532 (siehe S. 6).

Bulla, Chetan M. und Birje, Mahantesh N. (2021). »A Multi-Agent-Based Data Collection and Aggregation Model for Fog-Enabled Cloud Monitoring«. In: *International Journal of Cloud Applications and Computing* 11.1, S. 73–92. doi: 10.4018/IJCAC.2021010104 (siehe S. 107).

Cabac, Lawrence (Sep. 2003). »Generating Code Structures for Petri Net-Based Agent Interaction Protocols Using Net Components«. In: *Workshop: Algorithms and Tools for Petri Nets* (siehe S. 103, 318).

Cabac, Lawrence (2007). »Multi-Agent System: A Guiding Metaphor for the Organization of Software Development Projects«. In: *Proceedings of the Fifth German Conference on Multiagent System Technologies*. Hrsg. von Petta, Paolo. Bd. 4687. Lecture Notes in Computer Science. Leipzig, Germany: Springer-Verlag, S. 1–12. doi: 10.1007/978-3-540-74949-3_1 (siehe S. 6, 32).

Cabac, Lawrence (Juni 2009). »Net Components: Concepts, Tool, Praxis«. In: *Petri Nets and Software Engineering, International Workshop, PNSE'09. Proceedings*. Hrsg. von Moldt, Daniel. Technical Reports Université Paris 13. 99,

avenue Jean-Baptiste Clément, 93 430 Villetaneuse: Université Paris 13, S. 17–33 (siehe S. 318).

Cabac, Lawrence (Apr. 2010). »Modeling Petri Net-Based Multi-Agent Applications«. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik (siehe S. 5, 6, 32, 33, 67, 95, 97, 103, 150, 279, 318, 434).

Cabac, Lawrence, Dörges, Till, Duvigneau, Michael, Moldt, Daniel, Reese, Christine und Wester-Ebbinghaus, Matthias (2008). »Agent Models for Concurrent Software Systems«. In: *Proceedings of the Sixth German Conference on Multiagent System Technologies, MATES'08*. Hrsg. von Bergmann, Ralph und Lindemann, Gabriela. Bd. 5244. Lecture Notes in Artificial Intelligence. Berlin, Heidelberg, New York: Springer-Verlag, S. 37–48 (siehe S. 6, 32).

Cabac, Lawrence, Dörges, Till, Duvigneau, Michael, Reese, Christine und Wester-Ebbinghaus, Matthias (Juni 2007). »Application Development with Mulan«. In: *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07)*. Hrsg. von Moldt, Daniel, Kordon, Fabrice, Hee, Kees van, Colom, José-Manuel und Bastide, Rémi. Siedlce, Poland: Akademia Podlaska, S. 145–159 (siehe S. 96).

Cabac, Lawrence, Duvigneau, Michael und Moldt, Daniel, Hrsg. (Juni 2012). *Petri Nets and Software Engineering. International Workshop PNSE'12, Hamburg, Germany, June 2012. Proceedings*. Bd. 851. CEUR Workshop Proceedings. CEUR-WS.org.

Cabac, Lawrence, Duvigneau, Michael, Moldt, Daniel und Rölke, Heiko (2005). »Agent Technologies for Plug-in System Architecture Design«. In: *Proceedings of the Workshop on Agent-oriented Software Engineering (AOSE)*. Utrecht, Netherlands (siehe S. 102).

Cabac, Lawrence, Duvigneau, Michael, Moldt, Daniel und Rölke, Heiko (Juni 2006). »Applying Multi-agent Concepts to Dynamic Plug-in Architectures«. In: *Agent-Oriented Software Engineering VI: 6th International Workshop, AOSE 2005, Utrecht, Netherlands, July 21, 2005. Revised Selected Papers*. Hrsg. von Mueller, Jörg und Zambonelli, Franco. Bd. 3950. Lecture Notes in Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, S. 190–204. doi: 10.1007/11752660_15 (siehe S. 102).

Cabac, Lawrence, Duvigneau, Michael, Moldt, Daniel und Schleinzer, Benjamin (2007). »Plugin-Agents as Conceptual Basis for Flexible Software Structures«. In: *Multi-Agent Systems and Applications V. Fifth International Central and East European Conference, CEEMAS'07, Leipzig. Proceedings*. Bd. 4696. Lecture Notes in Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, S. 340–342. doi: 10.1007/978-3-540-75254-7_44 (siehe S. 102).

Cabac, Lawrence, Duvigneau, Michael, Reese, Christine, Dörges, Till und Wester-Ebbinghaus, Matthias (2007). »Models and Tools for Mulan Applications«. In: *Multi-Agent Systems and Applications V. Fifth International Central and East European Conference, CEEMAS'07, Leipzig. Proceedings*. Hrsg. von Burkhard, H.-D., Lindemann, G., Verbrugge, R. und Varga, L. Bd. 4696. Lecture Notes in Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, S. 328–330. doi: 10.1007/978-3-540-75254-7_40 (siehe S. 103).

Cabac, Lawrence, Haustermann, Michael und Mosteller, David (2018). »Software development with Petri nets and agents: Approach, frameworks and tool set«. In: *Science of Computer Programming* 157. Hrsg. von Kordon, Fabrice und Moldt, Daniel, S. 56–70. doi: 10.1016/j.scico.2017.12.003 (siehe S. 95).

Cabac, Lawrence, Kristensen, Lars Michael und Rölke, Heiko, Hrsg. (2016). *Petri Nets and Software Engineering. International Workshop, PNSE'16, Toruń, Poland, June 20-21, 2016. Proceedings*. Bd. 1591. CEUR Workshop Proceedings. CEUR-WS.org.

Cabac, Lawrence, Moldt, Daniel und Rölke, Heiko (Juni 2003). »A Proposal for Structuring Petri Net-Based Agent Interaction Protocols«. In: *24th International Conference on Application and Theory of Petri Nets, Eindhoven, Netherlands, June 2003*. Hrsg. von Aalst, Wil van der und Best, Eike. Bd. 2679. Lecture Notes in Computer Science. Springer-Verlag, S. 102–120 (siehe S. 103).

Cabac, Lawrence, Moldt, Daniel, Wester-Ebbinghaus, Matthias und Müller, Eva (Sep. 2009). »Visual Representation of Mobile Agents – Modeling Mobility within the Prototype MAPA«. In: *Proceedings of the Fifth International Workshop on Modeling of Objects, Components and Agents, MOCA'09, Hamburg*. Hrsg. von Duvigneau, Michael und Moldt, Daniel. Bericht FBI-HH-B-290/09. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik, S. 7–28 (siehe S. 128, 161).

Casquero, Oskar, Armentia, Aintzane, Sarachaga, Isabel, Pérez, Federico, Orive, Dario und Marcos, Marga (2019). »Distributed scheduling in Kubernetes based on MAS for Fog-in-the-loop applications«. In: *24th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2019, Zaragoza, Spain, September 10-13, 2019*. IEEE, S. 1213–1217. doi: 10.1109/ETFA.2019.8869219 (siehe S. 93).

Castro-Jul, Fátima, Conan, Denis, Chabridon, Sophie, Redondo, Rebeca P. Díaz, Vilas, Ana Fernández und Taconet, Chantal (2017). »Combining Fog Architectures and Distributed Event-Based Systems for Mobile Sensor Location Certification«. In: *Ubiquitous Computing and Ambient Intelligence - 11th International Conference, UCAmI 2017, Philadelphia, PA, USA, November 7-*

10, 2017, Proceedings. Hrsg. von Ochoa, Sergio F., Singh, Pritpal und Bravo, José. Bd. 10586. Lecture Notes in Computer Science. Springer, S. 27–33. doi: 10.1007/978-3-319-67585-5_3 (siehe S. 107).

Castro-Jul, Fátima, Redondo, Rebeca P. Díaz, Vilas, Ana Fernández, Chabridon, Sophie und Conan, Denis (2019). »Fog Architectures and Sensor Location Certification in Distributed Event-Based Systems«. In: *Sensors* 19.1, S. 104. doi: 10.3390/s19010104 (siehe S. 107).

Celebi, M. Emre, Celiker, Fatih und Kingravi, Hassan A. (2011). »On Euclidean norm approximations«. In: *Pattern Recognition* 44.2, S. 278–283. doi: 10.1016/j.patcog.2010.08.028 (siehe S. 368).

Chen, Shuyi, Hanai, Masatoshi, Hua, Zhengchang, Tziritas, Nikos und Theodoropoulos, Georgios (2020). »Efficient Direct Agent Interaction in Optimistic Distributed Multi-Agent-System Simulations«. In: *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2020, Miami, FL, USA, June 15-17, 2020*. Hrsg. von Liu, Jason, Giabbanelli, Philippe J. und Carothers, Christopher D. ACM, S. 123–128. doi: 10.1145/3384441.3395977 (siehe S. 105).

Christensen, James (Dez. 1994). »Holonic Manufacturing Systems: Initial Architecture and Standards Directions«. In: *Proceedings of the 1st European Workshop on Holonic Manufacturing Systems* (siehe S. 6, 159).

Christensen, S. und Hansen, N. (1994). »Coloured Petri Nets Extended with Channels for Synchronous Communication«. In: *Application and Theory of Petri Nets 1994*. Hrsg. von Robert, Valette. Bd. 815. Lecture Notes in Computer Science. Springer-Verlag, S. 159–178. isbn: 3-540-58152-9 (siehe S. 24, 26, 27, 102).

Chu, Hsiao-Keng Jerry (1996). »Zero-Copy TCP in Solaris«. In: *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*. USENIX Association, S. 253–264 (siehe S. 239).

Clemen, Thomas, Ahmady-Moghaddam, Nima, Lenfers, Ulfia A., Ocker, Florian, Osterholz, Daniel, Ströbele, Jonathan u. a. (2021). »Multi-Agent Systems and Digital Twins for Smarter Cities«. In: *SIGSIM-PADS '21: SIGSIM Conference on Principles of Advanced Discrete Simulation, Virtual Event, USA, 31 May - 2 June, 2021*. Hrsg. von Giabbanelli, Philippe J. ACM, S. 45–55. doi: 10.1145/3437959.3459254 (siehe S. 105).

Constantinescu, Ion, Willmott, S. und Dale, J. (2003). *D2.3: Agentcities Network Architecture* (siehe S. 94).

Conway, Melvin E. (Apr. 1968). »How Do Committees Invent?« In: *Datamation* (siehe S. 32, 158).

Cormen, T., Stein, C., Rivest, R. und Leiserson, C. (2001). *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education. ISBN: 0070131511 (siehe S. 328).

Councill, Bill und Heineman, George T. (2001). »Definition of a Software Component and Its Elements«. In: *Component-Based Software Engineering: Putting the Pieces Together*. USA: Addison-Wesley Longman Publishing Co., Inc., S. 5–19. ISBN: 0201704854 (siehe S. 75).

Cristian, Flaviu (1989). »Probabilistic Clock Synchronization«. In: *Distributed Computing* 3.3, S. 146–158. DOI: 10.1007/BF01784024 (siehe S. 302).

Dähling, Stefan, Razik, Lukas und Monti, Antonello (2021). »Enabling scalable and fault-tolerant multi-agent systems by utilizing cloud-native computing«. In: *Autonomous Agents and Multi-Agent Systems* 35.1, S. 10. DOI: 10.1007/s10458-020-09489-0 (siehe S. 90, 119).

Daschkewitsch, Arkadij (2019). »Modularisierung des Renew-Plugin Systems«. Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 85, 407).

Dijkstra, Edsger W. (Nov. 1974). »Self-Stabilizing Systems in Spite of Distributed Control«. In: *Communications of the ACM* 17.11, S. 643–644. ISSN: 0001-0782. DOI: 10.1145/361179.361202 (siehe S. 36).

Donatelli, Susanna und Haar, Stefan, Hrsg. (2019). *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings*. Bd. 11522. Lecture Notes in Computer Science. Springer. ISBN: 978-3-030-21570-5. DOI: 10.1007/978-3-030-21571-2.

Douady, Adrien, Hubbard, John H., Lavaurs, P., Tan, Lei und Sentenac, Pierre (1984). »Étude dynamique des polynômes complexes«. In: *Prépublications mathémathiques d'Orsay* (siehe S. 392).

Dreschler, Leonie (1981). »Ermittlung markanter Punkte auf den Bildern bewegter Objekte und Berechnung einer 3D-Beschreibung auf dieser Grundlage«. Dissertation. D-2000 Hamburg 13 Schlüterstr. 70 (heute: Vogt-Kölln Str. 30, D-22527 Hamburg): Universität Hamburg, Fachbereich Informatik (siehe S. 63, 104).

Dreschler-Fischer, Leonie und Triendl, Ernst E. (1985). »Ein allgemeiner und modularer Ansatz zum Korrspondenzproblem«. In: *Mustererkennung 1985, DAGM-Symposium, Erlangen, 24.-26. September 1985, Proceedings*. Hrsg. von Niemann, Heinrich. Bd. 107. Informatik-Fachberichte. Springer, S. 70–74. DOI: 10.1007/978-3-642-70638-7_14 (siehe S. 104).

Duboc, Leticia, Rosenblum, David S. und Wicks, Tony (2006). »A framework for modelling and analysis of software systems scalability«. In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. Hrsg. von Osterweil, Leon J., Rombach, H. Dieter und Soffa, Mary Lou. ACM, S. 949–952. doi: 10.1145/1134285.1134460 (siehe S. 46).

Duvigneau, Michael (Dez. 2002). »Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten«. Diplomarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 6, 98, 122, 133, 134, 136, 137, 156, 160, 201, 279).

Duvigneau, Michael (Okt. 2009). »Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen«. https://ediss.sub.uni-hamburg.de/handle/ediss/3023. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik (siehe S. 75, 123, 206, 434).

Duvigneau, Michael und Moldt, Daniel, Hrsg. (Sep. 2009). *Proceedings of the Fifth International Workshop on Modeling of Objects, Components and Agents, MOCA'09, Hamburg*. Bericht FBI-HH-B-290/09. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik.

Duvigneau, Michael, Moldt, Daniel und Rölke, Heiko (Juli 2002). »Concurrent Architecture for a Multi-agent Platform«. In: *Agent-Oriented Software Engineering. 3rd International Workshop, AOSE 2002, Bologna. Proceedings*. Hrsg. von Giunchiglia, Fausto, Odell, James und Weiss, Gerhard. ACM Press, S. 147–159 (siehe S. 6, 98).

Duvigneau, Michael, Moldt, Daniel und Rölke, Heiko (2003). »Concurrent Architecture for a Multi-agent Platform«. In: *Agent-Oriented Software Engineering III. Third International Workshop, Agent-oriented Software Engineering (AOSE) 2002, Bologna, Italy, July 2002. Revised Papers and Invited Contributions*. Hrsg. von Giunchiglia, Fausto, Odell, James und Weiss, Gerhard. Bd. 2585. Lecture Notes in Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, S. 59–72. doi: 10.1007/3-540-36540-0_5 (siehe S. 7, 95).

Engelhardt, Henri (2020). »Untersuchung der Container-basierten Ausführung von Petrinetzalgorithmen und Werkzeugen am Beispiel von Renew / Renewkube«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 6, 244, 345, 403).

Farwer, Berndt und Lomazova, I. (2001). »A Systematic Approach towards Object-Based Petri Net Formalisms«. In: *Perspectives of System Informatics, Proceedings of the 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk*. Hrsg. von Bjorner, D. und Zamulin, A. Bd. 2244. Lecture Notes in Computer Science. Springer-Verlag, S. 255–267 (siehe S. 6).

Feldmann, Matthias (Nov. 2019). »Containerization of the Reference Net Workshop and Evaluation of Interprocess Communication Technologies for Containerized Multi-Agent Net Applications«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 207, 288, 456).

Fellir, Fadoua, Attar, Adnane El, Nafil, Khalid und Chung, Lawrence (2020). »A multi-Agent based model for task scheduling in cloud-fog computing platform«. In: *IEEE International Conference on Informatics, IoT, and Enabling Technologies, ICIoT 2020, Doha, Qatar, February 2-5, 2020*. IEEE, S. 377–382. doi: 10.1109/ICIoT48696.2020.9089625 (siehe S. 107).

Fielding, Roy Thomas (2000). »Architectural Styles and the Design of Network-Based Software Architectures«. Dissertation. isbn: 0599871180 (siehe S. 44, 45).

Fischer, Klaus (1999). »Holonic Multiagent Systems - Theory and Applications«. In: *Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Évora, Portugal, September 21-24, 1999, Proceedings*. Hrsg. von Barahona, Pedro und Alferes, José Júlio. Bd. 1695. Lecture Notes in Computer Science. Springer, S. 34–48. doi: 10.1007/3-540-48159-1_3 (siehe S. 159).

Foundation for Intelligent Physical Agents (FIPA) (Dez. 2002). *FIPA Abstract Architecture Specification (SC00001L)* (siehe S. 29).

Foundation for Intelligent Physical Agents (FIPA) (März 2004). *FIPA Agent Management Specification (SC00023K)* (siehe S. 30, 31).

Fowler, Martin (Jan. 2004). *Inversion of Control Containers and the Dependency Injection pattern*. url: https://martinfowler.com/articles/injection.html (besucht am 05. 03. 2021) (siehe S. 399).

Fowler, Martin und Lewis, James (März 2014). *Microservices*. url: http://martinfowler.com/articles/microservices.html (besucht am 08. 01. 2022) (siehe S. 50).

Fu, Yuqi, Zhang, Shaolun, Terrero, Jose, Mao, Ying, Liu, Guangya, Li, Sheng u. a. (2019). »Progress-based Container Scheduling for Short-lived Applications in a Kubernetes Cluster«. In: *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*. IEEE, S. 278–287. doi: 10.1109/BigData47090.2019.9006427 (siehe S. 106).

Gade, Martin, Fiedler, Gerald und Dreschler-Fischer, Leonie (2003). »Two-dimensional sea surface current fields derived from multi-sensor satellite data«. In: *2003 IEEE International Geoscience and Remote Sensing Symposium, IGARSS 2003, Toulouse, France, July 21-15, 2003*. IEEE, S. 3540–3542. doi: 10.1109/IGARSS.2003.1294847 (siehe S. 104).

Gamma, Erich, Helm, Richard, Johnson, Ralph und Vlissides, John M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. Aufl. Addison-Wesley Professional. isbn: 0201633612 (siehe S. 75).

Garcia-Molina, Hector und Salem, Kenneth (1987). »Sagas«. In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. SIGMOD '87. San Francisco, California, USA: ACM, S. 249–259. isbn: 0-89791-236-5. doi: 10.1145/38713.38742 (siehe S. 41).

Garrison, Justin und Nova, Kris (2017). *Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment*. 1st. O'Reilly Media, Inc. isbn: 1491984309 (siehe S. 54, 55, 151, 187).

Gessert, Felix (2019). »Low Latency for Cloud Data Management«. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 99).

Gessert, Felix, Wingerath, Wolfram und Ritter, Norbert (2020). *Fast and Scalable Cloud Data Management*. Springer. isbn: 978-3-030-43505-9. doi: 10.1007/978-3-030-43506-6 (siehe S. 99).

Gilbert, Seth und Lynch, Nancy (Juni 2002). »Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services«. In: *SIGACT News* 33.2, S. 51–59. issn: 0163-5700. doi: 10.1145/564585.564601 (siehe S. 39).

Girault, Claude und Valk, Rüdiger (2003). *Petri nets for systems engineering - a guide to modeling, verification, and applications*. Springer. isbn: 978-3-540-41217-5 (siehe S. 16–19).

Glake, Daniel, Kiehn, Felix, Schmidt, Mareike und Ritter, Norbert (2021). »Towards Taming the Adaptivity Problem - Formalizing Poly-/MultiStore Topology Descriptions«. In: *Service-Oriented Computing - 15th Symposium and Summer School, SummerSOC 2021, Virtual Event, September 13-17, 2021, Proceedings*. Hrsg. von Barzen, Johanna. Bd. 1429. Communications in Computer and Information Science. Springer, S. 83–99. doi: 10.1007/978-3-030-87568-8_5 (siehe S. 100).

Glake, Daniel, Panse, Fabian, Ritter, Norbert, Clemen, Thomas und Lenfers, Ulfia (2021). »Data Management in Multi-Agent Simulation Systems«. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*. Hrsg. von Sattler, Kai-Uwe, Herschel, Melanie und Lehner, Wolfgang. Bd. P-311. LNI. Gesellschaft für Informatik, Bonn, S. 423–436. doi: 10.18420/btw2021-22 (siehe S. 100).

Glake, Daniel, Ritter, Norbert und Clemen, Thomas (2020). »Utilizing Spatio-Temporal Data in Multi-Agent Simulation«. In: *Winter Simulation Conference, WSC 2020, Orlando, FL, USA, December 14-18, 2020*. IEEE, S. 242–253. doi: 10.1109/WSC48552.2020.9384124 (siehe S. 100).

Glake, Daniel, Weyl, Julius, Dohmen, Carolin, Hüning, Christian und Clemen, Thomas (2017). »Modeling through model transformation with MARS 2.0«. In: *Proceedings of the Agent-Directed Simulation Symposium, Spring Simulation Multi-Conference, SpringSim (ADS) 2017, Virginia Beach, VA, USA, April 23 - 26, 2017*. Hrsg. von Zhang, Yu und Madey, Gregory R. Society for Computer Simulation International / ACM, 2:1–2:12 (siehe S. 100).

Gracia, Víctor Medel, Arronategui, Unai, Bañares, José Ángel, Tolosana, Rafael und Rana, Omer (2019). »Modeling, Characterising and Scheduling Applications in Kubernetes«. In: *Economics of Grids, Clouds, Systems, and Services - 16th International Conference, GECON 2019, Leeds, UK, September 17-19, 2019, Proceedings*. Hrsg. von Djemame, Karim, Altmann, Jörn, Bañares, José Ángel, Ben-Yehuda, Orna Agmon und Naldi, Maurizio. Bd. 11819. Lecture Notes in Computer Science. Springer, S. 291–294. doi: 10.1007/978-3-030-36027-6_26 (siehe S. 93).

Gracia, Víctor Medel, Rana, Omer F., Bañares, José Ángel und Arronategui, Unai (2016). »Modelling performance & resource management in kubernetes«. In: *Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC 2016, Shanghai, China, December 6-9, 2016*. Hrsg. von Jiang, Changjun, Rana, Omer F. und Antonopoulos, Nick. ACM, S. 257–262. doi: 10.1145/2996890.3007869 (siehe S. 93).

Grama, Ananth, Gupta, Anshul und Kumar, Vipin (1993). »Isoefficiency: measuring the scalability of parallel algorithms and architectures«. In: *IEEE Parallel & Distributed Technology: Systems & Applications* 1.3, S. 12–21. doi: 10.1109/88.242438 (siehe S. 46).

Gray, Jim und Reuter, Andreas (1992). *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. isbn: 1558601902 (siehe S. 40).

Griffel, Frank (1998). *Componentware - Konzepte und Techniken eines Softwareparadigmas*. dpunkt. isbn: 978-3-932588-02-0 (siehe S. 75).

Guerrero, Carlos, Lera, Isaac und Juiz, Carlos (2018). »Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture«. In: *Journal of Grid Computing* 16.1, S. 113–135. doi: 10.1007/s10723-017-9419-x (siehe S. 174).

Gusella, R. und Zatti, S. (1989). »The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD«. In: *IEEE Transactions on Software Engineering* 15.7, S. 847–853. doi: 10.1109/32.29484 (siehe S. 302).

Happ, Sonja, Dähling, Stefan und Monti, Antonello (2020). »Scalable assessment method for agent-based control in cyber-physical distribution grids«. In: *IET Cyper-Physical Systems: Theory & Applications* 5.3, S. 283–291. doi: 10.1049/iet-cps.2019.0096 (siehe S. 90, 119).

Härder, Theo und Reuter, Andreas (Dez. 1983). »Principles of Transaction-Oriented Database Recovery«. In: *ACM Computing Surveys* 15.4, S. 287–317. issn: 0360-0300. doi: 10.1145/289.291 (siehe S. 40).

Harris, Christopher G. und Stephens, Mike (1988). »A Combined Corner and Edge Detector«. In: *Proceedings of the Alvey Vision Conference, AVC 1988, Manchester, UK, September, 1988*. Hrsg. von Taylor, Christopher J. Alvey Vision Club, S. 1–6. doi: 10.5244/C.2.23 (siehe S. 63).

Hauschildt, Dirk (1987). *A Petri Net Implementation*. Fachbereichsmitteilung FBI-HH-M-145/87. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 101).

Heitmann, Frank (Aug. 2013). »Algorithms and Hardness Results for Object Nets«. eng. Diss. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 6).

Hess, Rob (Okt. 2010). »An open-source SIFTLibrary«. In: *Proceedings of the 18th International Conference on Multimedia, Firenze, Italy*. Hrsg. von Bimbo, Alberto Del, Chang, Shih-Fu und Smeulders, Arnold W. M. ACM, S. 1493–1496. doi: 10.1145/1873951.1874256 (siehe S. 413).

Hewelt, Marcin (2010). »Grundlegende Konstrukte einer einheitentheoretischen Modellierungstechnik«. Diplomarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik (siehe S. 45).

Hill, Mark D. (1990). »What is scalability?« In: *ACM SIGARCH Computer Architecture News* 18.4, S. 18–21. doi: 10.1145/121973.121975 (siehe S. 46).

Hsieh, Fu-Shiung (2018). »Design of scalable agent-based reconfigurable manufacturing systems with Petri nets«. In: *International Journal of Computer Integrated Manufacturing* 31.8, S. 748–759. doi: 10.1080/0951192X.2018.1429665 (siehe S. 90).

Jacob, Thomas (2001). »Anbindung eines Petrinetz-Simulators an eine Datenbank«. Studienarbeit. Universität Hamburg, Fachbereich Informatik (siehe S. 76, 205, 307).

Jacob, Thomas (2002). »Implementierung einer sicheren und rollenbasierten Workflow-Managementkomponente für ein Petrinetzwerkzeug«. Diplomarbeit.

Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 24, 127, 135, 136, 319).

Jacob, Thomas, Kummer, Olaf, Moldt, Daniel und Ultes-Nitsche, Ulrich (Aug. 2002). »Implementation of Workflow Systems using Reference Nets – Security and Operability Aspects«. In: *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. Hrsg. von Jensen, Kurt. DAIMI PB: Aarhus, Denmark, August 28–30, number 560. Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark: University of Aarhus, Department of Computer Science (siehe S. 83).

Janneck, Jan Robert (März 2021). »Modularizing a Plugin System Using Java Modules: Application to a Medium-Sized Open-Source Project«. Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 85, 95, 206, 284, 407, 424, 432).

Jensen, Kurt (2001). »Special Section on the Practical Use of High-Level Petri Nets: Preface by the section editor«. In: *International Journal on Software Tools for Technology Transfer* 3.4, S. 369–371. doi: 10.1007/s100090100066 (siehe S. 27).

Jensen, Kurt und Rozenberg, Grzegorz, Hrsg. (1991). *High-level Petri Nets: Theory and Applications*. Englisch. Berlin: Springer-Verlag. isbn: 978-3-540-54125-7 (siehe S. 27).

Jin, Hongyu, Wang, Yang, Wang, Qian, Liu, Jiankang, Wang, Shuhua, Zhang, Jun u. a. (2019). »Architecture Modelling and Task Scheduling of an Integrated Parallel CNC System in Docker Containers Based on Colored Petri Nets«. In: *IEEE Access* 7, S. 47535–47549. doi: 10.1109/ACCESS.2019.2909774 (siehe S. 106).

Jürgensen, Rainer (Aug. 2021). »Untersuchung des Zertifikateeinsatzes im Java-Umfeld – Beispielhafte Diskussion von Zertifikaten für die Open-Source Software Renew«. Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 279, 284, 431, 454).

Kaewkasi, C. und Chuenmuneewong, K. (2017). »Improvement of container scheduling for Docker using Ant Colony Optimization«. In: *2017 9th International Conference on Knowledge and Smart Technology (KST)*, S. 254–259. doi: 10.1109/KST.2017.7886112 (siehe S. 174).

Kang, Dong-Jae, Kim, Young-Ho, Cha, Gyu-Il, Jung, Sung-In, Kim, Myung-Joon und Bae, Hae-Young (2006). »Design and Implementation of Zero-Copy Data Path for Efficient File Transmission«. In: *High Performance Computing and Communications, Second International Conference, HPCC 2006, Munich, Germany, September 13-15, 2006, Proceedings*. Hrsg. von Gerndt, Michael

und Kranzlmüller, Dieter. Bd. 4208. Lecture Notes in Computer Science. Springer, S. 350–359. doi: 10.1007/11847366_36 (siehe S. 239).

Kay, Alan C. (März 1993). »The Early History of Smalltalk«. In: *SIGPLAN Not.* 28.3, S. 69–95. issn: 0362-1340. doi: 10.1145/155360.155364 (siehe S. 128).

Kemper, Alfons und Eickler, André (2011). *Datenbanksysteme - Eine Einführung, 8. Auflage*. Oldenbourg. isbn: 978-3-486-59834-6 (siehe S. 40).

Khalidi, Yousef A. und Thadani, Moti N. (1995). *An Efficient Zero-Copy I/O Framework for UNIX*. Techn. Ber. USA (siehe S. 239).

Kilian, Tim (2019). »Entwicklung einer modularen JavaScript-Bibliothek zur Modellerstellung. Unter besonderer Berücksichtigung eines Entwurfs einer Plugin-Architektur zur Metamodellierung«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 245).

Kim, Byeoungdo, Park, SeongHyeon, Lee, Seokhwan, Khoshimjonov, Elbek, Kum, Dongsuk, Kim, Junsoo u. a. (2021). »LaPred: Lane-Aware Prediction of Multi-Modal Future Trajectories of Dynamic Agents«. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, S. 14636–14645 (siehe S. 107).

Knight, Kevin (1989). »Unification: A Multidisciplinary Survey«. In: *ACM Computing Surveys* 21.1, S. 93–124. doi: 10.1145/62029.62030 (siehe S. 135).

Köhler, Michael (2004). *Objektnetze: Definition und Eigenschaften*. Hrsg. von Moldt, Daniel. Bd. 1. Agent Technology – Theory and Applications. Berlin: Logos Verlag, S. 254. isbn: 978-3-8325-0695-7 (siehe S. 6, 102).

Köhler, Michael (2007). »A Formal Model of Multi-Agent Organisations«. In: *Fundamenta Informaticae* 79.3-4, S. 415–430 (siehe S. 102).

Köhler, Michael und Farwer, Berndt (2007). »Object Nets for Mobility«. In: *International Conference on Application and Theory of Petri Nets 2007*. Hrsg. von Kleijn, J. und Yakovlev, A. Bd. 4546. Lecture Notes in Computer Science. Springer-Verlag, S. 244–262 (siehe S. 128).

Köhler, Michael, Moldt, Daniel und Rölke, Heiko (2001). »Modelling the Structure and Behaviour of Petri Net Agents«. In: *Proceedings of the 22nd Conference on Application and Theory of Petri Nets 2001*. Hrsg. von Colom, J.M. und Koutny, M. Bd. 2075. Lecture Notes in Computer Science. Springer-Verlag, S. 224–241 (siehe S. 95–97, 102).

Köhler, Michael, Moldt, Daniel und Rölke, Heiko (2003). »Modelling Mobility and Mobile Agents Using Nets within Nets«. In: *Proceedings of the 24th International Conference on Application and Theory of Petri Nets 2003 (ICATPN 2003)*. Hrsg. von Aalst, Wil van der und Best, Eike. Bd. 2679. Lecture Notes in Computer Science. Springer-Verlag, S. 121–139 (siehe S. 128).

Köhler, Michael und Rölke, Heiko (Aug. 2002). »Modelling mobility and mobile agents using nets within nets«. In: *Proceedings of the Second Workshop on Modelling of Objects, Components, and Agents (MOCA'02)*. Hrsg. von Moldt, Daniel. DAIMI PB: Aarhus, Denmark, August 26–27, number 561. Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark: University of Aarhus, Department of Computer Science, S. 141–157 (siehe S. 128).

Köhler-Bussmeier, Michael (2009). »Koordinierte Selbstorganisation und selbstorganisierte Koordination: Eine formale Spezifikation reflexiver Selbstorganisation in Multiagentensystemen unter spezieller Berücksichtigung der sozialwissenschaftlichen Perspektive«. Habilitationsschrift. University of Hamburg (siehe S. 102).

Köhler-Bussmeier, Michael (2014). »A Survey on Decidability Results for Elementary Object Systems«. In: *Fundamenta Informaticae* 130.1, S. 99–123 (siehe S. 102).

Köhler-Bussmeier, Michael und Wester-Ebbinghaus, Matthias (Sep. 2009a). »A Petri Net based Prototype for MAS Organisation Middleware«. In: *Proceedings of the Fifth International Workshop on Modeling of Objects, Components and Agents, MOCA'09, Hamburg*. Hrsg. von Duvigneau, Michael und Moldt, Daniel. Bericht FBI-HH-B-290/09. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik, S. 29–44 (siehe S. 103).

Köhler-Bussmeier, Michael und Wester-Ebbinghaus, Matthias (Sep. 2009b). »SONAR*: A Multi-Agent Infrastructure for Active Application Architectures and Inter-Organisational Information Systems«. In: *Multiagent System Technologies. 7th German Conference, MATES 2009, Hamburg, Germany, September 9-11, 2009. Proceedings*. Hrsg. von Braubach, Lars, Hoek, Wiebke van der, Petta, Paolo und Pokahr, Alexander. Bd. 5774. Lecture Notes in Artificial Intelligence. Berlin, Heidelberg, New York: Springer-Verlag, S. 248–257. ISBN: 978-3-642-04142-6 (siehe S. 102).

Köhler-Bussmeier, Michael, Kindler, Ekkart und Rölke, Heiko, Hrsg. (2021). *Proceedings of the International Workshop on Petri Nets and Software Engineering 2021 co-located with the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021), Paris, France, June 25th, 2021 (due to COVID-19: virtual conference)*. Bd. 2907. CEUR Workshop Proceedings. CEUR-WS.org.

Krön, Valentin (Jan. 2021). »Entwicklung einer Plugin-Erweiterung für das Werkzeug Renew zur Visualisierung der Entfaltung einzelner Prozesse im Rahmen der Ausführung einfacher Petrinetze«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 205, 302).

KUMMER, Olaf (1996). *Axiomensysteme für die Theorie der Nebenläufigkeit*. Berlin: Logos Verlag, S. 1–164. ISBN: 3-931216-28-4 (siehe S. 25).

KUMMER, Olaf (1998). »Simulating Synchronous Channels and Net Instances«. In: *Forschungsbericht Nr. 694: 5. Workshop Algorithmen und Werkzeuge für Petrinetze*. Hrsg. von DESEL, Jörg, KEMPER, Peter, KINDLER, Ekkart und OBERWEIS, Andreas. Forschungsbericht Nr. 694. Fachbereich Informatik, Universität Dortmund, S. 73–78 (siehe S. 25).

KUMMER, Olaf (1999). »A Petri Net View on Synchronous Channels«. In: *Petri Net Newsletter* 56. Hrsg. von DESEL, J., STARKE, P. und VALK, Rüdiger, S. 7–11 (siehe S. 25).

KUMMER, Olaf (2000). »Undecidability in Object-Oriented Petri Nets«. In: *Petri Net Newsletter* 59, S. 18–23 (siehe S. 25, 101).

KUMMER, Olaf (2001). »Introduction to Petri Nets and Reference Nets«. In: *Sozionik Aktuell* 1. ISSN 1617-2477, S. 1–9 (siehe S. 101).

KUMMER, Olaf (2002). *Referenznetze*. Berlin: Logos Verlag. ISBN: 978-3-8325-0035-1 (siehe S. 5, 6, 25–27, 64, 68, 72, 73, 101, 350, 351, 357).

KUMMER, Olaf, MOLDT, Daniel und WIENBERG, Frank (1998). »A Framework for interacting Design/CPN- and Java Processes«. In: *Workshop on Practical Use of Coloured Petri Nets and Design/CPN 1998*. Hrsg. von JENSEN, Kurt. Bd. Daimi PB-532. Aarhus University, S. 131–150 (siehe S. 101).

KUMMER, Olaf, MOLDT, Daniel und WIENBERG, Frank (Juni 1999). »Symmetric Communication between Coloured Petri Net Simulations and Java-Processes«. In: *Application and Theory of Petri Nets 1999, 20th International Conference, ICATPN'99, Williamsburg, Virginia, USA*. Hrsg. von DONATELLI, Susanna und KLEIJN, Jetty. Bd. 1639. Lecture Notes in Computer Science. Springer-Verlag, S. 86–105 (siehe S. 101).

KUMMER, Olaf und WIENBERG, Frank (März 1999a). *Renew – The Reference Net Workshop*. Available at: http://www.renew.de/. Release 1.0 (siehe S. 101).

KUMMER, Olaf und WIENBERG, Frank (1999b). »Renew – the Reference Net Workshop«. In: *Petri Net Newsletter* 56, S. 12–16 (siehe S. 6).

KUMMER, Olaf, WIENBERG, Frank, DUVIGNEAU, Michael, CABAC, Lawrence, HAUSTERMANN, Michael und MOSTELLER, David (Juni 2016). *Renew – The Reference Net Workshop*. Release 2.5 (siehe S. 67).

KUMMER, Olaf, WIENBERG, Frank, DUVIGNEAU, Michael, CABAC, Lawrence, HAUSTERMANN, Michael und MOSTELLER, David (Nov. 2020a). *Renew – The Reference Net Workshop*. Release 2.5.1 (siehe S. 6).

KUMMER, Olaf, WIENBERG, Frank, DUVIGNEAU, Michael, CABAC, Lawrence, HAUSTERMANN, Michael und MOSTELLER, David (Nov. 2020b). *Renew – User*

Guide (Release 2.5.1). Release 2.5.1. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group. Hamburg (siehe S. 101, 413).

Kummer, Olaf, Wienberg, Frank, Duvigneau, Michael, Köhler, Michael, Moldt, Daniel und Rölke, Heiko (Juni 2003). »Renew – The Reference Net Workshop«. In: *Tool Demonstrations. 24th International Conference on Application and Theory of Petri Nets (ATPN 2003). International Conference on Business Process Management (BPM 2003)*. Hrsg. von Veerbeek, Eric. Department of Technology Management, Technische Universiteit Eindhoven. Beta Research School for Operations Management und Logistics, S. 99–102 (siehe S. 84, 202).

Kummer, Olaf, Wienberg, Frank, Duvigneau, Michael, Schumacher, Jörn, Köhler, Michael, Moldt, Daniel u. a. (Juni 2004). »An Extensible Editor and Simulation Engine for Petri Nets: Renew«. In: *Applications and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings*. Hrsg. von Cortadella, Jordi und Reisig, Wolfgang. Bd. 3099. Lecture Notes in Computer Science. Berlin, Heidelberg, New York: Springer, S. 484–493. doi: 10.1007/978-3-540-27793-4_29 (siehe S. 101).

Laka, Wojciech (Dez. 2007). »Ausbau einer Infrastruktur für offene agentenorientierte Anwendungen im Kontext von Capa und OpenNet«. Diplomarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik (siehe S. 98).

Lakos, Charles (1995). »From Coloured Petri Nets to Object Petri Nets«. In: *Application and Theory of Petri Nets 1995, 16th International Conference, Turin, Italy, June 26-30, 1995, Proceedings*. Hrsg. von Michelis, Giorgio De und Diaz, Michel. Bd. 935. Lecture Notes in Computer Science. Springer, S. 278–297. doi: 10.1007/3-540-60029-9_45 (siehe S. 6).

Lamport, Leslie (1978). »Time, Clocks, and the Ordering of Events in a Distributed System«. In: *Communications of the ACM* 21.7, S. 558–565. doi: 10.1145/359545.359563 (siehe S. 36, 302).

Lanese, Ivan und Zavattaro, Gianluigi (2009). »Programming Sagas in SOCK«. In: *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*. Hrsg. von Hung, Dang Van und Krishnan, Padmanabhan. IEEE Computer Society, S. 189–198. doi: 10.1109/SEFM.2009.23 (siehe S. 145).

Legendre, Adrien-Marie (1830). *Théorie des nombres*. Nineteenth Century Collections Online (NCCO): Science, Technology, and Medicine: 1780-1925 Bd. 1. Firmin Didot frères (siehe S. 371).

Lenzen, Christoph und Rybicki, Joel (2019). »Self-Stabilising Byzantine Clock Synchronisation Is Almost as Easy as Consensus«. In: *J. ACM* 66.5, 32:1–32:56. doi: 10.1145/3339471 (siehe S. 302).

Lhaksmana, Kemas Muslim, Murakami, Yohei und Ishida, Toru (2018). »Role-Based Modeling for Designing Agent Behavior in Self-Organizing Multi-Agent Systems«. In: *International Journal of Software Engineering and Knowledge Engineering* 28.1, S. 79–96. doi: 10.1142/S0218194018500043 (siehe S. 92).

Lilienthal, Carola (2008). »Komplexität von Softwarearchitekturen: Stile und Strategien«. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik (siehe S. 94).

Lin, Donghui, Murakami, Yohei und Ishida, Toru (2018). »Integrating Internet of Services and Internet of Things from a Multiagent Perspective«. In: *Massively Multi-Agent Systems II - International Workshop, MMAS 2018, Stockholm, Sweden, July 14, 2018, Revised Selected Papers*. Hrsg. von Lin, Donghui, Ishida, Toru, Zambonelli, Franco und Noda, Itsuki. Bd. 11422. Lecture Notes in Computer Science. Springer, S. 36–49. doi: 10.1007/978-3-030-20937-7_3 (siehe S. 92).

Lin, Xiangru, Li, Guanbin und Yu, Yizhou (2021). »Scene-Intuitive Agent for Remote Embodied Visual Grounding«. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, S. 7036–7045 (siehe S. 107).

Lomazova, Irina und Ermakova, Vera O. (2016). »Verification of Nested Petri Nets Using an Unfolding Approach«. In: *Proceedings of the International Workshop on Petri Nets and Software Engineering 2016, including the International Workshop on Biological Processes & Petri Nets 2016 co-located with the 37th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2016 and the 16th International Conference on Application of Concurrency to System Design ACSD 2016, Toruń, Poland, June 20-21, 2016*. Hrsg. von Cabac, Lawrence, Kristensen, Lars Michael und Rölke, Heiko. Bd. 1591. CEUR Workshop Proceedings. CEUR-WS.org, S. 93–112 (siehe S. 6).

Lorenz, Robert und Metzger, Johannes, Hrsg. (2018). *Algorithms and Tools for Petri Nets, Proceedings of the 21th Workshop AWPN 2018, Augsburg, Germany*. Reports / Technische Berichte der Fakultät für Angewandte Informatik der Universität Augsburg 2018-02, S. 64.

Lowe, David G. (1999). »Object Recognition from Local Scale-Invariant Features«. In: *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*. ICCV '99. Washington, DC, USA: IEEE Computer Society, S. 1150–. isbn: 0-7695-0164-8 (siehe S. 368).

Lowe, David G. (Nov. 2004). »Distinctive Image Features from Scale-Invariant Keypoints«. In: *International Journal on Computer Vision* 60.2, S. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94 (siehe S. 368).

Maier, Christoph und Moldt, Daniel (Sep. 1997). »Object Coloured Petri Nets – A Formal Technique for Object Oriented Modelling«. In: *Proceedings of the Workshop on Petri Nets in System Engineering (PNSE'97), Hamburg, September 25-26, 1997*. Hrsg. von Farwer, Berndt, Moldt, Daniel und Stehr, Mark-Oliver. Bericht des Fachbereichs Informatik FBI-HH-B-205/97. Universität Hamburg, Fachbereich Informatik, S. 11–19 (siehe S. 101).

Maier, Christoph und Moldt, Daniel (2001). »Object Coloured Petri Nets – A Formal Technique for Object Oriented Modelling«. In: *Advances in Petri Nets: Concurrent Object-Oriented Programming and Petri Nets*. Hrsg. von Agha, Gul, De Cindio, Fiorella und Rozenberg, Grzegorz. Bd. 2001. Lecture Notes in Computer Science. Springer-Verlag, S. 406–427 (siehe S. 24, 101).

Markwardt, Kolja (2013). »Strukturierung petrinetzbasierter Multiagentenanwendungen am Beispiel verteilter Softwareentwicklungsprozesse«. Diss. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 434).

Marsso, Lina (2019). »On Model-based Testing of GALS Systems. (Etude de génération de tests à partir d'un modèle pour les systèmes GALS)«. Dissertation. Grenoble Alpes University, France (siehe S. 105).

Marukatat, Sanparith und Methasate, Ithipan (2013). »Fast nearest neighbor retrieval using randomized binary codes and approximate Euclidean distance«. In: *Pattern Recognition Letters* 34.9, S. 1101–1107. DOI: 10.1016/j.patrec.2013.03.006 (siehe S. 368).

Mills, David L. (1985). »Network Time Protocol (NTP)«. In: *RFC* 958, S. 1–14. DOI: 10.17487/RFC0958 (siehe S. 302).

Milner, Robin (1999). *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press. ISBN: 978-0-521-65869-0 (siehe S. 128).

Milner, Robin, Parrow, Joachim und Walker, David (1992a). »A Calculus of Mobile Processes, I«. In: *Inf. Comput.* 100.1, S. 1–40. DOI: 10.1016/0890-5401(92)90008-4 (siehe S. 128).

Milner, Robin, Parrow, Joachim und Walker, David (1992b). »A Calculus of Mobile Processes, II«. In: *Inf. Comput.* 100.1, S. 41–77. DOI: 10.1016/0890-5401(92)90009-5 (siehe S. 128).

Moldt, Daniel (Aug. 1996). »Höhere Petrinetze als Grundlage für Systemspezifikationen«. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 6, 24, 101).

Moldt, Daniel (Aug. 2005). »Petrinetze als Denkzeug«. In: *Object Petri Nets, Processes, and Object Calculi*. Hrsg. von Farwer, Berndt und Moldt, Daniel. Bericht des Fachbereichs Informatik FBI-HH-B-265/05. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik, S. 51–70 (siehe S. 6, 32, 45, 102).

Moldt, Daniel, Offermann, Sven und Ortmann, Jan (2005). »A Petri Net-Based Architecture for Web Services«. In: *Workshop on Service-Oriented Computing and Agent-Based Engineering, SOCABE 2005, Utrecht, Netherland, July 26, 2005. Proceedings*. Hrsg. von Cavedon, Lawrence, Kowalczyk, Ryszard, Maamar, Zakaria, Martin, David und Müller, Ingo, S. 33–40 (siehe S. 102).

Moldt, Daniel, Ortmann, Jan und Offermann, Sven (2004). »A Proposal for Petri Net Based Web Service Application Modeling«. In: *Web Engineering - 4th International Conference, ICWE 2004, Munich, Germany, July 26-30, 2004, Proceedings*. Hrsg. von Koch, Nora, Fraternali, Piero und Wirsing, Martin. Bd. 3140. Lecture Notes in Computer Science. Springer-Verlag, S. 93–97. isbn: 3-540-22511-0 (siehe S. 102, 128, 129, 152, 218).

Moldt, Daniel, Quenum, José, Reese, Christine und Wagner, Thomas (Juni 2010). »Improving a Workflow Management System with an Agent Flavour«. In: *Proceedings of the International Workshop on Petri Nets and Software Engineering, PNSE'10, Braga, Portugal*. Hrsg. von Duvigneau, Michael und Moldt, Daniel. Bericht FBI-HH-B-294/10. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik, S. 55–70. isbn: 978-972-8692-55-1 (siehe S. 102).

Moldt, Daniel, Röwekamp, Jan Henrik und Simon, Michael (2017). »A Simple Prototype of Distributed Execution of Reference Nets Based on Virtual Machines«. In: *Algorithms and Tools for Petri Nets Proceedings of the Workshop AWPN 2017, Kgs. Lyngby, Denmark October 19-20, 2017*. Hrsg. von Bergenthum, Robin und Kindler, Ekkart. DTU Compute Technical Report 2017-06, S. 51–57 (siehe S. 214, 257, 387, 461).

Moldt, Daniel und Wienberg, Frank (1997). »Multi-Agent-Systems based on Coloured Petri Nets«. In: *Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings*. Hrsg. von Azéma, Pierre und Balbo, Gianfranco. Lecture Notes in Computer Science 1248. Berlin, Heidelberg, New York: Springer Verlag, S. 82–101 (siehe S. 102).

Moravec, Hans P. (1981). »Rover Visual Obstacle Avoidance«. In: *Proceedings of the 7th International Joint Conference on Artificial Intelligence, IJCAI '81, Vancouver, BC, Canada, August 24-28, 1981*. Hrsg. von Hayes, Patrick J. William Kaufmann, S. 785–790 (siehe S. 63).

Mosteller, David, Cabac, Lawrence und Haustermann, Michael (2016). »Integrating Petri Net Semantics in a Model-Driven Approach: The Renew Meta-Modeling and Transformation Framework«. In: *Transaction on Petri Nets and Other Models of Concurrency XI*. Lecture Notes in Computer Science 11. Hrsg. von Koutny, Maciej, Desel, Jörg und Kleijn, Jetty, S. 92–113. doi: 10.1007/978-3-662-53401-4_5 (siehe S. 328).

Murakami, Yohei, Nakaguchi, Takao, Lin, Donghui und Ishida, Toru (2018). »Two-Layer Architecture for Distributed Massively Multi-agent Systems«. In: *Massively Multi-Agent Systems II - International Workshop, MMAS 2018, Stockholm, Sweden, July 14, 2018, Revised Selected Papers*. Hrsg. von Lin, Donghui, Ishida, Toru, Zambonelli, Franco und Noda, Itsuki. Bd. 11422. Lecture Notes in Computer Science. Springer, S. 53–65. doi: 10.1007/978-3-030-20937-7_4 (siehe S. 91, 118).

Mutlag, Ammar Awad, Ghani, Mohd Khanapi Abd, Mohammed, Mazin Abed, Lakhan, Abdullah, Mohd, Othman, Abdulkareem, Karrar Hameed u. a. (2021). »Multi-Agent Systems in Fog-Cloud Computing for Critical Healthcare Task Management Model (CHTM) Used for ECG Monitoring«. In: *Sensors* 21.20, S. 6923. doi: 10.3390/s21206923 (siehe S. 107).

Nguyen, Thanh-Tung, Yeom, Yu-Jin, Kim, Taehong, Park, Dae-Heon und Kim, Sehan (2020). »Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration«. In: *Sensors* 20.16, S. 4621. doi: 10.3390/s20164621 (siehe S. 174).

Nicol, David M. (1998). »Scalability, Locality, Partitioning and Synchronization PDES«. In: *Proceedings of the 12th Workshop on Parallel and Distributed Simulation, PADS '98, Banff, Alberta, Canada, May 26-29, 1998*. Hrsg. von Unger, Brian W. und Ferscha, Alois. IEEE Computer Society, S. 5–11. doi: 10.1109/PADS.1998.685264 (siehe S. 46, 47).

Oberkampf, William L. und Roy, Christopher J. (2010). *Verification and Validation in Scientific Computing*. 1st. USA: Cambridge University Press. isbn: 0521113601 (siehe S. 15).

Parnas, David Lorge (1971). »Information Distribution Aspects of Design Methodology«. In: *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28, 1971*. Hrsg. von Freiman, Charles V., Griffith, John E. und Rosenfeld, Jack L. North-Holland, S. 339–344 (siehe S. 35).

Pawlaszczyk, Dirk (2009). »Skalierbare agentenbasierte Simulation: Werkzeuge und Techniken zur verteilten Ausführung agentenbasierter Modelle«. Dissertation. Technische Universität Ilmenau, Germany. isbn: 978-3-939473-59-6 (siehe S. 91).

Pawlaszczyk, Dirk und Strassburger, Steffen (2009). »Scalability in Distributed Simulations of Agent-based Models«. In: *Proceedings of the 2009 Winter Simulation Conference, WSC 2009, Hilton Austin Hotel, Austin, TX, USA, December 13-16, 2009*. Hrsg. von Dunkin, Ann, Ingalls, Ricki G., Yücesan, Enver, Rossetti, Manuel D., Hill, Ray und Johansson, Björn. IEEE, S. 1189–1200. doi: 10.1109/WSC.2009.5429429 (siehe S. 91).

Pawlaszczyk, Dirk und Timm, Ingo J. (2006). »A Hybrid Time Management Approach to Agent-Based Simulation«. In: *KI 2006: Advances in Artificial Intelligence, 29th Annual German Conference on AI, KI 2006, Bremen, Germany, June 14-17, 2006, Proceedings*. Hrsg. von Freksa, Christian, Kohlhase, Michael und Schill, Kerstin. Bd. 4314. Lecture Notes in Computer Science. Springer, S. 374–388. doi: 10.1007/978-3-540-69912-5_28 (siehe S. 91).

Petri, Carl Adam (1962). *Kommunikation mit Automaten*. Dissertation, Schriften des IIM 2. Bonn: Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn (siehe S. 6, 13, 15, 36).

Petri, Carl Adam (1977). »General Net Theory«. In: *Computing System Design: Proc. of the Joint IBM University of Newcastle upon Tyne Seminar, Sep., 1976*. Hrsg. von Shaw, B. University of Newcastle upon Tyne, S. 131–169 (siehe S. 15).

Petri, Carl Adam (1987). »Concurrency Theory«. In: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986*. Hrsg. von Brauer, W., Reisig, W. und Rozenberg, G. Bd. 254. Lecture Notes in Computer Science. NewsletterInfo: 27. Springer-Verlag, S. 4–24 (siehe S. 15).

Petsch, Mathias, Pawlaszczyk, Dirk und Schorcht, Hagen (2007). »Regelbasierte Koordinierung von agentengestützten Transportprozessen«. In: *eOrganisation: Service-, Prozess-, Market-Engineering: 8. Internationale Tagung Wirtschaftsinformatik - Band 2, WI 2007, Karlsruhe, Germany, February 28 - March 2, 2007*. Hrsg. von Oberweis, Andreas, Weinhardt, Christof, Gimpel, Henner, Koschmider, Agnes, Pankratius, Victor und Schnizler, Björn. Universitaetsverlag Karlsruhe, S. 355–372 (siehe S. 91).

Pokahr, Alexander (2007). »Programmiersprachen und Werkzeuge zur Entwicklung verteilter agentenorientierter Softwaresysteme«. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik. isbn: 978-3-00-023105-6 (siehe S. 99).

Pokahr, Alexander und Braubach, Lars (2009). »From a Research to an Industry-Strength Agent Platform: JADEX V2«. In: *Business Services: Konzepte, Technologien, Anwendungen. 9. Internationale Tagung Wirtschaftsinformatik 25.-27. Februar 2009, Wien*. Hrsg. von Hansen, Hans Robert, Kara-

GIANNIS, Dimitris und FILL, Hans-Georg. Bd. 246. Österreichische Computer Gesellschaft, S. 769–780 (siehe S. 99).

POKAHR, Alexander, BRAUBACH, Lars und LAMERSDORF, Winfried (2005). »Jadex: A BDI Reasoning Engine«. In: *Multi-Agent Programming: Languages, Platforms and Applications*. Hrsg. von BORDINI, Rafael H., DASTANI, Mehdi, DIX, Jürgen und SEGHROUCHNI, Amal El Fallah. Bd. 15. Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, S. 149–174 (siehe S. 99).

PRITCHETT, Dan (Mai 2008). »BASE: An Acid Alternative: In Partitioned Databases, Trading Some Consistency for Availability Can Lead to Dramatic Improvements in Scalability.« In: *Queue* 6.3, S. 48–55. ISSN: 1542-7730. DOI: 10.1145/1394127.1394128 (siehe S. 41).

RANGISETTI, Anil Kumar, DWIVEDI, Rishabh und SINGH, Prabhdeep (2021). »Denial of ARP spoofing in SDN and NFV enabled cloud-fog-edge platforms«. In: *Cluster Computing* 24.4, S. 3147–3172. DOI: 10.1007/s10586-021-03328-x (siehe S. 107).

RAO, Anand S. und GEORGEFF, Michael P. (1991). »Modeling Rational Agents within a BDI-Architecture«. In: *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*. KR'91. Cambridge, MA, USA: Morgan Kaufmann Publishers Inc., S. 473–484. ISBN: 1558601651 (siehe S. 5).

RATTIHALLI, Gourav, GOVINDARAJU, Madhusudhan, LU, Hui und TIWARI, Devesh (2019). »Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes«. In: *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*. Hrsg. von BERTINO, Elisa, CHANG, Carl K., CHEN, Peter, DAMIANI, Ernesto, GOUL, Michael und OYAMA, Katsunori. IEEE, S. 33–40. DOI: 10.1109/CLOUD.2019.00018 (siehe S. 106, 189, 214).

RATZER, Anne Vinter, WELLS, Lisa, LASSEN, Henry Michael, LAURSEN, Mads, QVORTRUP, Jacob Frank, STISSING, Martin Stig u. a. (2003). »CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets«. In: *Applications and Theory of Petri Nets 2003*. Hrsg. von AALST, Wil VAN DER und BEST, Eike. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 450–462. ISBN: 978-3-540-44919-5 (siehe S. 317).

REESE, Christine (2003). »Multiagentensysteme: Anbindung der petrinetzbasierten Plattform CAPA an das internationale Netzwerk Agentcities«. Diplomarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 98).

Reese, Christine (2009). »Prozess-Infrastruktur für Agentenanwendungen«. https://ediss.sub.uni-hamburg.de/handle/ediss/2922. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 5, 94, 102, 127, 135, 136, 150, 279, 451).

Reese, Christine, Markwardt, Kolja, Offermann, Sven und Moldt, Daniel (2006). »Distributed Business Processes in Open Agent Environments«. In: *ICEIS 2006 - Proceedings of the Eighth International Conference on Enterprise Information Systems: Databases and Information Systems Integration, Paphos, Cyprus, May 2006*. Hrsg. von Manolopoulos, Yannis, Filipe, Joaquim, Constantopoulos, Panos und Cordeiro, José, S. 81–86. isbn: 972-8865-41-4 (siehe S. 102).

Reese, Christine, Ortmann, Jan, Moldt, Daniel, Offermann, Sven, Lehmann, Kolja und Carl, Timo (2005). »Architecture for Distributed Agent-Based Workflows«. In: *Proceedings of the Seventh International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2005), Utrecht, Niederlande, as part of AAMAS 2005 (Autonomous Agents and Multi Agent Systems), July 2005*. Hrsg. von Henderson-Sellers, Brian und Winikoff, Michael, S. 42–49 (siehe S. 102).

Reese, Christine, Wester-Ebbinghaus, Matthias, Dörges, Till, Cabac, Lawrence und Moldt, Daniel (2007). »A Process Infrastructure for Agent Systems«. In: *MALLOW'007 Proceedings. Workshop LADS'007 Languages, Methodologies and Development Tools for Multi-Agent Systems (LADS)*. Hrsg. von Dastani, Mehdi, El Fallah, Amal, Leite, Joao und Torroni, Paolo, S. 97–111 (siehe S. 102).

Reese, Christine, Wester-Ebbinghaus, Matthias, Dörges, Till, Cabac, Lawrence und Moldt, Daniel (2008). »Introducing a Process Infrastructure for Agent Systems«. In: *LADS'007 Languages, Methodologies and Development Tools for Multi-Agent Systems*. Hrsg. von Dastani, Mehdi, El Fallah, Amal, Leite, João und Torroni, Paolo. Bd. 5118. Lecture Notes in Artificial Intelligence. Revised Selected and Invited Papers, S. 225–242 (siehe S. 102).

Richardson, C. (2019). *Microservices Patterns: With Examples in Java*. Manning Publications. isbn: 9781617294549 (siehe S. 42, 43).

Richter, Marc (2019). »Entwicklung einer modularen JavaScript-Bibliothek zur Modellerstellung. Unter besonderer Berücksichtigung eines Editorentwurfs zur Anbindung an externe Petrinetz-Simulatoren«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 245).

Rodriguez, Maria Alejandra und Buyya, Rajkumar (2018). »Containers Orchestration with Cost-Efficient Autoscaling in Cloud Computing Environments«. In: *CoRR* abs/1812.00300. arXiv: 1812.00300 (siehe S. 174).

Rölke, Heiko (1999). »Modellierung und Implementation eines Multi-Agenten-Systems auf der Basis von Referenznetzen«. Diplomarbeit. Universität Hamburg, Fachbereich Informatik (siehe S. 95).

Rölke, Heiko (2004). *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*. Bd. 2. Agent Technology – Theory and Applications. Berlin: Logos Verlag (siehe S. 5, 95, 97, 128, 150, 201, 450).

Röwekamp, Jan Henrik (Mai 2011). »Algorithmische Betrachtung von Switching Graphs«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 461).

Röwekamp, Jan Henrik (Dez. 2013). »Komplexitätstheoretische Betrachtungen an einer probabilistischen Abwandlung der Nearest Neighbour-Suche in kd-Trees«. Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 461).

Röwekamp, Jan Henrik (2016). »Fast thresholding of high dimensional Euclidean distances using binary squaring«. In: *23rd International Conference on Pattern Recognition, ICPR 2016, Cancún, Mexico, December 4-8, 2016*. IEEE, S. 3103–3108. ISBN: 978-1-5090-4847-2. DOI: 10.1109/ICPR.2016.7900111 (siehe S. 365, 367, 461).

Röwekamp, Jan Henrik (2018). »Investigating the Java Spring Framework to Simulate Reference Nets with Renew«. In: *Algorithms and Tools for Petri Nets, Proceedings of the 21th Workshop AWPN 2018, Augsburg, Germany*. Hrsg. von Lorenz, Robert und Metzger, Johannes. Reports / Technische Berichte der Fakultät für Angewandte Informatik der Universität Augsburg 2018-02, S. 41–46 (siehe S. 462).

Röwekamp, Jan Henrik, Buchholz, Manuela und Moldt, Daniel (2021). »Petri Net Sagas«. In: *Proceedings of the International Workshop on Petri Nets and Software Engineering 2021 co-located with the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021), Paris, France, June 25th, 2021 (due to COVID-19: virtual conference)*. Hrsg. von Köhler-Bussmeier, Michael, Kindler, Ekkart und Rölke, Heiko. Bd. 2907. CEUR Workshop Proceedings. CEUR-WS.org, S. 65–84 (siehe S. 311, 410, 462).

Röwekamp, Jan Henrik, Feldmann, Matthias, Moldt, Daniel und Simon, Michael (2019). »Simulating Place / Transition Nets by a Distributed, Web Based, Stateless Service«. In: *Petri Nets and Software Engineering. International Workshop, PNSE'19, Aachen, Germany, June 24, 2019. Proceedings*. Hrsg. von Moldt, Daniel, Kindler, Ekkart und Wimmer, Manuel. Bd. 2424. CEUR Workshop Proceedings. CEUR-WS.org, S. 163–164 (siehe S. 91, 404, 462).

Röwekamp, Jan Henrik und Haustermann, Michael (2015). »Applying Petri Nets to Approximation of the Euclidean Distance with the Example of SIFT«. In: *Petri Nets and Software Engineering. International Workshop, PNSE'15, Brussels, Belgium, June 22-23, 2015. Proceedings*. Hrsg. von Moldt, Daniel, Rölke, Heiko und Störrle, Harald. Bd. 1372. CEUR Workshop Proceedings. CEUR-WS.org, S. 323–324 (siehe S. 365, 461).

Röwekamp, Jan Henrik und Moldt, Daniel (2019). »RenewKube: Reference Net Simulation Scaling with Renew and Kubernetes«. In: *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings*. Hrsg. von Donatelli, Susanna und Haar, Stefan. Bd. 11522. Lecture Notes in Computer Science. Springer, S. 69–79. ISBN: 978-3-030-21570-5. DOI: 10.1007/978-3-030-21571-2_4 (siehe S. 214, 462).

Röwekamp, Jan Henrik, Moldt, Daniel und Feldmann, Matthias (2018). »Investigation of Containerizing Distributed Petri Net Simulations«. In: *Petri Nets and Software Engineering. International Workshop, PNSE'18, Bratislava, Slovakia, June 25-26, 2018. Proceedings*. Hrsg. von Moldt, Daniel, Kindler, Ekkart und Rölke, Heiko. Bd. 2138. CEUR Workshop Proceedings. CEUR-WS.org, S. 133–142 (siehe S. 207, 214, 220, 387, 462).

Röwekamp, Jan Henrik, Taube, Marvin, Mohr, Patrick und Moldt, Daniel (2021). »Cloud Native Simulation of Reference Nets«. In: *Proceedings of the International Workshop on Petri Nets and Software Engineering 2021 co-located with the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021), Paris, France, June 25th, 2021 (due to COVID-19: virtual conference)*. Hrsg. von Köhler-Bussmeier, Michael, Kindler, Ekkart und Rölke, Heiko. Bd. 2907. CEUR Workshop Proceedings. CEUR-WS.org, S. 85–104 (siehe S. 277, 406, 462).

Sametinger, Johannes (1997). *Software Engineering with Reusable Components*. Berlin, Heidelberg: Springer-Verlag. ISBN: 3540626956 (siehe S. 75).

Schleinzer, Benjamin (Dez. 2007). »Flexible und hierarchische Multiagentensysteme – Modellierung und prototypische Erweiterung von Mulan und Capa«. Diplomarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik (siehe S. 123, 206).

Schleinzer, Benjamin, Cabac, Lawrence, Moldt, Daniel und Duvigneau, Michael (Nov. 2008). »From Agents and Plugins to Plugin-Agents, Concepts for Flexible Architectures«. In: *NTMS 2008, 2nd International Conference on New Technologies, Mobility and Security, November 5-7, 2008, Tangier, Morocco. Electronical proceedings*. Hrsg. von Aggarwal, Akshai, Badra, Mohamad und Massacci, Fabio. IEEE Xplore, S. 1–5. ISBN: 978-1-42443547-0. DOI: 10.1109/NTMS.2008.ECP.49 (siehe S. 102).

Schmidt, Alexander (2016). »Integration von ActiveMQ in eine Petrinetz-basierte Agentenumgebung«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 203).

Searle, John R. (1969). *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press. doi: 10.1017/CBO9781139173438 (siehe S. 31).

Sedaghat, Shahrzad und Jahangir, Amir Hossein (2021). »RT-TelSurg: Real Time Telesurgery Using SDN, Fog, and Cloud as Infrastructures«. In: *IEEE Access* 9, S. 52238–52251. doi: 10.1109/ACCESS.2021.3069744 (siehe S. 107).

Senger, Alexander (Nov. 2021). »Erweiterung des Renew Petrinetz-Simulators um cloudnative Systemkomponenten«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 355, 356, 413).

Seppke, B. (2013). »Untersuchungen zum Korrespondenzproblem bei der Bestimmung mesokaliger Strömungen der Meeresoberfläche anhand von Satellitenbildern«. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 63, 104).

Seppke, B., Dreschler-Fischer, L., Heiming, J. und Wengenroth, F. (2010). »Fast Derivation of Soil Surface Roughness Parameters Using Multiband SAR Imagery and the Integral Equation Model«. In: *20th International Conference on Pattern Recognition, ICPR 2010, Istanbul, Turkey, 23-26 August 2010*. IEEE Computer Society, S. 3931–3934. doi: 10.1109/ICPR.2010.956 (siehe S. 104).

Seppke, B., Dreschler-Fischer, L. und Hübbe, N. (2010). »Using Snakes with Asymmetric Energy Terms for the Detection of Varying-Contrast Edges in SAR Images«. In: *ICPR*. IEEE, S. 2792–2795 (siehe S. 104).

Seppke, B., Gade, M. und Dreschler-Fischer, L. (2010). »The use of spatial constraints in the derivation of mesoscale sea surface current fields from multisensor satellite data«. In: *IEEE International Geoscience & Remote Sensing Symposium, IGARSS 2010, July 25-30, 2010, Honolulu, Hawaii, USA, Proceedings*. IEEE, S. 2226–2229. doi: 10.1109/IGARSS.2010.5650882 (siehe S. 104).

Shah, Jay und Dubaria, Dushyant (2019). »Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform«. In: *IEEE 9th Annual Computing and Communication Workshop and Conference, CCWC 2019, Las Vegas, NV, USA, January 7-9, 2019*. IEEE, S. 184–189. doi: 10.1109/CCWC.2019.8666479 (siehe S. 106).

Shoham, Yoav (1993). »Agent-Oriented Programming«. In: *Artificial Intelligence* 60.1, S. 51–92. doi: 10.1016/0004-3702(93)90034-9 (siehe S. 28).

Siddiqui, Umar, Tahir, Ghalib Ahmed, Rehman, Attiq Ur, Ali, Zahra, Rasool, Raihan Ur und Bloodsworth, Peter (2012). »Elastic JADE: Dynamically Scalable Multi Agents Using Cloud Resources«. In: *2012 Second International Conference on Cloud and Green Computing, CGC 2012, Xiangtan, Hunan, China, November 1-3, 2012*. Hrsg. von Liu, Jianxun, Chen, Jinjun und Xu, Guandong. IEEE Computer Society, S. 167–172. doi: 10.1109/CGC.2012.60 (siehe S. 90).

Simon, Michael (März 2014). »Concept and Implementation of Distributed Simulations in Renew«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 78, 104, 140, 144).

Simon, Michael und Moldt, Daniel (2016). »Extending Renew's Algorithms for Distributed Simulation«. In: *Petri Nets and Software Engineering. International Workshop, PNSE'16, Toruń, Poland, June 20-21, 2016. Proceedings*. Hrsg. von Cabac, Lawrence, Kristensen, Lars Michael und Rölke, Heiko. Bd. 1591. CEUR Workshop Proceedings. CEUR-WS.org, S. 173–192 (siehe S. 65, 78, 83, 104, 203, 252, 357, 422, 427).

Simon, Michael und Moldt, Daniel (2018). »About the Development of a Curry-Coloured Petri Net Simulator«. In: *Algorithms and Tools for Petri Nets, Proceedings of the 21th Workshop AWPN 2018, Augsburg, Germany*. Hrsg. von Lorenz, Robert und Metzger, Johannes. Reports / Technische Berichte der Fakultät für Angewandte Informatik der Universität Augsburg 2018-02, S. 53–54 (siehe S. 104).

Simon, Michael, Moldt, Daniel, Schmitz, Dennis und Haustermann, Michael (2019). »Tools for Curry-Coloured Petri Nets«. In: *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings*. Hrsg. von Donatelli, Susanna und Haar, Stefan. Bd. 11522. Lecture Notes in Computer Science. Springer, S. 101–110. isbn: 978-3-030-21570-5. doi: 10.1007/978-3-030-21571-2_7 (siehe S. 104, 157, 226).

Stachowiak, Herbert (1973). *Allgemeine Modelltheorie*. Wien, New York: Springer Verlag (siehe S. 14).

Štefanko, Martin, Chaloupka, Ondřej und Rossi, Bruno (2019). »The Saga Pattern in a Reactive Microservices Environment«. In: *Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, July 26-28, 2019*. Hrsg. von Sinderen, Marten van und Maciaszek, Leszek A. SciTePress, S. 483–490. doi: 10.5220/0007918704830490 (siehe S. 411).

Stonebraker, Michael (1986). »The Case for Shared Nothing«. In: *IEEE Database Engineering Bulletin* 9.1, S. 4–9 (siehe S. 37).

Štuikys, Vytautas, Burbaitė, Renata, Drąsutė, Vida und Bespalova, Kristina (2016). »Robot-Oriented Generative Learning Objects: An Agent-Based Vision«. In: *Agent and Multi-Agent Systems: Technology and Applications, 10th KES International Conference, KES-AMSTA 2016, Puerto de la Cruz, Tenerife, Spain, June 15-17, 2016, Proceedings*. Hrsg. von Jezic, Gordan, Chen-Burger, Yun-Heh Jessica, Howlett, Robert J. und Jain, Lakhmi C. Bd. 58. Smart Innovation, Systems and Technologies. Springer, S. 247–257. doi: 10.1007/978-3-319-39883-9_20 (siehe S. 107).

Suo, Simon, Regalado, Sebastian, Casas, Sergio und Urtasun, Raquel (2021). »TrafficSim: Learning To Simulate Realistic Multi-Agent Behaviors«. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, S. 10400–10409 (siehe S. 107).

Szyperski, Clemens A., Gruntz, Dominik und Murer, Stephan (2002). *Component software - beyond object-oriented programming, 2nd Edition*. Addison-Wesley component software series. Addison-Wesley. isbn: 0201745720 (siehe S. 75).

Taherizadeh, Salman und Grobelnik, Marko (2020). »Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications«. In: *Advances in Engineering Software* 140. doi: 10.1016/j.advengsoft.2019.102734 (siehe S. 106).

Tan, Wen Jun, Andelfinger, Philipp, Eckhoff, David, Cai, Wentong und Knoll, Alois C. (2021). »Causality and Consistency of State Update Schemes in Synchronous Agent-based Simulations«. In: *SIGSIM-PADS '21: SIGSIM Conference on Principles of Advanced Discrete Simulation, Virtual Event, USA, 31 May - 2 June, 2021*. Hrsg. von Giabbanelli, Philippe J. ACM, S. 57–68. doi: 10.1145/3437959.3459262 (siehe S. 105).

Tanenbaum, Andrew S. und Wetherall, David J. (2010). *Computer Networks*. 5th. USA: Prentice Hall Press. isbn: 0132126958 (siehe S. 237, 238).

Tell, Volker (März 2005). »Grundlagen für die prototypische Umsetzung eines Multiagentensystem basierten Leitmodells«. Diplomarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 45).

Tell, Volker und Moldt, Daniel (2005). »Ein Petrinetzsystem zur Modellierung selbstmodifizierender Petrinetze«. In: *Proceedings of the 12th Workshop on Algorithms and Tools for Petri Nets (AWPN 05)*. Hrsg. von Schmidt, Karsten und Stahl, Christian. Humboldt Universität zu Berlin, Fachbereich Informatik, S. 36–41 (siehe S. 45).

Timm, Ingo J. und Pawlaszczyk, Dirk (2005). »Large scale multiagent simulation on the grid«. In: *5th International Symposium on Cluster Computing*

and the Grid (CCGrid 2005), 9-12 May, 2005, Cardiff, UK. IEEE Computer Society, S. 334–341. DOI: 10.1109/CCGRID.2005.1558574 (siehe S. 91).

Valk, Rüdiger (1991). »Modelling Concurrency by Task/Flow EN Systems«. In: *Proceedings 3rd Workshop on Concurrency and Compositionality, GMD-Studien*. Bd. 191. GMD-Studien. St. Augustin, Bonn, Germany: Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Bonn (siehe S. 101).

Valk, Rüdiger (Juni 1995). »Petri Nets as Dynamical Objects«. In: *16th Intern. Conf. on Application and Theory of Petri Nets, Torino, Italy, Workshop proceedings*. Hrsg. von Agha, Gul und Cindio, Fiorella De. Workshop während der "'16th International Conference on Application and Theory of Petri Nets"', Turin, Italien, 26.–30. Juni, 1995. University of Torino (siehe S. 101).

Valk, Rüdiger (1998). »Petri Nets as Token Objects - An Introduction to Elementary Object Nets«. In: *19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal*. Hrsg. von Desel, Jörg und Silva, Manuel. Lecture Notes in Computer Science 1420. Berlin, Heidelberg, New York: Springer-Verlag, S. 1–25. DOI: 10.1007/3-540-69108-1_1 (siehe S. 6, 21, 24, 25, 101).

Wagner, Thomas (Sep. 2009). »A Centralized Petri Net- and Agent-based Workflow Management System«. In: *Proceedings of the Fifth International Workshop on Modeling of Objects, Components and Agents, MOCA'09, Hamburg*. Hrsg. von Duvigneau, Michael und Moldt, Daniel. Bericht FBI-HH-B-290/09. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik, S. 29–44 (siehe S. 103).

Wagner, Thomas (Juni 2012). »Agentworkflows for Flexible Workflow Execution«. In: *Petri Nets and Software Engineering. International Workshop PNSE'12, Hamburg, Germany, June 2012. Proceedings*. Hrsg. von Cabac, Lawrence, Duvigneau, Michael und Moldt, Daniel. Bd. 851. CEUR Workshop Proceedings. CEUR-WS.org, S. 199–214 (siehe S. 103).

Wagner, Thomas (2018). »Petri Net-based Combination and Integration of Agents and Workflows«. Diss. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 5, 29, 32, 83, 103, 127, 128, 135, 136, 201, 451).

Wagner, Thomas und Cabac, Lawrence (Juni 2013). »Advantages of a Full Integration between Agents and Workflows«. In: *Modeling and Business Environments MODBE'13, Milano, Italia, June 2013. Proceedings*. Hrsg. von Moldt, Daniel. Bd. 989. CEUR Workshop Proceedings. CEUR-WS.org, S. 353–354 (siehe S. 103).

Wagner, Thomas und Moldt, Daniel (2015). »Workflow Management Principles for Interactions Between Petri Net-Based Agents«. In: *Application and*

Theory of Petri Nets and Concurrency - 36th International Conference, Petri Nets 2015, Brussels, Belgium, June 21-26, 2015, Proceedings. Hrsg. von Devillers, Raymond und Valmari, Antti. Bd. 9115. Lecture Notes in Computer Science. Springer-Verlag, S. 329–349. ISBN: 978-3-319-19487-5. DOI: 10.1007/978-3-319-19488-2_17 (siehe S. 102, 103).

Wagner, Thomas, Moldt, Daniel und Köhler-Bussmeier, Michael (2016). »From eHornets to Hybrid Agent and Workflow Systems«. In: *Petri Nets and Software Engineering. International Workshop, PNSE'16, Toruń, Poland, June 20-21, 2016. Proceedings*. Hrsg. von Cabac, Lawrence, Kristensen, Lars Michael und Rölke, Heiko. Bd. 1591. CEUR Workshop Proceedings. CEUR-WS.org, S. 307–326 (siehe S. 104).

Wagner, Thomas, Quenum, José, Moldt, Daniel und Reese, Christine (2012). »Providing an Agent Flavored Integration for Workflow Management«. In: *Transactions on Petri Nets and Other Models of Concurrency V*. Hrsg. von Jensen, Kurt, Donatelli, Susanna und Kleijn, Jetty. Bd. 6900. Lecture Notes in Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, S. 243–264. ISBN: 978-3-642-29071-8 (siehe S. 102).

Wagner, Thomas, Schmitz, Dennis und Moldt, Daniel (2016). »Paffin: Implementing an Integration of Agents and Workflows«. In: *Multi-Agent Systems and Agreement Technologies - 14th European Conference, EUMAS 2016, and 4th International Conference, AT 2016, Valencia, Spain, December 15-16, 2016, Revised Selected Papers*. Hrsg. von Pacheco, Natalia Criado, Carrascosa, Carlos, Osman, Nardine und Inglada, Vicente Julián. Bd. 10207. Lecture Notes in Computer Science. Springer-Verlag, S. 67–75. ISBN: 978-3-319-59293-0. DOI: 10.1007/978-3-319-59294-7_7 (siehe S. 103).

Weber, Iaçanã I., Oliveira, Leonardo Londero de, Carara, Everton und Moraes, Fernando Gehm (2020). »Reducing NoC Energy Consumption Exploring Asynchronous End-to-end GALS Communication«. In: *33rd Symposium on Integrated Circuits and Systems Design, SBCCI 2020, Campinas, Brazil, August 24-28, 2020*. IEEE, S. 1–6. DOI: 10.1109/SBCCI50935.2020.9189896 (siehe S. 105).

Wester-Ebbinghaus, Matthias (Dez. 2010). »Von Multiagentensystemen zu Multiorganisationssystemen – Modellierung auf Basis von Petrinetzen«. https://ediss.sub.uni-hamburg.de/handle/ediss/3920. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Department Informatik (siehe S. 5, 103, 121, 193, 457).

Wester-Ebbinghaus, Matthias und Moldt, Daniel (2008a). »Modelling Multi-Agent Systems with Organizations in Mind«. In: *Proceedings of the 6th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS-2008, In conjunction with ICEIS 2008,*

Barcelona, Spain, June 2008. Hrsg. von Moldt, Daniel, Ultes-Nitsche, Ulrich und Augusto, Juan Carlos. Portugal: INSTICC PRESS, S. 81–90. isbn: 978-989-8111-43-2 (siehe S. 103).

Wester-Ebbinghaus, Matthias und Moldt, Daniel (2008b). »Structure in Threes: Modelling Organization-Oriented Software Architectures Built Upon Multi-Agent Systems«. In: *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Volume 3*. Hrsg. von Padgham, Lin, Parkes, David C., Müller, Jörg und Parsons, Simon. IFAAMAS, S. 1307–1310 (siehe S. 103).

Weyl, Julius, Lenfers, Ulfia A., Clemen, Thomas, Glake, Daniel, Panse, Fabian und Ritter, Norbert (2019). »Large-scale traffic simulation for smart city planning with mars«. In: *Proceedings of the 2019 Summer Simulation Conference, SummerSim 2019, Berlin, Germany, July 22-24, 2019*. Hrsg. von Durak, Umut. ACM, 2:1–2:12 (siehe S. 100).

Willrodt, Sven (Dez. 2019). »A Modular Model Checking Framework for Reference Nets in Renew«. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 6, 244, 345, 403).

Willrodt, Sven und Moldt, Daniel (2019). »Discussion of a Renew Implementation of a Modular Model Checking Framework for Reference Nets«. In: *Algorithms and Tools for Petri Nets. Proceedings of the 22nd. Workshop AWPN 2019*. Hrsg. von Bergenthum, Robin und Kindler, Ekkart. Fernuniversität in Hagen, Germany, S. 12–17. doi: 10.18445/20191003-092114-0 (siehe S. 345).

Willrodt, Sven, Moldt, Daniel und Simon, Michael (2020). »Modular Model Checking of Reference Nets: MoMoC«. In: *Proceedings of the International Workshop on Petri Nets and Software Engineering co-located with 41st International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2020), Paris, France, June 24, 2020 (due to COVID-19: virtual conference)*. Hrsg. von Köhler-Bussmeier, Michael, Kindler, Ekkart und Rölke, Heiko. Bd. 2651. CEUR Workshop Proceedings. CEUR-WS.org, S. 181–193 (siehe S. 6, 345, 403).

Wincierz, Martin (2018). »Verbesserung der Erweiterbarkeit und Benutzbarkeit der grafischen Oberfläche des Petrinetz Simulators Renew«. Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 66).

Wingerath, Wolfram (2019). »Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases«. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik (siehe S. 99).

Wolf, Karsten (2018). »Petri Net Model Checking with LoLA 2«. In: *Application and Theory of Petri Nets and Concurrency - 39th International Conference, PETRI NETS 2018, Bratislava, Slovakia, June 24-29, 2018, Proceedings*. Hrsg. von Khomenko, Victor und Roux, Olivier H. Bd. 10877. Lecture Notes in Computer Science. Springer, S. 351–362. doi: 10.1007/978-3-319-91268-4_18 (siehe S. 307).

Wooldridge, Michael (2009). *An Introduction to Multiagent Systems*. 2. Aufl. Chichester, UK: Wiley. isbn: 978-0-470-51946-2 (siehe S. 5, 29, 128).

Xu, Minxian, Toosi, Adel Nadjaran und Buyya, Rajkumar (2019). »iBrownout: An Integrated Approach for Managing Energy and Brownout in Container-Based Clouds«. In: *IEEE Trans. Sustain. Comput.* 4.1, S. 53–66. doi: 10.1109/TSUSC.2018.2808493 (siehe S. 174).

Zhang, Haitao, Ma, Huadong, Fu, Guangping, Yang, Xianda, Jiang, Zhe und Gao, Yangyang (2016). »Container Based Video Surveillance Cloud Service with Fine-Grained Resource Provisioning«. In: *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*. IEEE Computer Society, S. 758–765. doi: 10.1109/CLOUD.2016.0105 (siehe S. 174).

Zhang, Xingjun und Si, Wei (2021). »Efficient Auditing Scheme for Secure Data Storage in Fog-to-Cloud Computing«. In: *IEEE Access* 9, S. 37951–37960. doi: 10.1109/ACCESS.2020.2971630 (siehe S. 107).

Zhong, Zhiheng und Buyya, Rajkumar (2020). »A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources«. In: *ACM Transactions on Internet Technology* 20.2, 15:1–15:24. doi: 10.1145/3378447 (siehe S. 106).

Zia, Kashif, Farooq, Umar und Ferscha, Alois (2021). »When the Wisdom of Crowd is Able to Overturn an Unpopular Norm? Lessons Learned from an Agent-Based Simulation«. In: *SIGSIM-PADS '21: SIGSIM Conference on Principles of Advanced Discrete Simulation, Virtual Event, USA, 31 May - 2 June, 2021*. Hrsg. von Giabbanelli, Philippe J. ACM, S. 69–79. doi: 10.1145/3437959.3459248 (siehe S. 105).

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, 22. Januar 2023

Jan Henrik Röwekamp