Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Christian Hüning

Konzeption und Entwurf einer Architektur zum Einsatz von Multi-Agenten-Simulationen in der ökologischen Systemmodellierung

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Hüning

Konzeption und Entwurf einer Architektur zum Einsatz von Multi-Agenten-Simulationen in der ökologischen Systemmodellierung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thiel-Clemen
Zweitgutachter: Prof. Dr. Sarstedt

Eingereicht am: 20. September 2013

Christian Hüning

Thema der Arbeit

Konzeption und Entwurf einer Architektur zum Einsatz von Multi-Agenten-Simulationen in der ökologischen Systemmodellierung

Stichworte

Multiagentensystem, Ökologie, Verteilte Systeme, Systemmodellierung, Lastverteilung

Kurzzusammenfassung

Ziel der Arbeit soll es sein, eine Softwarearchitektur zum Einsatz einer Multi-Agenten-Simulationen unter besonderer Berücksichtigung der Anforderungen aus der ökologischen Systemmodellierung zu konzipieren und zu entwerfen. Als Quelle der Anforderungen wird dazu ein Systemmodell zur Untersuchung und Verbesserung des CO2-Haushaltes im Abdoulaye Naturschutzgebiet in Zentral-Togo entworfen.

Christian Hüning

Title of the paper

Conception and design of an architectur for the usage of multi-agent-simulations in ecological modelling

Keywords

Multi-Agent-System, Ecology, Distributed Systems, Systemmodelling, Load Balancing

Abstract

This work is about the conception and design of an architectur for the usage of multi-agent-simulations in ecological modelling. The requirements for this architectur will be extracted from a model which deals with the examination and improvement of the CO2 balance in Abdoulaye nature reserve in central Togo.

Inhaltsverzeichnis

1 Einführung **1**
1.1 Motivation . . . 1
1.2 Ziele der Arbeit . . . 2
1.3 Aufbau der Arbeit . . . 3

2 Grundlagen **4**
2.1 Zeitmanagement und Synchronisation in Verteilten Simulationssystemen . . . 4
2.1.1 Überblick . . . 4
2.1.2 Zeitarten . . . 4
2.1.3 Simulationstechniken . . . 5
2.1.4 Konservative & Optimistische Zeitsynchronisation . . . 6

3 Anforderungsanalyse **7**
3.1 Szenariobeschreibung Abdoulaye . . . 7
3.1.1 Zusammenarbeit mit H. Pereki . . . 7
3.1.2 Abdoulaye Naturschutzgebiet . . . 7
3.1.3 Carbon-Trade / CO2 Handel . . . 9
3.1.4 REDD+ Programm . . . 9
3.1.5 CDM - Clean Development Mechanism . . . 10
3.1.6 Ziele der Simulation . . . 10
3.2 Anforderung an die Simulationslösung . . . 11
3.2.1 Glossar . . . 11
3.2.2 Anwendungsfälle aus Modellobjektsicht . . . 12
3.2.3 Anwendungsfälle aus Entwicklersicht . . . 17
3.2.4 Nicht-funktionale Anforderungen . . . 20
3.2.5 Mengengerüste . . . 21
3.2.6 Ausgrenzungen . . . 21
3.2.7 Auswertung der Anwendungsfälle . . . 21

4 Stand der Technik **23**
4.1 MAS Systeme & Architekturen . . . 23
4.1.1 GAMA . . . 23
4.1.2 WALK . . . 26
4.1.3 Architektur von Vigueras . . . 28
4.1.4 Bewertung . . . 30

5 Konzeption **33**
5.1 Abdoulaye Wald Modell . . . 33
5.1.1 Abdoulaye Merkmale . . . 33
5.1.2 Ausgrenzungen im Modell . . . 33
5.1.3 Akteure und Umwelt . . . 34
5.1.4 Indikatoren . . . 35
5.1.5 Modell der Simulation . . . 35
5.2 Ein Layer-basierter Ansatz . . . 37
5.2.1 Konzept . . . 37
5.2.2 Trennung von Umwelt und Agenten . . . 38
5.2.3 Layerschnitt im Abdoulaye-Modell . . . 39
5.3 Management von Simulationsdaten . . . 41
5.3.1 Arten der Simulationsdatenspeicherung . . . 41
5.3.2 Anforderungen an die Speicherung der Daten . . . 42
5.3.3 Anforderungen an das Lesen von Daten . . . 42
5.3.4 Skalierung der Daten . . . 42
5.3.5 Datacube & Sternschema . . . 42
5.3.6 Untersuchung verschiedener DBMS als Cache . . . 44
5.3.7 Performance Test von PostGRE, Couchbase und VoltDB . . . 49
5.4 Layer als Plugins . . . 53
5.4.1 Überblick . . . 53
5.4.2 Erfahrungen aus anderen Projekten . . . 53
5.4.3 Frameworks im Vergleich . . . 54
5.5 Zeitmanagement . . . 55
5.5.1 Anforderungen an Präzision und Auswertung . . . 55
5.5.2 Hybridansatz . . . 56
5.5.3 Probleme bei der Auswertung . . . 58

6 Realisierung **60**
6.1 Entscheidung für Java 8 . . . 60
6.2 Erweiterung von JSPF . . . 61
6.2.1 Schnittstellenlimitierungen . . . 61
6.3 Systementwurf . . . 62
6.4 Gemeinsame Komponenten aller Systemteile . . . 63
6.4.1 Kommunikation . . . 63
6.4.2 Dezentraler Namensdienst . . . 64
6.5 SimulationCore Interna . . . 64
6.6 LayerContainer Interna . . . 65
6.6.1 Layer Verteilung . . . 66
6.7 Schnittstellen für Entwickler . . . 67
6.8 Zeitmanagement & Synchronisation . . . 69
6.9 Startvorgang des Systems . . . 71
6.9.1 Initialisierung . . . 71

6.9.2 Ausführungsablauf . . . 73
6.10 Implementierung des SAPD-Algorithmus . . . 75

7 Diskussion & Ausblick 78
7.1 Bewertung von RUN . . . 78
7.1.1 Agenten & Umwelt . . . 78
7.1.2 Visualisierung . . . 78
7.1.3 Performanz & Skalierbarkeit . . . 78
7.1.4 GIS Unterstützung . . . 79
7.1.5 Erweiterbarkeit . . . 80
7.2 Zusammenfassung . . . 80
7.3 Ausblick . . . 81
7.3.1 Linearisierung & Zeit . . . 81
7.3.2 Globales 3D-Koordinatensystem für Agenten & Umwelt . . . 81
7.3.3 Datacube Adapter . . . 81
7.3.4 Snapshot Adapter . . . 82
7.3.5 Beschreibungssprache für Model- & Szenariodefinition . . . 82

1 Einführung

1.1 Motivation

In der Modellierung für Simulationssysteme gibt es zwei große Richtungen. Beim sog. 'equation-based modeling' (EBM) wird das Modell aus Gleichungen gebildet, die eine Ausführung der Simulation in diskreten Schritten erlauben. Beim 'Agent-based modeling' besteht das Modell hingegen aus einzelnen Agenten, die jeweils das Verhalten der durch sie repräsentierten Individuen nachbilden.

Insbesondere in der Ökologie findet die letztere Variante der Simulation mittlerweile großen Anklang, da sich auf diese Art Individuen-basierte Systeme erstellen lassen, die reale Vorgänge und Phänomene häufig wirklichkeitsgetreuer abzubilden versprechen als ihre EBM-Varianten (vgl. (Parunak *et al.*, 1998)).

Trotz der großen Beliebtheit dieser Simulationssysteme, konnte sich bislang keine Standardarchitektur oder Software durchsetzen. Dies liegt u.a. an der Vielzahl der Möglichkeiten bei Modellierung und technischer Umsetzung eines solchen Systems sowie an den besonderen Erfordernissen der einzelnen Fachdomänen in denen die Systeme zum Einsatz kommen. Auch die vielen verschiedenen Datenformate in den unterschiedlichen Disziplinen macht eine standardisierte Umsetzung nicht einfacher. Als Beispiel seien hier nur die zahlreichen Formate für Geodaten angeführt[1].

Natürlich gibt es Multi-Agenten-Systeme (MAS), die gewissen Standards folgen und sehr erfolgreich, u.a. auch in der Lehre, genutzt werden (vgl. (Amouroux *et al.*, 2007)). Dennoch entsteht ständig neue, individuelle Simulationssoftware. Die Gründe dafür sind, dass die o.g. MAS häufig den spezifischen Gegebenheiten bzgl. Performanz, Datentypen, Visualisierung oder auch informatorischer Fachkenntnis der mit der Entwicklung betrauten Person(en) nicht oder nur ungenügend entgegenkommen. Die resultierenden Systeme sind dann oft nur sehr

[1] http://de.wikipedia.org/wiki/GIS-Datenformat

problemspezifisch nutzbar und erlauben beispielsweise häufig keine Integration von Daten anderer Forschergruppen, die sich mit gleichen Fragestellungen beschäftigen, aber ein anderes Datenformat für die Datenerhebung genutzt haben (z.B. GPS Daten).

Ein weiteres, besonders in der Ökologie verbreitetes Problem, tritt im Zusammenhang mit der stark unterschiedlichen, räumlichen und zeitlichen Skalierung von Messdaten auf. So können beispielsweise die GPS-Bewegungsdaten eines Geparden minütlich und auf wenige Meter genau vorliegen, die Temperaturdaten für den Nationalpark, in dem der Gepard sich bewegt, u.U. nur stundenaktuell und gleich für ein mehrere Hektar großes Gebiet. Das Problem ergibt sich nun, wenn man für einen Geparden zu einem Zeitpunkt X einen exakten Temperaturwert benötigt, da diese Werte nicht in der gleichen, feinen räumlichen und zeitlichen Skalierung vorliegen, wie bei der Gepardenposition.

1.2 Ziele der Arbeit

Ziel der Arbeit soll es sein, eine Softwarearchitektur zum Einsatz einer Multi-Agenten-Simulationen unter besonderer Berücksichtigung der Anforderungen aus der ökologischen Systemmodellierung zu konzipieren und zu entwerfen. Als Quelle der Anforderungen wird dazu ein Systemmodell zur Untersuchung und Verbesserung des CO2-Haushaltes im Abdoulaye Naturschutzgebiet in Zentral-Togo entworfen. Ursprung des Simulationsszenarios ist die Doktorarbeit von Pereki Hodabalo, der zur Zeit als Doktorant und Gaststudent an der HAW Hamburg seine Doktorarbeit schreibt.

Einige der Anforderungen ergeben sich bereits aus der Motivation für diese Arbeit. So soll es möglich sein ein fertiges Modell später zu erweitern. Zum einen durch das Hinzufügen neuer Daten in verschiedenen Formaten und aus unterschiedlichen Quellen, zum anderen durch die Anreicherung des bestehenden Modells mit neuen Simulationsaspekten. Wir wollen diesen Prozess als Information Integration (Thiel-Clemen, 2013) bezeichnen.

Des Weiteren soll die Architektur explizit nicht domänenspezifisch sein, eine zweckmäßige und für den jeweiligen Adressaten ausgelegte Visualisierung ermöglichen und nicht zuletzt auf eine effiziente Ausführung auch großer Simulationen ausgelegt sein.

1.3 Aufbau der Arbeit

Im Kapitel Grundlagen werden notwendige technische oder fachliche Sachverhalte eingeführt. Das Kapitel Anforderungsanalyse erläutert das Simulationsszenario im Abdoulaye Naturschutzgebiet und erarbeitet die Anforderungen an das zu erstellende Simulationssystem. Anschließend erörtert das Kapitel Stand der Technik die Architektur und den bisherigen Entwicklungsstand der Frameworks WALK (Thiel, 2013) und GAMA (Amouroux *et al.*, 2007) sowie der Architektur von Vigueras (Vigueras *et al.*, 2013). Das Kapitel Konzeption beschreibt aufbauend auf den Ergebnissen der Analyse die postulierte Softwarearchitektur, um das ökologische Modell zu realisieren. Das Kapitel Realisierung behandelt die Implementierung. Abschließend fasst das letzte Kapitel die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche und sinnvolle, zukünftige Erweiterungen des entstandenen Systems.

2 Grundlagen

2.1 Zeitmanagement und Synchronisation in Verteilten Simulationssystemen

2.1.1 Überblick

Zeitmanagement und Synchronisation in Verteilten Systemen ist ein nicht zu unterschätzendes Problem und von erheblicher Wichtigkeit für die korrekte und performante Ausführung eines solchen Systems. Für die spätere Diskussion verschiedener Synchronisationsansätze sind einige Grundlagen erforderlich, die im Folgenden beschrieben werden.

2.1.2 Zeitarten

Bei der Betrachtung der vorliegenden Problematik ist es wichtig, verschiedene Arten von Zeit im Simulationskontext zu unterscheiden. Pawlaszczyk & Timm (2007) und Fujimoto (1999) definieren bereits vier Arten: Die Physische Zeit, die Simulationszeit, die Lokale Virtuelle Zeit (LVZ) sowie die sog. Wallclock-Zeit. Hier kommt notwendigerweise noch eine fünfte Art hinzu, die Fachliche Zeit.

Physische Zeit Die tatsächliche, physische Zeit oder Uhrzeit, wie wir sie im alltäglichen Leben verwenden.

Simulationszeit Die Repräsentation der physischen Zeit im Simulationsmodell. Diese kann je nach Bedarf in unterschiedlichsten Schrittgrößen dargestellt werden.

wallclock-Zeit Ein Intervall der physischen Zeit, den das Simulationsprogramm benötigt, um zu laufen. Entspricht also der gesamten, echten Laufzeit des Programmes.

Lokale Virtuelle Zeit Ausgehend von der Einteilung der gesamten verteilten Simulation in logische (Teil-)Prozesse ist die LVZ die lokale Zeit innerhalb dieser Teilprozesse. Diese Zeit kann (und sollte sinnvollerweise) unabhängig vom Rest des Systems fortschreiten, sich aber an der Simulationszeit orientieren, um im Gesamtkontext eingeordnet werden zu können.

Fachliche Zeit Abhängig vom Simulationsmodell und der betrachteten Fachlichkeit können historische Daten verwendet oder Prognosen über die Zukunft getroffen werden, so dass die Einordnung in einen real existenten Zeitrahmen von großer Bedeutung ist. Die Fachliche Zeit soll diesem Bedarf gerecht werden, in dem sie den, sich aus dem Kontext der verwendeten Daten ergebenden, fachlichen Zeitraum der Simulation darstellt.

2.1.3 Simulationstechniken

In Abhängigkeit der zeitlichen Beschaffenheit eines Simulationsmodelles lässt sich der zeitliche Verlauf der dazugehörigen Simulation als kontinuierlich oder diskret beschreiben (Pawlaszczyk & Timm, 2007). Fortschritte in einer diskreten Simulationen finden zu definierten Punkten in der Zeit statt, wohingegen dies in kontinuierlichen Systemen nicht festgelegt ist.

Diskrete Simulationen lassen sich in Zeitschritt-basierte und Ereignis-basierte Modelle unterteilen. Der Unterschied besteht in der Art, in der die Zeitpunkte zum Fortschritt definiert werden. In ereignisorientierten Systemen geschieht dies über das Auslösen und Empfangen von Ereignissen. Agent A löst also etwa ein Ereignis E aus, auf welches Agent B reagiert. In Zeitschritt-basierten Systemen geschieht der Fortschritt zu von einem Zeitgeber vorgegebenen Zeitpunkten. Dies bedeutet, dass alle Agenten bzw. Bestandteile der Simulation angestoßen werden. Intuitiv geht man davon aus, dass Aktionen, die gleichzeitig geschehen, entsprechend unabhängig voneinander ablaufen. Diese Annahme trifft jedoch in vielen Anwendungsfällen verteilter Simulationsmodelle nicht oder nur bedingt zu, da die Aktionen der Agenten zwar verteilt und damit gleichzeitig aber ganz und gar nicht notwendigerweise voneinander unabhängig ablaufen. Das bedingt die Synchronisation der einzelnen Agenten und / oder Knoten um dem Kausalitätsproblem entgegen zu wirken.

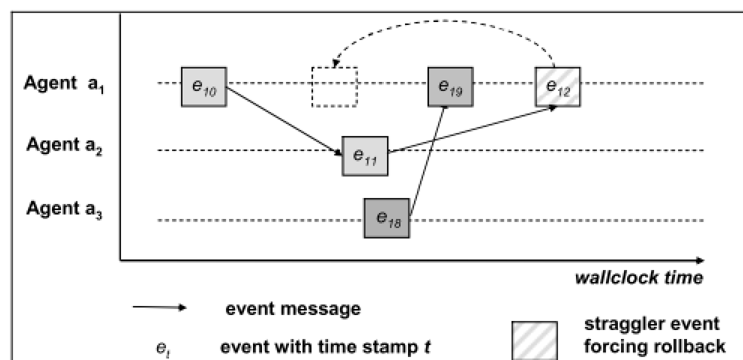

Abbildung 2.1: Kausalitätsproblem, Quelle: Pawlaszczyk & Timm (2007)

Dieses entsteht, wenn ein Prozess oder Agent eine Nachricht empfängt, deren Zeitstempel kleiner ist, als der aktuelle lokale Zeitstempel des letzten lokalen Events. In der Abbildung verletzt das späte Ankommen der Nachricht e12 das Kausalitätsprinzip, da zuvor bereits e19 eingetroffen ist. Möglicherweise sähe das Verhalten von Agent a1 auf Nachricht e19 anders aus, wenn zuvor e12 eingetreten wäre. Daher ist es nun erforderlich, diesen Umstand zu erkennen, ein Rollback der Zustände durchzuführen und die Ausführung in der korrekten Reihenfolge zu wiederholen.

2.1.4 Konservative & Optimistische Zeitsynchronisation

Die im vorherigen Kapitel beschriebene Synchronisation kann grundlegend auf zwei Arten durchgeführt werden: Konservativ oder Optimistisch.

Die Konservative Synchronisation vermeidet kausale Fehler komplett, in dem entsprechend abhängige Aktionen mit Beschränkungen versehen werden. Dadurch blockieren die verteilten Prozesse allerdings extrem häufig, was zur Aufhebung der wirklich parallelen Ausführung führt und somit deren Performanzvorteil im schlimmsten Fall annulliert.

Die Optimistische Synchronisation lässt Fehler grundsätzlich zu, geht aber davon aus, dass diese durch Rollbacks zu beheben sind und außerdem nicht allzu häufig auftreten. Prominenter Vertreter der optimistischen Synchronisation ist Jeffersons Time Warp Algorithmus (Jefferson & Sowizral, 1983), der in der Folge seiner Entstehung vielfach optimiert und verbessert wurde.

3 Anforderungsanalyse

3.1 Szenariobeschreibung Abdoulaye

Dieses Kapitel gibt einen Überblick des Simulationsszenarios. Zunächst wird auf die Zusammenarbeit mit Pereki Hodabalo eingegangen, dann wird das Abdoulaye Naturschutzgebiet vorgestellt und anschließend die von Pereki referenzierten Umweltschutzprogramme kurz erläutert. Der letzte Abschnitt beschreibt die Intention der Simulation.

3.1.1 Zusammenarbeit mit H. Pereki

H. Pereki ist als Botaniker der Domänenexperte für die Simulation. Viele der fachlichen Grundlagen für die Simulation stammen aus persönlichen Gesprächen zwischen ihm und mir, so dass aufgrund dessen und der Tatsache, dass der Abdoulaye Park bisher nicht Gegenstand intensiver Forschung war, das Meiste der nachfolgenden Analyse auf mündlichen Aussagen H. Perekis sowie seinem Paper (Pereki, 2013) beruht.

3.1.2 Abdoulaye Naturschutzgebiet

Das Abdoulaye Naturschutzgebiet wurde 1951 etabliert und umfasst eine Fläche von 300 km^2 in Zentraltogo etwa 500 km nördlich von Lomé. Es ist ein Naturschutzgebiet der IUCN (International Union for Conservation of Nature and Natural Resources) Kategorie IV und damit ein "geschütztes Gebiet zur Artenerhaltung durch gezielte Managementeingriffe".[1]

[1] http://bioval.jrc.ec.europa.eu/APAAT/pa/20978/

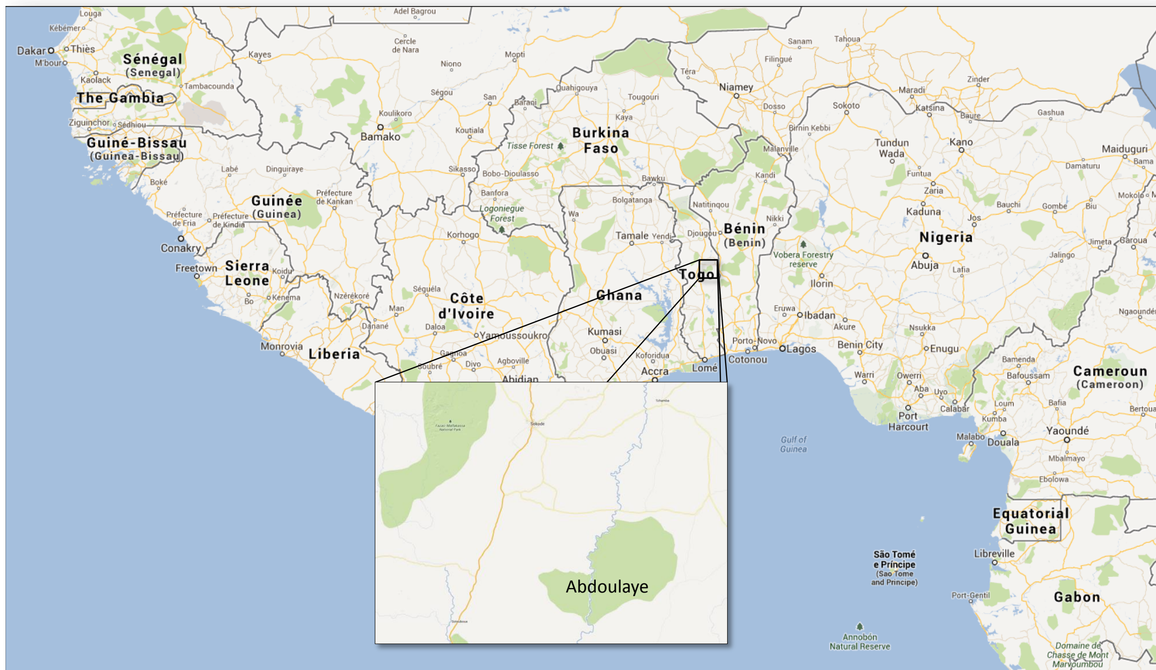

Abbildung 3.1: Abdoulaye Nationalpark (Quelle: Google Maps)

In Abdoulaye herrscht ein tropisches Savannenklima vor, welches durch eine Regensaison von April bis Oktober und eine Trockenzeit von November bis März gekennzeichnet ist. Besonders interessant ist der Park aufgrund seines tropischen Trockenwaldes. Es handelt sich dabei um eines der letzten Überbleibsel der "Dahomey Gap".

Der Naturpark liegt in einer der ärmsten Regionen ganz Togos und ist gänzlich durch die Landwirtschaft geprägt. Aus dieser Situation ergibt sich das Kernproblem in Abdoulaye. Die Landwirte brauchen gutes Farmland und die Brandrodung von Waldfläche ist ein Garant für geeigneten Boden, allerdings auch nur auf kurze Sicht. Die Nährstoffe aus den verbrannten Bäumen, wie z.B. Stickstoff, Kalium oder Magnesium, lagern sich mit der Asche auf dem Boden ab und fördern somit das Wachstum der Saat. Da die Nährstoffe aber nur an der Oberfläche in der Asche vorhanden sind und nicht nachhaltig im Boden selbst, lassen sich nur wenige Ernten einfahren, bevor das nächste Stück des Waldes gerodet werden muss. Bei geringer Besiedelung und einer damit einhergehenden ausreichend langen Brachephase, hat der Wald

sogar ausreichend Zeit sich zu regenerieren. Eine steigenden Bevölkerungsdichte und / oder unumsichtiger Umgang mit diesen natürlichen Rohstoffen führt allerdings schnell zur Zerstörung der empfindlichen Ökosysteme.

Zwar gibt es für den Abdoulaye Naturpark eine Art Forstamt, jedoch sind die eingesetzten Beamten laut Pereki zahlenmäßig zu wenig und verfügen auch nur über ungenügende Ortskenntnis sowie unzureichende Mittel, um die knapp 17600 Hektar Wald zu managen.

Ein weiteres Problem sind Nomaden, die mir ihren Viehherden durch den Park ziehen. Auf ihren Wanderungen fressen die Tiere junge Baumsprösslinge entlang ihrer Route und festigen den Boden durch ihr schieres Gewicht derart, dass neue Baumsamen nicht in der Lage sind zu keimen.

3.1.3 Carbon-Trade / CO2 Handel

Die Idee beim CO2-Handel ist es, Firmen mit Credits zu versorgen, die jeweils für eine gewisse Menge CO2 stehen. Stoßen diese Firmen bei ihrer Produktion nun CO2 aus, müssen sie diesen Austoß mit den erhaltenen Credits bezahlen. Verbraucht eine Firma nicht all ihre Credits, z.b. dank der Einführung klimafreundlicherer Produktionsprozesse, kann sie die übrigen Credits an andere Firmen verkaufen, die ihrerseits noch mehr CO2 emittieren müssen, als es ihre Credits erlauben. Auf diese Weise erhofft man sich einen Anreiz für Unternehmen zu schaffen, die Produktionsprozesse nachhaltiger zu gestalten und in neue, umweltfreundlichere Technologien zu investieren.

3.1.4 REDD+ Programm

REDD steht für **R**educing **E**missions from **D**eforestation and **D**egradation und wurde 2007 auf der UN Klimakonferenz auf Bali eingeführt. Die Idee hinter REDD ist es, Organisationen, Firmen und Staaten durch Kompensationszahlungen einen Anreiz zu geben, entweder gezielt oder durch die bewusste Umstrukturierung ihrer Geschäftsprozesse Waldgebiete zu schützen und damit die CO2-Emissionen niedrig zu halten. Eine Erweiterung des Modells, REDD+ genannt, bezieht die Speicherung von CO2 in den Wäldern durch nachhaltigeres Managament der entsprechenden Gebiete mit ein. Dabei wird vor allem auch auf die Verbesserung der wirtschaftlichen Lage der lokalen Bevölkerung geachtet. Der Anreiz, den Wald zu schonen, soll hier darüber geschaffen werden, dass der in den Bäumen gespeicherten Menge CO2 ein

finanzieller Wert zugeordnet wird. Werden nun Waldflächen gerodet, kann dies direkt in einen finanziellen Verlust umgerechnet werden[2].

3.1.5 CDM - Clean Development Mechanism

Der Clean Development Mechanism ist eine der flexiblen Maßnahmen, die 2007 im Kyoto Protokoll festgelegt wurden. Er ermöglicht es Industrieländern, die Kosten für die Erreichung eines Zieles, etwa der Reduktion von CO2-Emissionen, zu senken und gleichzeitig den Entwicklungsländern eine "ökologisch nachhaltige wirtschaftliche Entwicklung durch einen Zufluss an Geld und Technologie zu ermöglichen"(CDM in der Wikipedia). Dazu kann ein Land, welches einer Auflage zur Reduktion unterliegt, bei einem anderen Land, welches das nicht tut, Certified Emission Reductions, sog. CERs, erwerben und auf diesem Wege dem Verkäuferland Geldmittel zur Senkung des Treibhausgasaustoßes zur Verfügung stellen.

3.1.6 Ziele der Simulation

Bisherige Studien haben sich insbesondere in Togo fast ausschließlich auf die Untersuchung der Artenvielfalt und der Struktur der vorhandenen Ökosysteme konzentriert (Pereki, 2013). Laut (Pereki, 2013) ist das Fehlen der Daten zum Umfang der Biomasse des Waldes im Abdoulaye Park ein großes Hindernis zur effektiven Nutzung des REDD+ Programms, da sich u.a. die Bestimmung des Wertes der vorgesehenen Credits daraus ableitet. Ähnliches gilt auch für die Teilnahme am CDM Programm.

Basierend auf den Zielen, die H. Pereki für seine Doktorarbeit formuliert hat, ist das erklärte Ziel der Simulation, die Vorgänge an der Schnittstelle zwischen dem Abdoulaye Naturschutzgebiet und den dort lebenden Menschen möglichst genau abzubilden und dabei die Methoden zur Bestimmung der Biomasse, wie in Pereki (2013) beschrieben, als Bemessungsgrundlage für die Gesundheit des Ökosystems zu nutzen. Mit Hilfe des Systems gilt es, Maßnahmen zu finden, um zum einen die Artenerhaltung im Abdoulaye Naturschutzgebiet zu gewährleisten und zum anderen die Möglichkeiten zur Teilnahme am CO2-Handel für die Region herzustellen. Letzteres würde, so die Hoffnung, das ökonomische Wohlergehen der regionalen Bevölkerung steigern und im gleichen Zuge die Stabilität des Ökosystems im Abdoulaye Park erhalten.

[2] http://www.un-redd.org

3.2 Anforderung an die Simulationslösung

Ausgehend vom Simulationszenario und den von H. Pereki verfolgten Zielen, lassen sich Anforderungen an das zu implementierende MAS-System formulieren. Da hier eine Architektur für MAS entwickelt werden soll, werden zunächst Use-Cases aus dem Abdoulaye Szenario erfasst und anschließend in Form von Anforderungen an die Architektur ausgewertet.

Die Betrachtung der Anwendungsfälle muss aus zwei getrennten Blickwinkeln geschehen. Zum einen gibt es die Sicht der Modell-Entwickler, die dieses Modell mit Hilfe der Architektur umsetzen möchten. Diese Sicht werden wir als "Entwicklersicht" betrachten. Dann gibt es noch die Sicht der Modellobjekte, die im System auftreten und das System aus fachlicher Sicht benutzen. Diese Sicht werde ich "Modellobjektsicht" nennen.

3.2.1 Glossar

Nachfolgend werden einige Begriffe benutzt, deren Eindeutigkeit je nach Umfeld nicht gegeben ist. Daher hier ein Glossar, um Missverständnisse zu vermeiden:

Simulations-, Modellentwickler, Entwickler Ein Informatiker, der die Architektur nutzt, um eine bestimmte Simulation umzusetzen.

System Das System, welches durch die nachfolgend beschriebene Architektur gebildet wird.

Anwender Ein Anwender einer Simulation, der nicht programmiert, sondern die Simulation über grafische Benutzeroberflächen etc. steuert bzw. auswertet.

Modell Eine Beschreibung einer konkreten, domänenspezifischen Fachlichkeit, die als Simulation umgesetzt werden soll

Szenario Eine konkrete Belegung eines Modells. Beispiel: Die Definition, dass es "Bauern" und "Bäume" als Agenten gibt, gehört zum Modell. Die Definition, dass nun 100 Bauern und 10000 Bäume instantiiert und über einen Zeitraum von 10 Jahren simuliert werden, gehört zum Szenario.

3.2.2 Anwendungsfälle aus Modellobjektsicht

Anwendungsfall MO1 : Bauer fällt Baum

Name	MO1
Beschreibung	Ein Bauer fällt einen Baum aus seiner Umgebung
Akteure	Bauernagent
Auslöser	Entscheidung des Bauernagenten. Entweder um ein Feld anzulegen oder zur Brennholzgewinnung
Vorbedingungen	keine
Nachbedingungen	Der zu fällende Baum ist nicht mehr Bestandteil der Umwelt
Standardablauf	1. Bauer fragt bei der Umwelt Bäume aus seiner Umgebung ab 2. Bauer wählt einen Baum 3. Bauer teilt Umwelt mit, dass er den gewählten Baum fällt

Anwendungsfall MO2 : Bauer möchte ein Feld anlegen

Name	MO2
Beschreibung	Ein Bauer legt ein Feld in seiner Umgebung an
Akteure	Bauernagent
Auslöser	Entscheidung des Bauernagenten.
Vorbedingungen	keine
Nachbedingungen	Die Umwelt hat sich verändert. Insbesondere der Gebietstyp
Standardablauf	1. Bauer fragt bei der Umwelt den Zustand (Niederschlagswerte, Bodentyp, Bewuchs, CO2-Masse) umliegender Patches ab 2. Bauer wählt Patch aus 3. Ist der Bewuchs zu dicht, führt Bauer Anwendungsfall MO1 aus 4. Bauer legt Feld an 5. Bauer teilt Umwelt die Veränderung des Patches mit

Anwendungsfall MO3 : Baum lässt Samen fallen

Name	MO3
Beschreibung	Ein Baum lässt Samen fallen
Akteure	Baumagent
Auslöser	Entscheidung des Bauernagenten.
Vorbedingungen	Samen müssen reif sein
Nachbedingungen	In der Umwelt sind neue Samen verteilt
Standardablauf	1. Prüfung auf Chance für das Fallenlassen. 2. Baum lässt eine zufällig aus einem vordefinierten Bereich ausgewählte Menge Samen fallen. 3. Umwelt entscheidet, wo die Samen landen 4. Umwelt entscheidet, ob die Samen keimen

Anwendungsfall MO4 : Ranger bestraft Bauer für illegales Waldroden

Name	MO4
Beschreibung	Ein Parkranger erwischt einen Bauern beim illegalen Roden des Waldes und verhängt eine Geldstrafe
Akteure	Parkranger & Bauer
Auslöser	Beobachtung des Parkrangers und/oder Trigger von Extern
Vorbedingungen	keine
Nachbedingungen	Der Bauer hat weniger finanzielle Mittel zur Verfügung
Standardablauf	1. Ranger erkennt Regelverstoß in seinem Einzugsbereich 2. Ranger entscheidet über die schwere des Regelverstoßes durch eine Schadensanalyse (z.b. Beantwortung der Frage "Wie viele Bäume wurden gefällt?"). 3. Ranger zieht Geldstrafe von Bauer ein.

Anwendungsfall MO5 : Wetter verändert sich

Name	MO5
Beschreibung	Das Wetter in einer Region des Parks ändert sich. Dies betrifft Werte wie Niederschlag, Temperatur, Luftfeuchtigkeit.
Akteure	Wetter
Auslöser	Beginn der Regenzeit / Trockenzeit, übliche Wetterveränderungen durch das regionale Klima
Vorbedingungen	keine
Nachbedingungen	keine
Standardablauf	1. Das Wetter in einer Region verändert sich. 2. Die Veränderung wird berechnet. 3. Alle sich für das Wetter interessierende Modellobjekte werden benachrichtigt.

Anwendungsfall MO6 : Bodentyp ändert sich

Name	MO6
Beschreibung	Der Bodentyp in einer Region des Parks ändert sich durch Einwirkung von außen wie etwa Rodung, zu viel oder zu wenig Regen, Herden von Nutztieren, Regeneration des Waldes
Akteure	Gebiet
Auslöser	Bauern die Bäume fällen, Nutztierherden, die grasen, viel Regen, lange Dürre, Feuer
Vorbedingungen	keine
Nachbedingungen	keine
Standardablauf	1. Das Gebiet wird durch einen der Auslöser beeinträchtigt 2. Die Veränderung wird berechnet. 3. Alle sich für Bodentypveränderungen interessierende Modellobjekte werden benachrichtigt.

Anwendungsfall MO7 : Aggregation von CO2 Werten über mehrere Environment-Patches

Name	MO7
Beschreibung	Die Berechnung von CO2 Werten erfolgt innerhalb einer Baumstruktur. Dabei ist die unterste Stufe ein konkreter Baumagent, jede höhere Stufe setzt sich aus den akkumulierten Werten der in ihr enthaltenen Baumagenten zusammen.
Akteure	Forest, Government
Auslöser	Anfrage an die Forest Komponente
Vorbedingungen	-
Nachbedingungen	-
Standardablauf	1. Die Government Komponente erfragt den CO2 Wert eines bestimmten Gebietes 2. Die Forest Komponente errechnet den CO2 Wert für jeden Baumagenten. 3. Die CO2 Werte der Baumagenten werden für die nächsthöhere Rasterstufe des Gebietes aufsummiert. Dies wird so oft wiederholt, bis der Wert für das angefragte Gebiet vorliegt. 4. Der Wert wird zurückgegeben.

3.2.3 Anwendungsfälle aus Entwicklersicht

Anwendungsfall E1 : Erzeugung des Waldes und des Gebietes aus GIS Daten

Name	E1
Beschreibung	Eine der Hauptanforderungen ist es, den Wald sowie das Gebiet Abdoulaye insgesamt in der Simulation abzubilden. Detaillierte Informationen über die Form, Ausdehnung sowie den Bewuchs des Gebietes liegen in Form von GIS-Dateien vor.
Akteure	Entwickler
Auslöser	Notwendigkeit die Umwelt möglichst originalgetreu zu erstellen
Vorbedingungen	Daten müssen in kompatiblen Formaten vorhanden sein
Nachbedingungen	Informationen aus den GIS-Dateien wurden in die Datenbank überführt und sind abrufbar
Standardablauf	1. Simulationsentwickler laden benutzerdefinierte GIS Daten über die GUI in das System 2. Das System importiert die GIS-Daten in die Datenbank mit dem Ziel sie dem Simulationsentwickler zugänglich zu machen 3. Der Simulationsentwickler fragt die importierten GIS Daten programmatisch über eine Schnittstelle ab, um damit sein Environment zu erstellen.

Anwendungsfall E3 : Umsetzung eines Modells

Name	E3
Beschreibung	Der Entwickler setzt das erstellte Modell mit Hilfe des Simulationssystems um.
Akteure	Entwickler
Auslöser	Fertigstellung des Modells
Vorbedingungen	Modell muss fertig sein
Nachbedingungen	Das Simulationsmodell ist lauffähig umgesetzt.
Standardablauf	1. Der Entwickler erstellt Simulationslayer aus den Modellelementen. 2. Der Entwickler formalisiert Interaktionen zwischen den Modellelementen in ein Kommunikationsprotokoll. 3. Der Entwickler implementiert die Simulationslayer unter Berücksichtigung der Container-Interfaces. 4. Der Entwickler definiert eine Szenariobeschreibung 5. Der Entwickler startet das Simulationssystem mit der gewählten Szenariobeschreibung 6. Das System initialisiert die Simulation aus der Szenariobeschreibung.

Anwendungsfall E4 : Die Simulation soll zur Laufzeit visualisiert werden

Name	E4
Beschreibung	Die Simulation soll zur Laufzeit visualisiert werden.
Akteure	Adapter der Simulation, Nachbarsystem zur Visualisierung
Auslöser	Start des Visualisierungssystem
Vorbedingungen	Die Simulation läuft
Nachbedingungen	keine
Standardablauf	1. Die Simulation wird gestartet. 2. Das Visualisierungsprogramm(VSP) wird gestartet. 3. Das VSP empfängt regelmäßig Daten aus dem Visualisierungsadapter der Simulation. 4. Das VSP stellt diese Daten entsprechend dar.

Anwendungsfall E5 : Die Simulation soll nach der Berechnung visualisiert werden

Name	E5
Beschreibung	Die Simulation soll nach der Berechnung visualisiert werden.
Akteure	Datenspeicher der Simulation, Nachbarsystem zur Visualisierung
Auslöser	Start des Visualisierungssystems
Vorbedingungen	Die Simulation ist beendet oder weit genug fortgeschritten
Nachbedingungen	keine
Standardablauf	1. Die Simulation wird gestartet. 2. Die Simulation schreibt die Daten zum Simulationsverlauf in den Datenspeicher 3. (optional) Die Simulation wird beendet. 4. Das Visualisierungsprogramm(VSP) wird gestartet. 5. Das VSP fragt entsprechend seiner Anforderungen Daten aus dem Simulationsdatenspeicher ab. 6. Das VSP stellt diese Daten entsprechend dar.

3.2.4 Nicht-funktionale Anforderungen

Flexibilität Die Architektur soll flexibel sein in dem Sinne, dass der Entwickler in der Lösung seiner Probleme möglichst wenig eingeschränkt wird. Dies ist den mannigfaltigen Bedürfnissen verschiedener Modelle aus den unterschiedlichsten Fachbereichen geschuldet.

Kommunikationstransparenz Es soll dem Entwickler auf fachlicher Ebene verborgen bleiben, ob Aufrufe lokal oder entfernt stattfinden.

Lokationstransparenz Es soll dem Entwickler auf fachlicher Ebene verborgen bleiben, ob genutzte Komponenten lokal oder entfernt positioniert sind.

Verteilbarkeit Die Architektur soll es ermöglichen, eine domänenspezifische Verteilungslogik zu implementieren. Auf diesem Wege soll dem Umstand begegnet werden, dass die Verteilung von Agenten und Systemkomponenten nicht immer den gleichen Gesetzmäßigkeiten folgt, sondern ggf. nach Fachdomäne stark unterschiedlich sein kann.

Performanz Die Architektur soll es ermöglichen ein Modell derart zu implementieren, dass mindestens alle 250 ms (Henderson & Bhatti, 2003), besser aber mindestens alle 150 ms (vgl. (Jarschel *et al.*, 2013)) ein neues Bild von der Visualisierungskomponente gerendert werden kann.

3.2.5 Mengengerüste

Es ist wichtig, sich Gedanken über die Menge der zu simulierenden Modellobjekte zu machen. Daher hier eine Auflistung. Dabei ist zu beachten, dass die meisten Werte nicht absolut korrekt sind und das auch gar nicht sein können. Die Anzahl der Bäume im Abdoulaye Park kann derzeit z.B. nur geschätzt werden.

Bäume Nach Aussage H. Perekis gibt es im Abdoulaye Nationalpark ca. 4.000.000 Bäume.

Bauern N/A

Nomaden & Herden N/A

Ranger Es gibt derzeit lediglich 4 Ranger verteilt auf 2 Forstbüros.

Gebiet Das Gebiet umfasst etwas über 300 Quadratkilometer an Fläche

3.2.6 Ausgrenzungen

Aufgrund des Umfangs der architektonischen Maßnahmen und des begrenzten Umfangs dieser Arbeit, wird auf später notwendige GUI Werkzeuge für Modellierer, Administratoren und Analysten sowie Visualisierungslösungen hier zunächst nicht weiter eingegangen werden.

3.2.7 Auswertung der Anwendungsfälle

Aus den Anwendungsfällen ergeben sich folgende Anforderungen:

1. Die verschiedenen Layer der Simulation wie Bauern, Wetter, Gelände etc. müssen separat abgebildet werden können, um das Separation-of-Concerns Prinzip umzusetzen. Die internen Anforderungen der einzelnen Ebenen sind zu spezifisch, als dass es Sinn machen würde, alles auf einer Ebene zu implementieren.

2. Die Layer sollen sich wie Plugins in einem Container verwalten und zusammenstecken lassen.

3. Die Interaktionen zwischen diesen Layern, muss über deren Plugin-Interfaces festgelegt werden.
4. Die Kommunikation zwischen den Layern muss so effizient wie möglich sein
5. Die Kommunikation zwischen den Layern muss sowohl aktiv wie auch event-orientiert möglich sein.
6. Die Kommunikation zwischen den Layern sollte ausfallsicher sein.
7. Die Vielzahl von möglichen Aktionen und Veränderungen an der Umwelt muss in beliebiger Detailliertheit gespeichert werden können.
8. Die Speicherung der Simulationsdaten muss möglichst schnell erfolgen.
9. Die Lösung zur Datenhaltung muss arbiträre Abfragen ermöglichen und sollte den Ansprüchen der Information Integration von Thiel-Clemen (2013) genügen.
10. Das Simulationssystem parst in einer eigenen Beschreibungssprache erstellte Szenariodateien.
11. Das Simulationssystem liest in einem Format vorliegende Kommunikationsprotokolle.

Einige der o.g. Anforderungen sind zwar weich, allerdings lassen sich bis zu einem ersten Testlauf des späteren Systems, auch keine näheren Eingrenzungen von notwendige Höchst- oder Mindestwerte vornehmen. Auch sind diese Werte je nach individueller Simulation sehr unterschiedlich. Möchte man keine Echtzeitvisualisierung, sinken die Anforderungen an die Performance zum Beispiel deutlich. Gleichzeitig werden andere Anforderungen wie Korrektheit oder Sicherheit vielleicht mehr in den Fokus gerückt. Daher gehe ich bei den meisten quantifizierbaren Anforderungen davon aus, die Lösung möglichst effizient nach aktuellem Wissenstand umzusetzen.

4 Stand der Technik

4.1 MAS Systeme & Architekturen

In dieser Sektion werden zwei bereits bestehende MAS Systeme sowie eine Architektur zur verteilten Ausführung einer großen Simulation auf ihre Tauglichkeit zur Umsetzung des Abdoulaye-Models untersucht. Die Hauptkriterien sind dabei in Anlehnung an (Amouroux *et al.*, 2007, S. 2f):

- Beliebige Agenten
- Beliebige Umwelt
- Visualisierung
- Performanz & Skalierbarkeit
- GIS Unterstützung
- Erweiterbarkeit

4.1.1 GAMA

Überblick

GAMA ist eine Modellierungs- und Simulationsplattform, die aufbauend auf der Eclipse-Plattform eine komplette Entwicklungsumgebung für Multiagentensysteme bietet. Herausragendes Merkmal ist dabei vor allem eine relativ einfache, aber dennoch mächtige DSL[1] mit Namen "GAML", die es auch Nicht-Programmierern erlaubt, komplexe Modelle zu erstellen. Der Entwicklung von GAMA ging eine Analyse der zu diesem Zeitpunkt relevanten MAS-Frameworks voraus (Amouroux *et al.*, 2007, S. 3) und es wurde in Abwägung der diversen Vor- und Nachteile entschieden, GAMA basierend auf RepastJ zu implementieren. Dementsprechend ist GAMA in Java geschrieben und somit auf allen Java-fähigen Betriebssystemen lauffähig

[1] Domain Specific Language

(Amouroux *et al.*, 2007). In GAMA ist alles ein Agent, auch das Environment wird hier zum Beispiel als Grid von 'Cell-Agents' abgebildet.

Agenten

Wie bereits erwähnt, ist in GAMA alles ein Agent. Alle Agenten sind in einem, immer vorhandenen Weltagenten enthalten. Agenten, die Akteure im Sinne der Simulation darstellen, werden als 'Species' über die DSL deklariert und mit einem Verhalten versehen. Dabei kann es sowohl reaktives wie auch aktives Verhalten geben. Die reaktiven Agenten können über "when"-Klauseln vor ihrer Ausführung bestimmte Bedingungen prüfen. Auf diese Weise lassen sich, zumindest in begrenztem Umfang, kontextsensitive Aktionen implementieren.

Umwelt

Die Umwelt in GAMA wird, wie eingangs erwähnt, auch mittels Agenten abgebildet. Diese Umweltagenten können drei verschiedene Typen von Topologie haben: Kontinuierlich, Raster oder Graph. Die kontinuierliche Topologie erlaubt sowohl eine beliebige Definition geometrischer Formen, als auch die beliebige Bewegung innerhalb des Koordinatensystems. In einem Raster ist die Umwelt in Zellen eingeteilt. Agenten befinden sich immer in einer Zelle und bewegen sich zellenweise vorwärts. Die Graphtopologie erzeugt schließlich einen Graphen, sodass Agenten sich an Knoten befinden, die mit Kanten verbunden sind. Der Weltagent hat als Standarddefinition eine kontinuierliche Topologie mit einem Rechteck der Größe 100m x 100m. Eine der Besonderheiten von GAMA ist die Möglichkeit, die Abmessungen der Geometrie einer solchen Gridworld auf verschiedenste Art und Weise zu definieren. Die Maße können entweder direkt als Breite und Höhe angegeben werden, oder auch aus einer GIS Shapedatei, einer Rasterdatei oder einer Bilddatei mit verlinkter Weltdatei extrahiert werden. Für die dateibasierten Verfahren ist es nur erforderlich, die entsprechende Datei anzugeben. Den Rest übernimmt das Framework. Zu guter Letzt ist es noch möglich mehrere Shape- oder Rasterdateien anzugeben und GAMA dann die Vereinigung der darin enthaltenen Umrisse übernehmen zu lassen.

Visualisierung

Weiterhin hervorzuheben sind die Möglichkeiten zur Visualisierung der Simulation in 2D- und 3D-Grafik einerseits und in Form von Diagrammen andererseits. Diese Visualisierung lässt sich ebenfalls sehr komfortabel über die DSL ausdrücken.

Eine Simulation kann auf beliebig viele Arten gleichzeitig visualisiert werden. Dabei ist es möglich, die Aktualisierungsfrequenz der einzelnen grafischen Darstellungen von „Synchron

zum Modell“ bis „Aus“ in feingranularen Stufen einzustellen, so dass die Gesamtperformanz der Simulation nicht zu sehr leidet.

Performanz & Skalierbarkeit

Bezüglich der Performanz bei großen und sehr großen Simulationen mit mehr als 100.000 Agenten, lässt sich sagen, dass GAMA leider nicht sonderlich gut skaliert. Bei Testläufen auf Systemen verschiedener Größe (Macbook Pro, 4 Kerne mit 2,3 Ghz, 8 GB Ram sowie Dual XEON Hexacore 2,9 Ghz, 48 GB Ram) wurden stets nur 4 (logische) Kerne genutzt, so dass eine Referenzsimulation ungeachtet der verwendeten Hardware ab ca. 75.000 Agenten mit einer durchschnittlichen Zyklusdauer von ca. 650ms erheblich zu langsam lief.

GAMA bietet als 3rd-Party-Erweiterung einen sogenannten „Headless-Mode“, der im Wesentlichen die GUI abschaltet und somit alle Ressourcen der Simulation zur Verfügung stellt. Die Hoffnung, GAMA würde hier auf die verfügbaren Kerne skalieren, erfüllte sich leider nicht. Laut Aussage des Entwicklers ist der Headless-Mode dazu gedacht, ein Modell zu untersuchen, in dem man mehrere, verschiedentlich parametrisierte Simulationen auf diesem Modell parallel startet. Jede dieser Simulationen nutzt dann, falls verfügbar, andere Kerne eines Multiprozessorsystems, Grids oder Clusters aus. Auf die Geschwindigkeit der einzelnen Simulation hat dies allerdings keine Auswirkung.

GIS-Unterstützung

GAMA unterstützt den Import von GIS-Daten in Form von Shape-Dateien durch die Nutzung der OpenMap Bibliothek[2] auf einfache Art und Weise und erlaubt es diese Daten sofort in der Simulation zu nutzen[3]. Leider erlaubt es diese Implementierung nicht auch andere GIS Formate wie zum Beispiel GeoTIFF zu lesen.

Erweiterbarkeit

GAMA basiert auf der Eclipse-Plattform und lässt sich auch über Plugins in dieser Plattform erweitern. Diese Plugins erweitern dann die Auszeichnungssprache GAML, sodass sich neue Funktionalitäten wie gewohnt nutzen lassen. Weiterhin gibt es die Möglichkeit den Entities über Plugins sog. Skills zur Verfügung zu stellen. Beispielsweise ist die Datenbankanbindung der Agenten auf diese Art umgesetzt. Wer noch weiter eintauchen möchte, kann sich auch den Quellcode von der Projektseite herunterladen.

[2] http://openmap.bbn.com/
[3] http://code.google.com/p/gama-platform/wiki/RoadTraficModel1v13

4.1.2 WALK

Überblick

WALK ist ein an der HAW-Hamburg im Rahmen des Forschungsschwerpunktes „Bevölkerungsschutz“ entwickeltes MAS System zur Simulation von Evakuierungsszenarien. Bei der Entwicklung von WALK wurde insbesondere darauf geachtet, möglichst alle Systembestandteile modular und austauschbar zu halten, um größtmögliche Flexibilität zu ermöglichen.

Agenten

WALK sieht keine spezielle Art von Agenten vor, ist allerdings grundsätzlich darauf ausgelegt, Agenten in der definierten Umwelt zu bewegen. Die Agentenimplementierung kann beliebig ausfallen, muss aber in der sprache JAVA erfolgen. Eine spezielle DSL zur Definition des Agentenverhaltens liegt hier nicht vor. Für die Umsetzung von Simulationen deren Agenten keinerlei wirklichen Ortsbezug aufweisen bzw. für deren Simulation dieser Ortsbezug eine nur untergeordnete oder gar keine Rolle spielt (etwa eine Simulation aus dem Finanzbereich), ist WALK eher ungeeignet. Anders sieht es aus, wenn die örtliche Bedeutung im erstellten Modell anders interpretiert wird. Etwa als soziale Nähe oder Ähnliches. Für derartige Fälle könnte WALK als System eine effiziente Lösung sein.

Umwelt

Die Definition der Umwelt ist in WALK durch Sektoren gegeben. Ein Sektor ist im Wesentlichen ein Rechteck im 3D-Raum, welches durch zwei dreidimensionale Vektoren definiert ist. Damit ist WALK in der Lage, dreidimensionale Räume durch die Aneinanderreihung und / oder Stapelung von Sektoren darzustellen. In der derzeitigen Implementierung wird die dritte Dimension allerdings ausgeblendet, der Z-Wert ist immer 0. Dies hat seinen Grund in der Partitionierung der Agenten, welche im 2D-Raum erheblich weniger komplex ist, als im 3D-Raum (Thiel, 2013, S. 51).

Visualisierung

WALK bietet bereits eine simple Visualisierung an. Diese kann wahlweise als 2D oder 3D Variante mit Java2D bzw. JMonkey genutzt werden. Die Visualisierung ist über einen Binding-Mechanismus komplett vom Rest der Simulation entkoppelt, sodass sie sich einfach gegen eine andere Lösung austauschen lässt. Für die Zukunft dürfte das sehr interessant sein, vor allem wenn umfangreichere Szenarien visualisiert werden sollen.

GIS-Unterstützung

Eine GIS-Komponente ist in WALK bereits vorgesehen. Sie soll die Umwelt initial mit Daten befüllen. Bisher ist diese Funktionalität allerdings noch nicht implementiert. Vielmehr liegt hier eine Mock-Implementierung vor, die eine Reihe von Testszenarien bereitstellt. Darunter die RiMEA Tests zur Validierung des Verhaltens der Fußgängeragenten.

Erweiterbarkeit

Wie bereits erwähnt ist WALK sehr modular aufgebaut. So sind alle Komponenten über eine Kommunikationskomponente miteinander verbunden und dadurch allesamt verteilbar. Die jeweilige Komponentenimplementierung ist ebenfalls austauschbar. Beispielsweise existieren bislang zwei verschiedenen Implementierungen für die Kommunikationskomponente. Eine auf Basis von TCP/IP und eine, die RabbitMQ als Kommunikationsträger nutzt (Thiel, 2013, S. 58ff).

Performanz & Verteilung

Christian Thiel hat sich 2013 in seiner Masterarbeit (Thiel, 2013) intensiv mit dem Thema der Verteilung und damit einhergehend mit der intelligenten Distribution der Agenten auf die verteilten Knoten beschäftigt. Als Ergebnis dieser Arbeit lassen sich nun auch große Simulationen mit 300.000 Agenten auf handelsüblichen Notebooks ausreichend performant durchführen. Es ist hier der Hinweis von C. Thiel in seiner Masterarbeit zu beachten, dass die verwendeten Agenten eine sehr simple KI besitzen und daher bei realen Szenarien von deutlich weniger Agenten je Maschine auszugehen ist (Thiel, 2013, S. 93).
Thiel setzt die Verteilung über sein DistributedEnvironment (Thiel, 2013, S. 37ff) um. Die Umwelt wird dazu in Sektoren geschnitten, die zusammengefasst zu Clustern auf mehrere Stationen (Rechnern) verteilt werden können. Die Partitionierung dieser Sektoren ist der Hauptuntersuchungsgegenstand in Thiels Arbeit und wird in WALK mittels eines zuvor wählbaren Algorithmus (z.b. QHull, KMeans) umgesetzt (Thiel, 2013, S. 93ff). Die Algorithmen berücksichtigen dabei in der Regel das Verhalten der Agenten, um diese und damit die Umwelt in entsprechende Gruppen zu teilen. Beispielsweise kann bei Fußgängern als Agenten von einem starken Lokalitätsverhalten ausgegangen werden, so dass die räumliche Distanz und die Bewegungsrichtung als Merkmale für eine Partitionierung herangezogen werden können.
Dabei ist zu beachten, dass die Agenten relativ gleichmäßig auf die verfügbare Hardware verteilt sein sollten. Da die Simulation synchron in diskreten Schritten abläuft, müssen stets alle Agenten auf allen Stationen einen Schritt ausführen ehe die gesamte Simulation voranschreiten

kann. Sind die Agenten ungleichmäßig verteilt, kann es, zumindest bei identischer Hardware, zu Engpässen in der Berechnung kommen und mehrere Stationen mit weniger Agenten warten auf eine Station mit vielen Agenten (Thiel, 2013, Kapitel 7.2.1, S. 91). Um diesem Problem zu begegnen, werden die Agenten dynamisch zur Laufzeit umverteilt, was sich laut (Thiel, 2013, S. 100) mit dem Algorithmus von Peschlow insgesamt am besten bewerkstelligen ließ.

4.1.3 Architektur von Vigueras

Überblick

Die von Vigueras *et al.* (2013) vorgeschlagene Architektur für skalierbare Multiagentensysteme für interaktive Anwendungen sieht einen massiv verteilten Ansatz vor und basiert auf der Überlegung, dass vorhandene Systeme nicht groß genug skalierbar sind (Vigueras *et al.*, 2013, Kap. 2). Auch sind die angebotenen Visualisierungslösungen nicht skalierbar und arbeiten zudem nicht schnell genug (Abstand zweier Bilder < 250ms), um der Anforderung an Benutzerinteraktion gerecht zu werden.

GIS-Unterstützung sieht Vigueras zunächst nicht vor. Es wird sich ausschließlich auf die Skalierbarkeit des Systems durch Zugabe von mehr Hardware konzentriert. Die betrachtete Simulation ist eine Fußgängersimulation und daher denen in WALK bisher umgesetzten Szenarien recht ähnlich.

Ein bedeutender Unterschied zu WALK ist, dass Vigueras eine asynchrone und damit nicht in diskreten Zeitschritten ablaufende Simulation vorsieht.

Agenten

Vigueras trifft keinerlei Annahmen über die Agenten, abgesehen davon, dass sie sich bewegen. Die Bewegung als solche wird nicht vorausgesetzt, sondern dient vielmehr als anschauliches Merkmal, um die entstehenden Probleme der Umweltpartitionierung ansprechen zu können. Die Architektur würde entsprechend auch für jede andere Art von Agenten funktionieren, die nicht explizit auf Bewegung ausgelegt ist. Tatsächlich würde das die "Grenzkonflikte" erheblich vereinfachen.

Umwelt

Auch für die Umwelt wird architektonisch an sich nichts vorausgesetzt. Der zentrale Verteilungsaspekt bedeutet natürlich implizit, dass die Umwelt teilbar sein muss, ansonsten hat der Entwickler hier freie Hand.

Performanz & Verteilung

Die Architektur sieht ebenfalls eine Partitionierung der Umwelt vor. Allerdings handelt es sich hierbei zunächst um eine statische Partitionierung, die zur initialen Aufteilung der Agenten dient. Die Umwelt wird in Regionen geteilt, wobei jede Region von einem sog. ActionServer (AS) verwaltet wird. Die Agenten jeder Region werden als Threads eines ClientProcess (CP) abgebildet. Jeder AS und CP läuft nun auf einem eigenen Rechner, um über ausreichende Rechenkapazität zu verfügen.
Das Innenleben eines AS besteht aus einer semantischen Datenbank, die den Weltzustand hält, einer 'CrowdControl'-Komponente sowie einem Kommunikationsinterface. Die semantische Datenbank enthält zwar den gesamten Datenbestand, der AS kann aber nur den Teil der Welt, für den er zuständig ist, exklusiv in seiner Kopie der Datenbank verändern. In der 'CrowdControl'-Komponente leben ActionExecutioner-Threads, die dafür zuständig sind, von den CPs eingehende Agentenaktionen zu prüfen und im Erfolgsfall auszuführen. Auch jeder CP enthält eine Kopie der semantischen Datenbank. Sie dient als Zustandsgrundlage für die Agenten.
Der konkrete Ablauf sieht dann so aus, dass ein Agent eines CPs einen Aktionswunsch an den für ihn zuständigen AS schickt. Dieser prüft die Durchführbarkeit des Wunsches mit Hilfe eines seiner ActionExecutioner-Threads u.a. auch gegen den Zustand in der lokalen Datenbank und akzeptiert oder verbietet dann die Ausführung der Aktion. Wird die Aktion ausgeführt, geht eine entsprechende Nachricht zurück an den CP. Zusätzlich verschickt der AS Nachrichten über die Änderungen in seiner Region an die anderen AS, damit diese ihre Kopie der Datenbank aktualisieren können.
Etwas komplexer wird der Ablauf, sobald ein Agent eine Aktion im Randgebiet zwischen zwei Regionen durchführen will. In diesem Fall erhält der ActionExecutioner-Thread des AS abermals den Aktionswunsch, stellt aber zusätzlich zu der Prüfung seiner eigenen Datenbank noch eine Anfrage an den benachbarten AS. Die Aktion kann dann erst durchgeführt werden, sobald alle beteiligten Instanzen zugestimmt haben (2-Phase-Commit).
Die interne Struktur des AS ermöglicht eine massiv parallele Bearbeitung von eingehenden Aktionswünschen. Die Reihenfolge der Aktionswünsche wird dabei über eine FIFO-Queue hergestellt, wobei regionsübergreifende Aktionen aufgrund ihrer aufwendigeren Bearbeitungen in einer eigenen Queue auflaufen, die dann entsprechend priorisiert werden kann. Vigueras et al. konnten an dieser Stelle auch schon GPU-basierte Verfahren einsetzen.

Visualisierung

Für die Visualisierung wird ein ganz ähnlicher Ansatz gewählt, wie für die anderen Komponenten des Systems. Visual Client Prozesse (VCP) stellen virtuelle Kameras mit Sicht auf die Umwelt dar. Jeder VCP läuft dabei wieder auf einem eigenen Rechner, um über dessen volle Kapazität verfügen zu können. Der Ausschnitt, den ein VCP abdeckt kann entweder nur innerhalb einer Region liegen oder aber auch mehrere Regionen schneiden. In jedem Fall kommuniziert der VCP mit den entsprechenden ActionServern, um über den Fortschritt der Simulation im Bilde zu bleiben. Die VS schicken dazu die akzeptierten Zustandsänderungen für ihre Region auch an den VCP, so dass dieser daraus eine Visualisierung erzeugen kann. Diese Struktur ermöglicht es, eine Vielzahl von unterschiedlichen Visualisierungen anzubieten und jede davon beliebig einfach oder komplex zu gestalten, je nachdem was der Benutzer benötigt.

4.1.4 Bewertung

Universelle Einsetzbarkeit

Alle drei Frameworks bzw. Architekturn sind grundsätzlich in der Lage, beliebige Agenten zu nutzen. Am flexibelsten erscheint jedoch der Ansatz von Vigueras, da hier keinerlei Annahmen zum Verhalten der Agenten getroffen werden. Allerdings ist darauf hinzuweisen, dass man von einem asynchronen Simulationsverlauf ausgeht. Man bekommt also nicht die Möglichkeit, diskrete Zeitschritte bzw. Zustände der Simulation zu betrachten, sondern kann sich stets nur einen Verlauf ansehen. Für diesen Verlust an Präzision zum Vorteil der Performanz muss sich bewusst entschieden werden.

Die Implementierung in WALK ist derzeit stark auf die Simulation von Fußgängern ausgerichtet und müsste zunächst mit einigem Aufwand um andere Konzepte erweitert werden, um anderweitig einsetzbar zu sein.

GAMA bietet im Rahmen seiner DSL mit Reflexen, Aktionen, Aufgaben, die gewichtet, sortiert oder auch probabilistisch geordnet werden können, sowie endlichen Zustandsautomaten nahezu beliebiges Verhalten an. Komplette Sub-Modelle sind so zwar ebenfalls möglich, müssen sich aber immer innerhalb der gegebenen Strukturen bewegen, was eventuell bei sehr komplexen Modellen zu einem Problem werden könnte.

Performanz & Skalierung

Der Ansatz von Vigueras skaliert nachweislich sehr gut bei einer steigenden Anzahl von Agenten. Durch Hinzufügen von Hardware lässt sich der entstehende lineare Mehraufwand

entsprechend auffangen (Vigueras *et al.*, 2013, Kap. 4). Aufgrund der Asynchronizität der Kommunikation und der Threads läuft die Simulation in weiten Teilen unabhängig von einander ab und etwaige Wartezeiten werden auf ein Minimum reduziert. Diese Designentscheidung hat allerdings zur Folge, dass es nicht möglich ist, eine synchrone Simulation in diskreten Schritten auszuführen.
Auch hat dies Auswirkungen auf die Bedeutung der Partitionierung. Wie (Thiel, 2013) untersucht hat, ist es zur performanten Ausführung verteilter, synchroner Fußgängersimulationen notwendig die Partitionierung entsprechend intelligent durchzuführen. Auftretende Probleme waren hier wartende Stationen, aufgrund von lokaler Überlastung mit zu vielen Agenten sowie zu häufige Netzwerkkommunkation durch ungünstige Partitionierungen. Durch die fehlende Synchronizität im Ansatz von Vigueras entfällt zumindest das Warten auf andere Stationen, da jeder Teilnehmer des Systems weiter rechnen kann, sobald er selbst dazu bereit ist. Der hohen Netzlast versuchen Vigueras et al. mittels des QHull-Algorithmus zu begegnen, wie sie in (Vigueras *et al.*, 2010) darlegen.

GAMA schließlich skaliert innerhalb einer Simulation nicht mit zusätzlicher Hardware und ist damit leider nicht wie beworben für die Durchführung von Simulationen "mit Millionen von Agenten"[4] geeignet. Zumindest nicht, wenn die Simulation entsprechend schnell durchgeführt werden soll.

Bedienbarkeit

Der große Trumpf von GAMA ist die Erstellung von Szenarien über die mitgelieferte DSL. Zwar erfordert diese einiges an Einarbeitungszeit, jedoch ist man anschließend auch ohne tiefschürfende Programmierkenntnisse in der Lage, eigene Modelle in funktionierende Simulationen zu überführen. Der Umstand, dass GAMA dabei plattformunabhängig läuft, trägt sein Übriges zur einfachen Benutzbarkeit bei.

WALK erfordert derzeit noch sehr gute Programmierkenntnisse in Java, um eigene Modelle zu integrieren, falls man dazu andere, als die bisher vorhandenen Agenten benötigt. Die Umwelt lässt sich theoretisch auch ohne diese Kenntnisse beschreiben, ist aber dennoch sehr aufwendig in ihrer Darstellung als vektorbasierte Sektorenmenge. Zur Beschreibung eines konkreten Szenarios innerhalb eines Modells muss letztlich noch eine XML-Datei erstellt werden, deren Struktur ebenfalls erst einmal erlernt werden muss. Zusammenfassend kann man feststellen, dass in WALK bereits einige gute Vorarbeit für eine zukünftig einfachere Modellumsetzung getroffen worden ist, man hier aber noch dringend einen Prozess benötigt, der die verschiedenen Teile zusammenbringt und vereinheitlicht.

[4] siehe http://code.google.com/p/gama-platform/

Vigueras Ansatz kann hinsichtlich der Bedienbarkeit nicht bewertet werden, da es sich hier um ein Konzept handelt und nicht um ein fertiges System. Es ist daher nur für Informatiker geeignet, die ein MAS implementieren möchten.

Relevanz für das Abdoulaye-Modell

Die Anzahl der Bäume im Abdoulaye Park beläuft sich auf ca. 6.000.000. Da das Modell vorsieht, jeden Baum als Agenten zu betrachten, und zu den Bäumen noch komplexe Agenten an der Schnittstelle sozio-ökologischer Systeme zu berücksichtigen sind, wird die mit GAMA umsetzbare Menge an Agenten deutlich überschritten. Die fehlende Möglichkeit GeoTIFF und andere GIS-Formate zu importieren ist leider auch ein Show-Stopper. GAMA ist daher als Basis zur Umsetzung des Abdoulaye Modells leider nicht geeignet.

Die Architektur von Vigueras et al. ist aufgrund ihrer Asynchronizität ebenfalls nicht direkt anwendbar. Der Verlust an Präzision dürfte für viele Anwendungsszenarien nicht hinnehmbar sein, da man gerne an bestimmten Stellen sehr genau in das Geschehen hineinsehen können möchte. Jedoch sind die Ideen zur Visualisierung und insbesondere die Idee zur Definition von Kameras sehr spannend und sollten definitiv näher untersucht werden, wenn in der zu entwickelnden Architektur über Visualisierung nachgedacht werden wird.

WALK ist bestens zur Simulation von Fußgängern oder fußgängerähnlichen Agenten geeignet. Die bereits implementierten Partitionierungsalgorithmen sollten unbedingt weiterverwendet werden. Die Technologie zur Szenariodefinition ist als Unterbau solide, sollte aber unbedingt durch die Erstellung von grafischen Werkzeugen benutzerfreundlicher gestaltet werden. WALK soll als Plugin in der vorliegenden Architektur zum Einsatz kommen.

5 Konzeption

5.1 Abdoulaye Wald Modell

5.1.1 Abdoulaye Merkmale

- 500km nördlich von Lomé
- Größe Abdoulaye Naturschutzgebiet: 79830 Hektar
- Davon bewaldet: 17563 Hektar
- Regensaison: April - Oktober
- Trockenzeit: November - März
- Hauptsächlich dichte Trockenwälder
- Homogene, offene Wälder mit Anogiessus leiocarpus und Pterocarpus erinaceus.
- Der nördliche Teil des Waldes (Alibi) wird durch lokale Gemeinden verwaltet, der südliche durch die Regierung. Frage: Wer macht es besser?

5.1.2 Ausgrenzungen im Modell

Folgende Dinge sollen nicht im Modell berücksichtigt werden, da sie nach Absprache mit Pereki Hodabalo keine Relevanz haben:

Wilde Tiere Wilde Tiere im Abdoulaye Wald haben keinen Einfluss auf die Biomasse des Waldes. Nutztiere hingegen schon. Letztere sind daher auch Bestandteil der Modellierung. S.u.

Fische und andere Wassertiere Die Flüsse sind Bestandteil der Modellierung, darin lebende Tier allerdings nicht, da auch sie keinen Einfluss auf die Biomasse haben. Die Flussläufe sorgen für die Entstehung von Feuchtwaldgebieten in ihrer unmittelbaren Nähe. Diese haben hinsichtlich der CO_2-Aufnahmefähigkeit andere Merkmale als Trockenwälder und weitere Waldarten.

5.1.3 Akteure und Umwelt

Akteure

Folgende Akteure treten in dem Modell in Erscheinung:

Baum Bäume sind die Hauptagenten der Simulation. Sie werden die größte Anzahl an Agenten stellen.

Kuh Kühe tauchen in der Simulation in Form von Herden und als Einzeltiere auf. Einzeltiere finden sich bei den Bauern und haben eher untergeordneten Einfluss auf die Entwicklung der Biomasse. Herden hingegen ziehen mit Nomaden umher und stören das Ökosystem ganzer Waldabschnitte durch Abgrasen von Wiesen und das Feststampfen des Bodens.

Bauer Die Bauern haben ihre Felder und Höfe an den Randgebieten des Waldes. Da der gesamte Wald ein Naturschutzgebiet ist, dürfen die Bauen eigentlich weder Bäume fällen, noch Waldstücke roden um weitere Felder anzulegen oder aus den Bäumen Kohle zu gewinnen. Jedoch geschieht beides immer wieder und macht somit eine Kontrolle der Bauern durch das Forest Office nötig.

Nomaden Nomaden ziehen mit Herden von Kühen durch das Abdoulaye Gebiet. Dabei stampfen die Tiere den Boden fest, so dass Baumsamen nicht sprießen können. Zudem fressen die Kühe junge Baumsprösslinge.

Förster Die Förster agieren von zwei Büros aus. Zu ihren Problemen zählt, dass sie sehr wenige sind und zumeist eine nur schlechte Kenntnis des Waldgebietes besitzen, sodass sie auf Hilfe der örtlichen Bevölkerung angewiesen sind. Erwischen sie einen Bauern beim illegalen Fällen von Bäumen oder beim Legen von Feuer, verhängen sie Geldstrafen. Außerdem nehmen Förster diverse Aufgaben zur Wiederaufforstung des Waldes wahr. Beispielsweise das gezielte Legen von Feuern im Wald um bestimmte Gebiete zu sanieren.

Umwelt

Die Umwelt wird durch folgende Parameter gestaltet bzw. gebildet:

Temperatur Jährlich, Täglich etc.

Regen Stärke, Dauer etc.

Wind Wichtig sind hier Stärke und Richtung. Der Wind weht beispielsweise von Bäumen gefallene Samen in andere Teile des Parks.

Waldtyp Trockenwald, Savanne, Parkland, Wald in Flussnähe

CO2 Menge Menge an CO2, die je Parzelle von 20m x 5m gehalten wird. Dann extrapolieren.

Feuer Feuer werden sowohl von Bauern als auch von Förstern gelegt. In beiden Fällen wird für das Modell zunächst davon ausgegangen, dass die Feuer sich nicht verbreiten.

Anzahl Bäume je Spezies und Parzelle Die Verteilung der Bäume unterschiedlicher Spezies spielt eine nicht kleine Rolle, da die unterschiedlichen Spezies durchaus bedeutsame Unterschiede bzgl. ihrer Fähigkeit, CO2 zu binden, aufweisen.

Lichtintensität Nach Seidl *et al.* (2012) spielt die Lichtintensität für das Wachstum, die Weiterverbreitung und die Fähigkeit, CO2 aufzunehmen, eine wichtige Rolle.

5.1.4 Indikatoren

Wichtiger Bestandteil der Modellbildung sind Indikatoren, die in der Summe das Ziel der Simulation definieren. Für das Abdoulaye Modell werden folgende Indikatoren festgelegt:

Biomasse

Die Biomasse je Gebiet als aggregierter Wert der Biomasse der in diesem Gebiet vorhandenen Bäume.

Artenvielfalt

Die Artenvielfalt je Gebiet in Form eines Indexes. Etwa Simpson-Index oder Shannon-Index.

Wohlstand der Bevölkerung

Ein zusammengesetzter Wert aus Nahrungsversorgung, Wohnungsituation, allgemeiner Zufriedenheit.

5.1.5 Modell der Simulation

Das vorläufige Modell der Simulation besteht aus drei großen Bereichen. Der Bereich "Forest" bildet alle Vorgänge des Ökosystems im Abdoulaye Park ab. Außerdem werden hier die Werte für den CO2-Gehalt etc. berechnet. "LandUse" fasst alle Aktivitäten der Menschen zusammen. Folglich beinhaltet dieser Bereich die Agenten für Bauern, Ranger etc.. Diese interagieren von hier aus mit dem Wald. Als dritte Instanz überwacht und steuert der Bereich "Government" die beiden anderen.

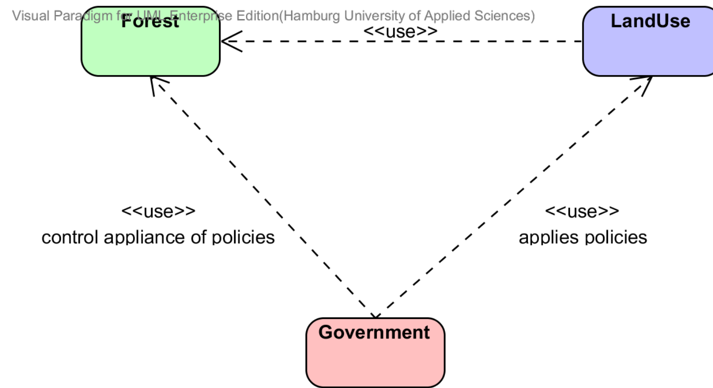

Abbildung 5.1: Domänenmodell Überblick

Die Details für die Modellierung des Waldes zeigt die nachfolgende Abbildung. Hervorzuheben ist hier die Modellierung der Umgebung in Form von stufenweise feiner bzw. gröber werdenden "Environment-Patches". Über diese lassen sich später die aggregierten Berechnungen diverser Werte durchführen.

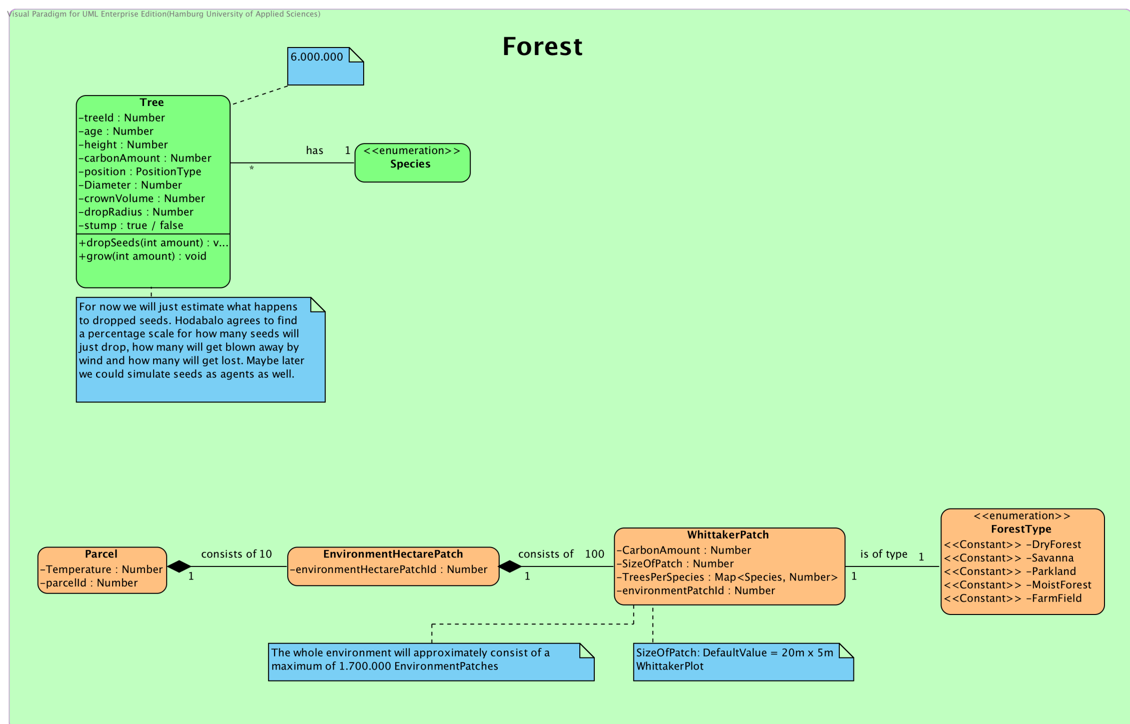

Abbildung 5.2: Forest-Submodell

Die Modellierung des LandUse-Bereiches basiert auf einem Gespräch welches ich mit Hodabalo im April 2013 geführt habe. Er befindet sich gerade auf einer Prospektionsreise im Togo, so dass sich dieses Submodell gegebenenfalls noch einmal ändern wird.

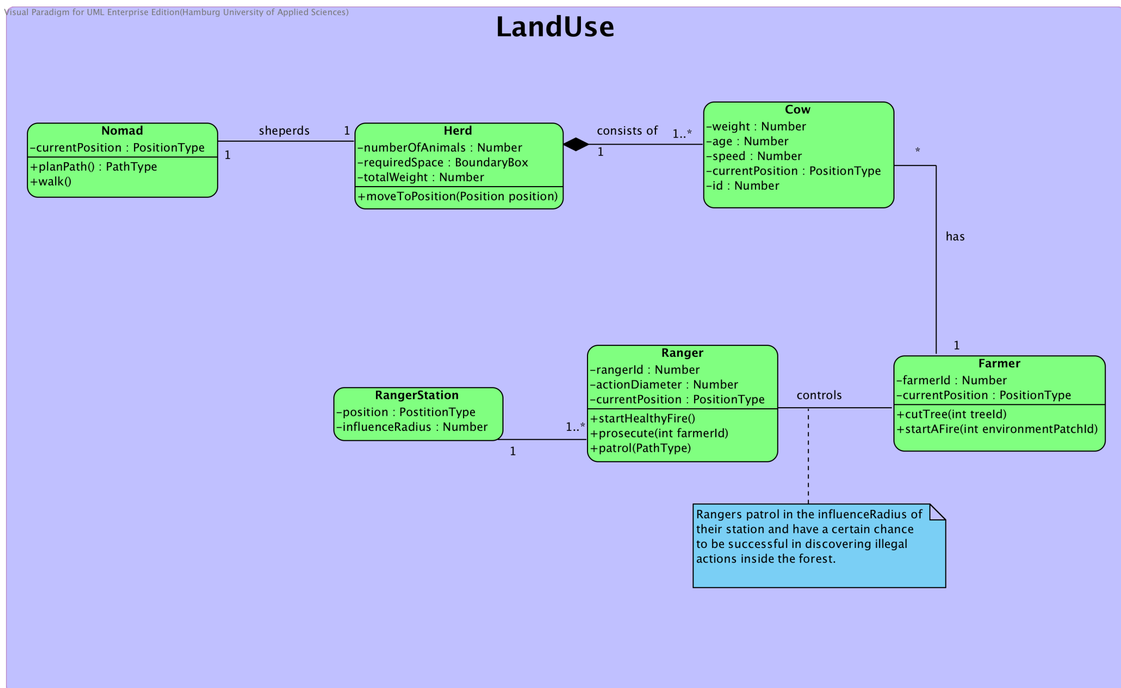

Abbildung 5.3: LandUse-Submodell

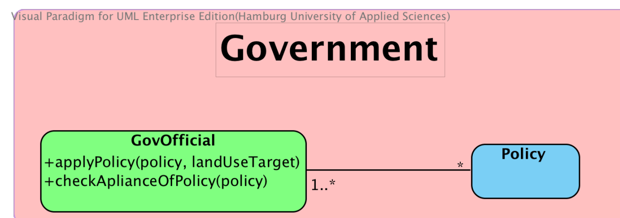

Abbildung 5.4: Government-Submodell

5.2 Ein Layer-basierter Ansatz

5.2.1 Konzept

Die Anforderungen aus dem vorherigen Kapitel zeigen, dass es durchaus sinnvoll ist, die verschiedenen Modellobjekte implementationstechnisch zu trennen, da sie teils deutlich von-

einander abweichende Techniken in der Umsetzung benötigen. Um dem SoC-Ansatz[1] gerecht zu werden, wird nachfolgend eine Aufteilung der Modellobjekte auf verschiedene Schichten oder Layer vorgeschlagen. Die Idee ist der Struktur von Geodaten entlehnt. Diese werden häufig in sog. Shapefiles gespeichert und über Layer zusammengefasst, so dass man eine schichtbasierte Repräsentation der Daten aus den Shapefiles enthält.
Ein Layer bildet dabei stets einen Aspekt des Gesamtmodells ab. Aspekte sollten notwendig umfangreiche, aber in sich abgeschlossene und ausreichend überschaubare Bereiche sein. Eine sinnvolle Trennung wäre zum Beispiel zwischen dem Gelände und den Agenten, die die Hauptakteure einer Simulation darstellen, gegeben.
Grundsätzlich existieren auf den Layern Modellobjekte. Ein Modellobjekt ist dabei eine Entität aus dem gebildeten Modell. Diese Objekte können entweder statisch oder dynamisch sein, im letzteren Fall lässt sich dann noch eine Klassifizierung nach den Attributen 'aktiv' und 'reaktiv' vornehmen. Aktive, dynamische Modellobjekte nehmen von sich aus Aktionen an anderen Modellobjekte oder sich selbst vor, reaktive Modellobjekte reagieren nur auf äußere Einwirkungen und agieren dann entsprechend.
Die Gesamtheit der Layer für sich genommen ergäbe noch keine fertige Simulation. Dazu ist es erforderlich, die Layer und auch die Agenten untereinander kommunizieren zu lassen. Dazu können die Layer Abhängigkeiten zu anderen Layern definieren und dadurch deren Interfaces für ihr Innenleben benutzbar machen.

5.2.2 Trennung von Umwelt und Agenten

Die Unterscheidung zwischen Umwelt und Agenten als harte Trennlinie zwischen aktiven und passiven Elementen einer Simulation verschwimmt hier. Jeder Layer kann Teile der Umwelt und auch Agenten enthalten. Agenten können auch genutzt werden, um bestimmte Aspekte der Umwelt abzubilden, etwa Feuer oder Wetter. Die Entscheidung darüber, wie die Aufteilung der für die Simulation notwendigen Agenten und Umweltteile auf die einzelnen Layer geschieht, obliegt gänzlich dem Modellentwickler.
Als eine Art 'Best Practice' hat sich bereits jetzt herausgestellt, zirkuläre Abhängigkeiten in Analogie zum Komponentenentwurf in der Softwareentwicklung zu vermeiden und diese entweder mit einem Zwischenlayer oder durch Zusammenlegen der Modellobjekte auf einen gemeinsamen Layer aufzulösen.

[1] Separation of Concerns

5.2.3 Layerschnitt im Abdoulaye-Modell

Die folgende Grafik veranschaulicht die gefundenen Schichten im Abdoulaye Modell und ihre Beziehungen untereinander.

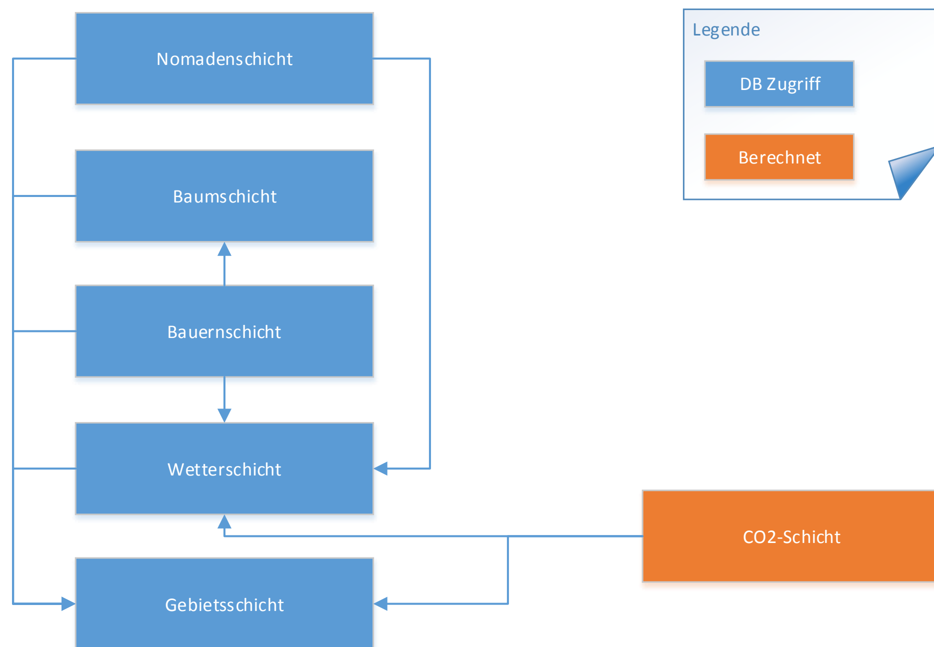

Abbildung 5.5: Abdoulaye Layerkonzept

Baumschicht

Art Dynamisch mit Agenten, Bäume sind die Hauptpagenten der Simulation

Menge ca. 4 Millionen

Quelle Satellitendaten und Prospektion durch Hodabalo mit anschließender Extrapolation

Skalierung Kontinuierliche Veränderungen. Bäume wachsen, fallen m

Bauernschicht

Art Dynamisch mit Agenten

Menge ca. 1000

Quelle H. Pereki versucht Informationen zum Verhalten der Bauern in Abdoulaye von Kommilitonen zu bekommen

Skalierung Vermutlich Aktionen in gewissen Zeitintervallen, um Ernte-, Regen- und Dürreperioden gerecht zu werden

Wetterschicht

Art Noch unklar, muss aber auf jeden Fall Daten über Temperatur, Luftfeuchtigkeit, Sonnenscheindauer, vielleicht Lichtintensität (Seidl *et al.*, 2012), Niederschlag und Wind umfassen.

Quelle Sensordaten, falls verfügbar. Satellitendaten, Prospektion, oder historische Daten ansonsten

Gebietsschicht

Art Statisch in Form und Ausdehnung. Höhenwerte abfragbar. Enthält Gebietstyp, dieser ist dynamisch und kann sich durch den Einfluss und das Zusammenspiel diverser Schichten verändern. Dazu zählen: Bauern, das Wetter, Nomaden mit ihren Herden

Menge ca. 1.700.000 Sektoren

Quelle GIS-Daten, Weitere Daten aus dem Datacube, sofern vorhanden.

Skalierung Sektor < Hektar < Parzelle

CO2-Schicht

Art Ein akkumulierter Wert des in Biomasse gebundenen CO2. Die Angabe bezieht sich immer auf ein definiertes Gebiet.

Menge Ein Wert je Sektor.

Quelle Berechnungsfunktion von H. Pereki.

Skalierung Sektor hat Einzelwert, größere Felder sind akkumuliert.

Nomadenschicht

Art Dynamisch mit Agenten

Menge N/A

Quelle N/A

5.3 Management von Simulationsdaten

Die von MAS erzeugten Daten zu speichern ist aus mindestens zweierlei Gründen interessant. Zum einen erhält die Simulation selbst so die Möglichkeit, auf historische Daten zuzugreifen um damit ihre Berechnungen zu bedienen. Zum anderen ist es so möglich eine spätere Analyse und Auswertung oder auch Visualisierung in Form eines Replays durchzuführen. Ist die Speicherung schnell genug, lässt sich die Visualisierung womöglich sogar parallel zum Simulationsverlauf durchführen. Nachfolgend eine Analyse der wichtigsten Aspekte dieses Teils der MAS Architektur.

5.3.1 Arten der Simulationsdatenspeicherung

Es gibt hier zwei relevante Arten, wie sich Simulationsdaten speichern lassen.

Persistente Simulationsobjekte

Eine Möglichkeit ist es die Objekte der Simulation ähnlich einem OR Mapper wie Hibernate in eine Datenbank zu speichern. Dies bietet sich an, um etwa den Zustand der Simulation wegzuschreiben und entsprechend später wieder darauf zuzugreifen. Dies könnte man beispielsweise zur Erstellung von Snapshots (in beliebiger Frequenz) nutzen, zu denen man später wieder zurückspulen kann.

Reine Simulationsdaten

Die andere Möglichkeit besteht darin, die in den Objekten gehaltenen und von den Objekten erzeugten Daten auf beliebige Art in eine Datenbank zu schreiben. Hier ist nicht notwendigerweise vorausgesetzt, dass sich die tatsächlichen Simulationsobjekte später wieder einfach und schnell rekonstruieren lassen, sondern es geht vielmehr darum, interessante und relevante Daten für die spätere Analyse, eine Visualisierung oder die Verschneidung mit anderen

Teilen der Simulation zu speichern. Entsprechend ist es auch nicht erforderlich, den gesamten Objektzustand zu sichern, sondern ggf. nur einen Teil davon (etwa den CO2-Wert eines Baumobjektes).

5.3.2 Anforderungen an die Speicherung der Daten

Um die Simulation nicht aufzuhalten, darf das Speichern der Daten nicht zu lange dauern. Dabei ist es egal, ob es sich um Objekte oder reine Daten handelt. Es sollte dazu auch möglich sein, eine modellspezifische Kompression der Daten zu implementieren. Weiterhin dürfen Daten nicht verloren gehen und müssen jederzeit eindeutig einem Zeitpunkt aus der Simulation zugeordnet werden können.

5.3.3 Anforderungen an das Lesen von Daten

Beim Lesen der Daten ist zu unterscheiden um welche Art es sich handelt. Die persistierten Objekte sollten möglichst einfach und effizient wieder geladen werden können. Für die Simulationsdaten in beliebiger Struktur muss es möglich sein, auch entsprechend beliebige Abfragen durchzuführen. Insbesondere ist eine Verschneidung der Daten in verschiedenen räumlichen und zeitlichen Dimensionen erforderlich. Das Lesen muss möglichst performant gestaltet werden, da ggf. die Ausführung des nächsten Simulationsschrittes davon abhängt.

Für eine spätere Analyse der durchgeführten Simulation sollen die Daten ebenfalls unter Berücksichtigung der o.g. Anforderungen zur Verfügung stehen. Die Geschwindigkeit ist hier nicht mehr von so entscheidender Bedeutung, sollte sich aber natürlich dennoch in einem üblichen Rahmen bewegen.

5.3.4 Skalierung der Daten

Die verwendeten Daten werden zwangsweise eine teils stark divergierende Skalierung aufweisen. So sind der Simulation entstammende Daten voraussichtlich gemäß der Simulationsschritte skaliert, wohingegen GPS Positionsdaten von sich durch ein Gebiet bewegenden Tieren, in einer wesentlich gröberen Skalierung vorliegen können. Dies gilt so oder so ähnlich vermutlich für alle Messwerte, die von Außerhalb in eine Simulation integriert werden sollen.

5.3.5 Datacube & Sternschema

Wie Thiel-Clemen (2013) zeigt, lassen sich inhomogen skalierte Daten recht elegant mit Hilfe einer Datenbank mit Sternschema, einem sogenannten Datacube, zusammenführen.

Das Sternschema erfordert eine relationale Datenbank und sieht vor, um eine zentrale Faktentabelle herum, beliebig viele Dimensionstabellen zu definieren. Die Dimensionstabellen enthalten jeweils eindimensionale Daten aus der Fachdomäne. Die Einträge in der Faktentabelle referenzieren eine beliebige Menge der Dimensionstabellen und speichern so, unter einem aus den Fremdschlüsseln der Dimensiontabellen zusammengesetzten Primärschlüssel, Faktenwissen zu einer beliebigen Zusammenstellung der Dimensionsdaten (vgl. Abbildung 2.1).

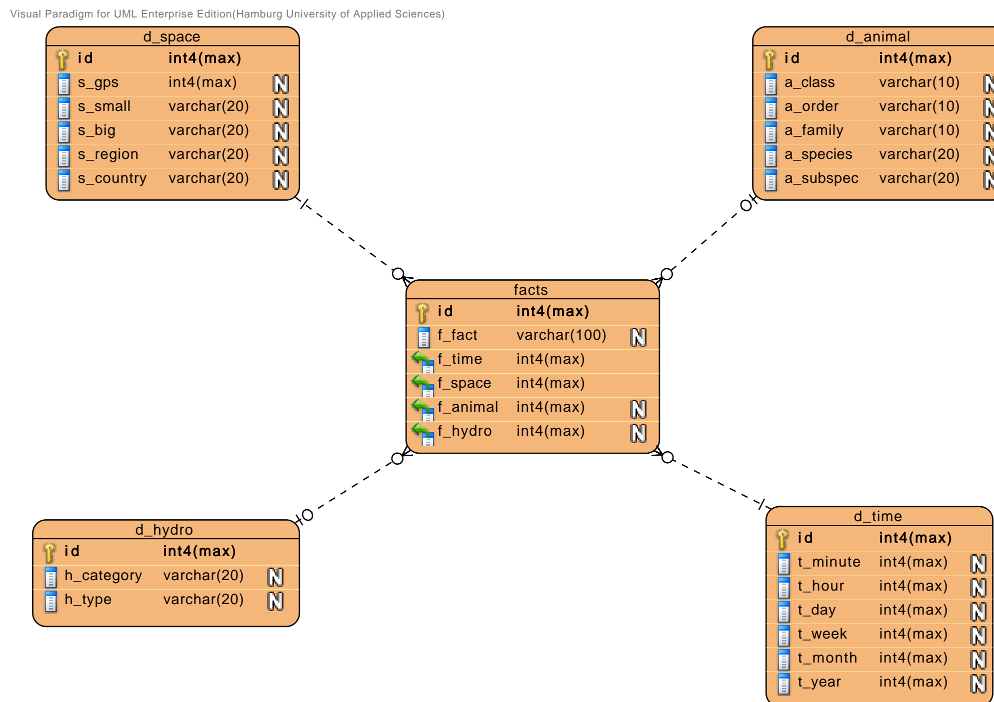

Abbildung 5.6: Exemplarisches Sternschema (Quelle: (Thiel-Clemen, 2013))

Anfragen an eine derartig gestaltete Datenbank sind üblicherweise sehr schnell, da ihre JOIN-Tiefe nie größer als 1 ist. Allerdings setzt dies voraus, dass das Sternschema strikt eingehalten wird, die Dimensionstabellen also keine weiteren Untertabellen mehr referenzieren. Der schreibende Zugriff hingegen ist vergleichsweise langsam, da für jeden Eintrag die entsprechenden Dimensionstabellen gefüllt und die Fremdschlüssel referenziert werden müssen.

Der langsame Schreibzugriff legt nahe, zwischen die Simulationsapplikation und den Datacube eine zusätzliche Datenbank als Cache einzubauen, welche die Daten aus der Simulation

zunächst entgegen nimmt und anschließend in den Cube überführt. Lesend wäre dann eine 2-stufige Abfrage möglich, bei der man zunächst im Cube nach den Daten sucht und nur, falls die Daten dort noch nicht vorhanden sind, auf den Cache zugreift.

5.3.6 Untersuchung verschiedener DBMS als Cache

Die Verwendung einer zusätzlichen Datenbank als Cache stellt spezielle Anforderungen an die DBMS-Lösung. Diese muss In-Memory lauffähig sein und eine sehr gute Lese- wie Schreibperformanz aufweisen. Zusätzlich muss beachtet werden, dass die Daten aus der Cache-DB in das Sternschema des Datacubes überführt werden (können) müssen. Dazu ist ein gesonderter ETL-Prozess[2] nötig, der ggf. eine Übersetzung durchführt. Möchte man die Übersetzung vermeiden, muss auch die Caching-DB bereits ein Sternschema aufweisen.

PostGRE-SQL

PostGRE ist ein 1989 von Michael Stonebreaker entwickeltes, freies, objektrelationales DBMS. PostGRE unterstützt Transaktionen, ist ACID konform und entspricht weitestgehend dem SQL-Standard ANSI-SQL von 2008[3]. PostGRE erfreut sich großer Beliebtheit in der OpenSource Gemeinschaft und wird durch viele Erweiterungen um sinnvolle Funktionen bereichert. Eine dieser Erweiterungen ist PostGIS, was sich für dieses Projekt als sehr nützlich herausstellen sollte, da damit direkt in der Datenbank GIS Daten gespeichert und entsprechende Operationen - etwa A*, Flächenberechnungen etc. - durchgeführt werden können.

PostGRE arbeitet als festplattenpersistentes Datenbanksystem nicht In-Memory, lässt sich aber durch entsprechende Parametrisierung und die Verwendung eines tmpfs-Laufwerkes dahingehend anpassen, dass es sich einer In-Memory Datenbank geschwindigkeitsmäßig annähert.

Couchbase

Couchbase ist eine dokumentenbasierte NoSQL Datenbank, die aus der Fusion von CouchDB und Membase entstanden ist. Durch den dokumentenbasierten Ansatz lassen sich Daten einfach in Form von JSON Strings abspeichern, dies gilt auch und ganz besonders für Objekte. Diese können mit einer Bibliothek wie z.B. GSON [4] automatisch in das JSON-Format umgewandelt und andersherum auch wieder daraus rekonstruiert werden.

[2] Extract, Transform, Load. Ein Prozess der Daten aus einer oder mehreren Datenquellen in eine gemeinsame Zieldatenbank überführt.

[3] http://www.postgresql.org/docs/current/static/features.html

[4] http://code.google.com/p/google-gson/

Couchbase zeichnet sich, wie die allermeisten NoSQL Datenbanken, durch eine exzellente Verteilbarkeit und damit eine gute Skalierbarkeit aus. Die Datenbank wird mittels Sharding innerhalb eines Clusters verteilt, wobei, anders als bei den SQL Datenbanken, die Daten nicht repliziert, sondern schlicht auf verschiedene Knoten verteilt werden.

NoSQL typisch bietet Couchbase Views an, mittels derer die gespeicherten Dokumente im Sinne einer Abfrage zusammengestellt werden können. Zur Definition eines Views wird das Map/Reduce Pattern verwendet (Yang *et al.*, 2007).

Die Datenbank bietet zwei Betriebsmodi an, die man für die 'Buckets' genannten Datencontainer einstellen kann. Im Standardmodus namens 'Couchbase' werden eingehende Daten zunächst im RAM aufgenommen und nach und nach auf die Festplatte persistiert. Sind die Daten geschrieben, werden diese indiziert und anschließend die definierten Views geupdatet. Auf diese Weise verfügt Couchbase über eine hervorragende Schreibperformanz.

Der alternative Modus heißt 'Memcached' und verzichtet auf die Persistenz der Daten, arbeitet also komplett im RAM. Damit einhergehend ist es hier auch nicht möglich, Views oder Indizes anzulegen. Gedacht ist diese Variante vor allem für das optimierte Auslesen häufig verwendeter Daten. Als Beispiel sei das Caching hochfrequentierter Daten aus einer relationalen Datenbank im Webumfeld genannt. Durch den Memcache muss nicht jedes mal eine eigene Datenbankanfrage durchgeführt werden, wenn ein Nutzer bspw. auf einen Artikel zugreifen möchte.

Möchte man Daten auslesen, so funktioniert dies in beiden Modi über den Key, der beim Speichern des Dokumentes angegeben wurde. Man erhält in diesem Fall das komplette JSON Dokument zurück. Möchte man indes differenzierter auf die Daten zugreifen, ist dies nur im 'Couchbase'-Modus möglich. Hier lassen sich Views definieren, über die man sich die Daten in nahezu beliebiger Weise zusammenstellen lassen kann.

Die Schreibgeschwindigkeit von Couchbase ist sehr gut. Die Lesegeschwindigkeit ist ebenfalls nicht schlecht. Problematisch wird es hier nur insofern, als dass die Aktualisierung eines Views stets eine gewisse Latenz hat, nachdem neue Daten in die Datenbank geschrieben wurden. Diese Latenz setzt sich zusammen aus der Abarbeitung der Queue, also der Zeit bis neue Datensätze aus dem Arbeitsspeicher auf die HDD persistiert wurden und einer gewissen Verzögerung bis die Aktualisierung der Views getriggert wird, sowie der Aktualisierung selbst.

Durch die Notwendigkeit, die Daten zunächst persistieren zu müssen, verspielt Couchbase seine Nützlichkeit als Cache-DB. Zwar könnte man bei Verwendung des 'Memcached'-Modus die Keys der gespeicherten Daten an den ETL-Prozess übermitteln, es ist dann jedoch fraglich, ob der Aufwand dieser Verwaltungsarbeit in Summe nicht am Ende länger dauert als die Daten direkt in den Datacube zu schreiben.

Man könnte statt üblicher HDDs schnellere SSDs oder gar PCIe-SSDs benutzen, um die Persistierung entsprechend zu beschleunigen, da es bei der Cache-DB nicht darum geht, größere Datenmengen über einen längeren Zeitraum zu speichern. Leider ergeben sich dann Probleme bei der View-Berechnung, da man aufgrund des o.g. Prozesses nie sicher sein kann, auch wirklich den aktuellen Stand zu sehen. Die Verzögerung bis zur Neuberechnung eines Views kann bis zu mehreren Sekunden (<= 3) betragen. Zwar lässt sich eine Neuberechnung beim Lesezugriff über die API forcieren, aber auch hier ist dieser Mehraufwand in Summe vermutlich wieder langsamer als direkt in den Cube zu schreiben.

Abgesehen von der Problematik, die Daten schnell genug auszulesen, ergibt sich ein weiteres Problem, wenn man die 2-stufige Abfrage durchführen möchte. Die Anfragen an den Datacube werden üblicherweise mittels eines Star-Joins gestellt. Dieser in der SQL Welt recht einfache JOIN über die Faktentabelle und mehrere Dimensionstabellen, gerät in der NoSQL Welt zur Unmöglichkeit, da es hier keine JOINS gibt. Der empfohlene Workaround sieht so aus, die Daten mittels eines Views auszulesen und den JOIN in einer eigenen Applikation durchzuführen. Das schließt Couchbase schließlich und endgültig als Cachinglösung aus.

HBase, Hypertable

Die spaltenbasierten Datenbanksysteme sind Reimplementierungen von Googles BigTable und zeichnen sich ebenfalls durch eine hervorragende Verteilbarkeit & Schreib/Lese-Performanz sowie die Verwendung von Map/Reduce (Yang *et al.*, 2007) aus. Dabei wird vor allem Wert auf Analyse- und Suchanfragen gelegt, was bei der Herkunft von Google nicht weiter verwundert. Hypertable wird dabei generell als bessere und schnellere, weil leichtgewichtigere, Alternative zu HBase gehandelt. HBase und Hypertable verfügen beide nicht über die Möglichkeit JOINs durchzuführen und sind durch ihre Ausrichtung auf Analyseanfragen u.Ä. keine Kandidaten für den benötigten DB-Cache.

MySQL Cluster, EXASolution und andere kommerzielle Lösungen

Neben den OpenSource Datenbanken gibt es natürlich noch die großen, kommerziellen Produkte. Zwei davon habe ich mir ein bisschen näher angesehen. EXASolution von der deutschen

Firma Exasol aus Nürnberg [5] ist eine In-Memory Lösung und hat den aktuellen TPC-H Benchmark in den Kategorien 100 GB, 300 GB, 1 TB , 3 TB und 10 TB gewonnen[6]. Der TPC-H Benchmark ist ein entscheidungsunterstützender Benchmark, der aus einer Sammlung von geschäftsorientierten ad-hoc Anfragen und konkurrierenden Datenmodifikationen besteht (siehe http://www.tpc.org/tpch/default.asp).

MySQL Cluster ist die kommerzielle Variante der bekannten und beliebten MySQL Community Edition und ist im wesentlichen eine spezielle knotenbasierte Storageengine. Sie zeichnet sich dadurch aus, dass sie in einer Shared-Nothing-Architektur läuft und damit horizontal skaliert, laut Hersteller eine Verfügbarkeit von 99,999 % aufweist, diverse SQL und auch NoSQL APIs anbietet, 100 % ACID-konform ist und ausserdem dank In-Memory Caching ebenfalls eine sehr hohe Performanz vorzuweisen hat.

Obwohl beide Lösungen ziemlich gut auf das Problemfeld zu passen scheinen, habe ich sie aufgrund ihres kommerziellen Hintergrundes nicht weitergehend getestet. Eine Jahreslizenz von MySQL Cluster kostet beispielsweise 10.000 USD für einen Server![7] Da die in dieser Arbeit beschriebene Simulationslösung aber vor allem auch in Afrika in einem sozio-ökonomischen Forschungsumfeld zum Einsatz kommen soll, ist die Einsparung von Kosten von hoher Priorität. Aus diesem Grund werde ich mich auf die kostenfrei verfügbaren OpenSource Lösungen konzentrieren.

VoltDB

VoltDB[8] ist die kommerzielle Implementation des 2007 und 2008 von Michael Stonebreaker, Robert Kallman et. al. vorgestellten H-Store Konzeptes (Stonebraker *et al.*, 2007) (Kallman *et al.*, 2008). VoltDB liegt in einer kostenfrei nutzbaren Community Edition vor, für Herstellersupport ist entsprechend zu bezahlen.

H-Store wurde aus der Erkenntnis heraus entwickelt, dass bestehende RDBMS aufgrund ihres Alters, ihrer Herkunft und den Bedingungen und Anforderungen, aus denen heraus sie entstanden sind, mittlerweile veraltet sind. Das gilt laut (Stonebraker *et al.*, 2007) sowohl für die Art und Weise, wie man RDBMS benutzt, als auch wofür. Laut Stonebreaker macht der Ansatz "one size fits it all" bei den heute verfügbaren RDBMS-Alternativen für spezielle Anwendungsszenarien keinen Sinn mehr, da die spezialisierten Datenbanksysteme in der Regel

[5] www.exasol.com
[6] Ergebnisse auf www.tpc.org
[7] http://www.mysql.com/products/
[8] www.voltdb.org

eine oder zwei Größenordnungen schneller seien. Auch (Sadalge & Fowler, 2013) postuliert unter dem Begriff "Polyglot Persistence" die Verwendung mehrerer spezialisierter Datenbanken innerhalb einer Anwendung für ihren jeweils optimalen Einsatzzweck.

Mit dem H-Store Konzept möchten Stonebreaker et. al nun zeigen, dass auch die letzte Bastion der RDBMS, die OLTP Business Applikationen, abgelöst und durch einen neuen Ansatz ersetzt werden können.

Der Clou bei VoltDB ist, dass es zwar In-Memory, 100 % ACID-konform, relational und sql-basierend ist, aber dennoch in einer Shared-Nothing-Architektur horizontal skaliert, wie man dies für gewöhnlich nur von NoSQL-Datenbanken gewohnt ist.

Erreicht wird das durch die konsequente Verwendung von Stored-Procedures in Java und die Ausnutzung des Umstandes, dass viele Anfragen nur wenige oder eine Tabelle, oder gar eine einzelne Zeile betreffen. Anders als bei herkömmlichen DBMS werden bei VoltDB das Datenbankschema sowie die möglichen Anfragen bereits vor der Laufzeit durch den Entwickler definiert und anschließend beim Starten der Datenbank in einen sogenannten Katalog kompiliert. Aus der Client-Applikation lassen sich dann die in der Datenbank definierten Prozeduren ansprechen.

Die Verwendung von Stored-Procedures umgeht, so Stonebreaker, den Umstand, dass man bei der Verwendung von bswp. ODBC für jede Teiloperation, die vom Client aus gesteuert wird, einen Nachrichtenaustausch zwischen Client und Server hat, die Latenz also eine sehr große Rolle spielt. Bei ODBC/JDBC obliegt die gesamte (An)Steuerung der Datenbank dem Client, so dass diese Kommunikation sehr intensiv ausfällt. Liegt die Logik für die DB-Operationen in der Datenbank selbst, so benötigt man lediglich eine Nachricht als Anfrage und eine als Antwort und minimiert somit die Auswirkung der Netzwerklatenz. Bei der Definition des Datenbankschemas kann der Entwickler angeben, wie VoltDB die Tabellen und Prozeduren partitionieren soll. Für eine einfache INSERT-Prozedur lässt sich bspw. anhand des Primärschlüssels partitionieren. Diese Partitionierung führt dann dazu, dass VoltDB für jeden Aufruf der INSERT-Prozedur einen eigenen Thread auf einem eigenen Kern des Serverrechners starten und die Aufrufe wirklich parallel abarbeiten kann. Prozeduren, die derartig funktionieren, werden "single-partition procedures" genannt. Prozeduren, die über mehrere oder gar alle Tabellen laufen, heissen "multi-partition procedures" und blockieren dann auch entsprechend die Ausführung anderer Queries auf den genutzten Tabellen.

Diese Partitionierung ermöglicht somit eine Skalierung durch einfaches Hinzufügen von Hardware. Folgerichtig lässt sich VoltDB im Rahmen einer Shared-Nothing-Architektur auch

sehr einfach auf mehrere Rechner (Nodes) verteilen. Für schnelleren Lesezugriff lassen sich ausgewählte Tabellen über die Nodes replizieren.

Ein sich aus diesem Ansatz ergebendes Problem ist, dass man VoltDB derzeit neustarten muss, um etwas an den Schemadefinitionen zu verändern, oder neue Prozeduren hinzuzufügen. Der Hersteller verspricht hierfür aber bereits an einer Lösung zu arbeiten.

Weitere, möglicherweise relevante Einschränkungen finden sich beim Lesezugriff. So sind derzeit lediglich implizite, innere Joins möglich und die Ergebnisse einer SELECT-Anfrage sind limitiert auf 50 MB.

Die von VoltDB versprochene Performanz gegenüber klassischen RDBMS wird von Benchmarks, die sich im Internet finden lassen,[9] durchaus bestätigt. Der Umstand, mit VoltDB eine performante, skalierbare und In-Memory laufende Datenbank zu haben, in der man ausserdem direkt das für die Information Integration angestrebte Sternschema umsetzen kann, macht diese Datenbank ausgesprochen spannend für den Einsatz als DB-Cache oder sogar als alleinige Speicherlösung.

5.3.7 Performance Test von PostGRE, Couchbase und VoltDB

Nachfolgend ein Benchmarktest von drei der oben besprochenen Datenbanken. Getestet werden PostGRE-SQL, PostGRE-SQL mit Performance Optimierungen, Couchbase als Vertreter der NoSQL Datenbanken und VoltDB, da es sich als aussichtsreicher Kandidat für den DB-Cache präsentiert hat.

Testkonfiguration

Der Test wird zwischen zwei Rechnern durchgeführt, die über ein Gigabit-LAN miteinander verbunden sind. Die Spezifikationen der Rechner lautet wie folgt:

Client MacBook Pro, 2,3 Ghz i7 QuadCore + HT, 8GB Ram, 240GB SSD, Mac OS X 10.8.4

Server 3,2 Ghz i5 Quadcore, 8 GB Ram, 240 GB SSD, Win 7 64 bit bzw. Ubuntu 10.04 LTS 64bit

Die Datenbanken wurden sämtlich mit einer Instanz und in ihren Standardeinstellungen getestet. Einzig bei PostGRE wurden einige Optimierung nach Vorschlägen aus der PostGRE Performance Mailinglist[10] [11] vorgenommen. Zu beachten ist, dass PostGRE in der optimierten

[9] www.mysqlperformanceblog.com
[10] http://rhaas.blogspot.de/2010/06/postgresql-as-in-memory-only-database_24.html
[11] http://www.postgresql.org/list/pgsql-performance/

Variante nicht mehr crash-safe ist, was für eine Caching-DB aber auch nicht notwendig sein sollte.

Das Datenbankschema ist ein Sternschema mit 10 Dimension und einer Faktentabelle. Jede Dimension speichert einen Zeitstempel, eine funktionale ID sowie einen Wert. Die Faktentabelle hält zu jeder Fremdschlüsselkombination aus den Dimensionstabellen einen weiteren Wert.

Die Testclients sind kleine Java Programme, welche jeweils 20 Threads erstellen und die einzufügenden Elemente gleichmäßig aufgeteilt auf diese Threads möglichst schnell zu schreiben versuchen. Gemessen wird die Zeit vom Start dieser Threads bis zum bestätigten, erfolgreichen Durchlauf des letzten Threads.

PostGRE-SQL und Couchbase liefen unter Windows 7 Professional 64-bit, VoltDB unterstützt derzeit nur Linux, Mac OS X und andere Unix Derivate. Daher wurde VoltDB unter Ubuntu 12.04 LTS 64bit getestet.

Für den Test von Couchbase kam der JSON Parser/Generator GSON[12] zum Einsatz.

Testdaten

Die Testdaten werden als Java Objekte erzeugt. Die Struktur dieser Objekte ist sehr einfach und wird in Abbildung 4.4 dargestellt. Die Dimensionswerte speisen sich aus einer im Konstruktor erzeugten UUID, die fachliche ID ist eine eigene UUID um eindeutig zu sein. Die Erzeugung einer UUID ist rechentechnisch recht aufwendig, weshalb zur Performancesteigerung für die Dimensionswerte stets die gleiche UUID genutzt wird.

[12] http://code.google.com/p/google-gson/

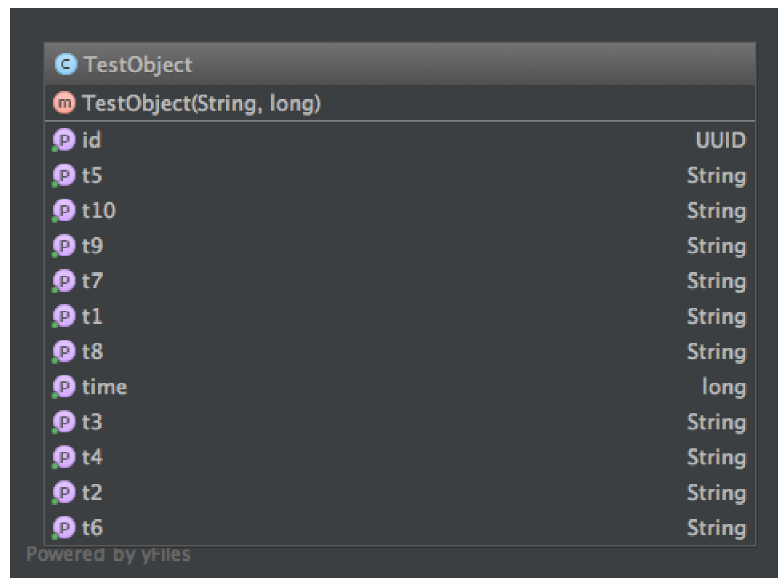

Abbildung 5.7: TestObject Klasse

Ergebnisse

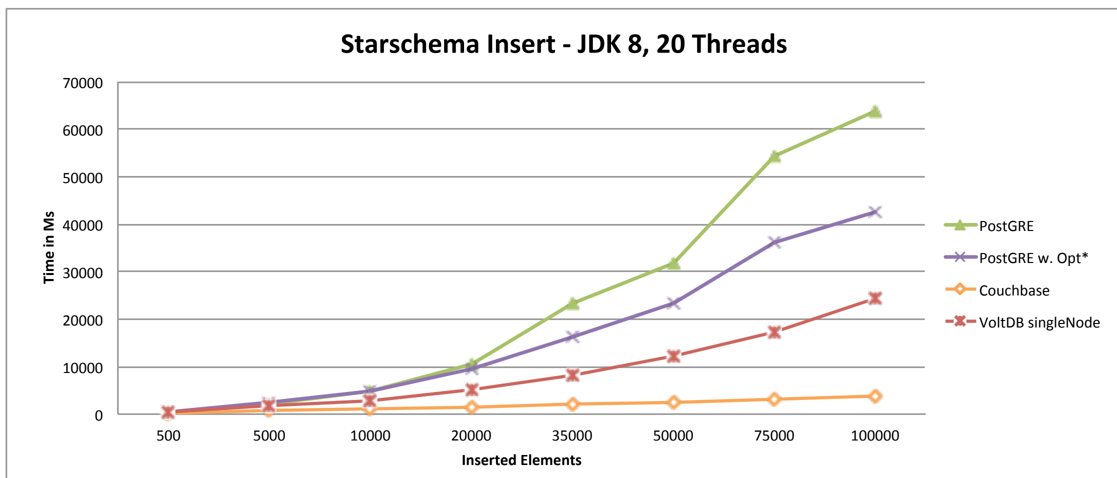

Abbildung 5.8: Ergebnisse Datenbank Benchmark

Für kleine Elementmengen <= 5000 liegen die Datenbanklösungen nur verschwindend gering auseinander. Genau betrachtet zeigt sich bei 500 Elementen sogar, dass der Overhead der Initialisierung bei VoltDB offenbar stärker zu Buche schlägt, als bei PostGRE-SQL. Schon ab 5000 Elementen steigt die Dauer der Inserts in beiden PostGRE-Varianten im Vergleich zu VoltDB und Couchbase deutlich an. Ab 20000 Elementen steigt die Standard PostGRE Variante

sprunghaft an, während Couchbase und VoltDB eine nahezu lineare Steigerung der Dauer aufweisen. Die optimierte PostGRE Variante ist im Schnitt ca. 19 % schneller als die normale Instanz.

VoltDB kann mit einer durchschnittlich 35 % besseren Performanz aufwarten, als die optimierte PostGRE-DB und ist im Vergleich zur normalen PostGRE-DB sogar im Schnitt 44 % schneller.

Auffällig ist die extrem hohe Performanz von Couchbase, allerdings ist hier zu beachten, dass kein Star-Insert stattfindet, sondern lediglich eine Persistierung der Objekte im JSON-Format.

Auswertung & Fazit

Couchbase fällt als Cache-Lösung aus den in der Datenbankbesprechung genannten Gründen zwar raus, besticht im Test aber dennoch durch seine enorme Schreibperformanz von komplett persistierten Objekten. Daher ist Couchbase ein spannender Kandidat für eine Schnappschussfunktion innerhalb des zu erstellenden Simulationssystem.

PostGRE bietet im Test eine ordentliche Geschwindigkeit beim persistenten Schreiben der Sternschemadaten, wird von VoltDB aber schon mit einer einzelnen Instanz um bis zu 44% geschlagen. Da die Sternschema-Inserts sich im Sinne der VotlDB komplett als single-partition Procedures ausdrücken lassen, sollte sich die Performanz der VoltDB linear mit der Anzahl der zur Verfügung stehenden Prozessoren noch steigern lassen. Ein Test auf einem vorhandenen Simulationsrechner mit 24 logischen Kernen steht dazu noch aus. Ein zusätzliches Plus an Geschwindigkeit verspricht zudem die per Sharding durchzuführende Verteilung der Datenbank auf mehrere Rechner, da das Simulationssystem im Produktivbetrieb ebenfalls verteilt sein wird. Dank des Shardings kann somit jeder Simulationsknoten mit einem beliebigen VoltDB-Knoten sprechen, was die Netzwerkauslastung und auch die Auslastung der einzelnen Maschinen insgesamt optimieren sollte. Ein persönlicher Test hierzu steht ebenfalls noch aus, jedoch zeigen Tests aus dem Internet bereits, wie gut VoltDB skaliert[13].

VoltDB persistiert die Daten nicht auf die Festplatte, ist aber auf eine entsprechende Rendundanz ausgelegt, um auch den Absturz einzelner Knoten ohne Datenverlust überstehen zu können. Als persistente Datenbank würde sich dann PostGRE-SQL anbieten. Aufgrund der identischen Datenbankschemata ließe sich eine Überführung hier über einen einfachen ETL-Prozess herstellen.

[13] http://www.mysqlperformanceblog.com/2011/02/28/is-voltdb-really-as-scalable-as-they-claim/

Architektonisch spricht Vieles für und bislang Nichts gegen VoltDB als DB-Cache. Da VoltDB SQL die Information Integration über den Datacube ebenfalls komplett abbilden kann, lässt man im Sinne der Gesamtperformance des Systems VoltDB auch als alleinige Datenbanklösung nutzen. Die Verwendung von PostGRE-SQL wäre dann optional und erst von Nöten, wenn man entweder eine wirklich festplattenpersistente Lösung benötigt, oder aber eine der vielen weiteren Funktionen von PostGRE nutzen möchte bzw. muss. Dazu gehören etwa Dinge wie PostGIS zur erheblich komfortableren Verwaltung von Geodaten oder das Stellen fortgeschrittenerer Abfragen an die Datenbank.

5.4 Layer als Plugins

5.4.1 Überblick

Als unmittelbare Konsequenz aus dem SoC-Ansatz[14] der Layer, ergibt sich die Integration der Layer als Plugins, die beliebig zusammengesteckt werden können. Das setzt eine solide Plugin-Architektur voraus.

Die Implementierung eines Plugins sollte dabei möglichst einfach sein und vorzugsweise keine Konfigurationsdatei in XML o.ä. erfordern. Das Laden von Plugins sollte ebenfalls einfach sein und bspw. aus einem Plugin-Ordner gebündelt möglich sein. Idealerweise bietet die Plugin-Architektur auch einen Weg zur transparenten Inter-Plugin- und damit zur Inter-Layer-Kommunikation an.

Da in der nahen Zukunft nicht davon auszugehen ist, dass es eine Vielzahl an Plugins verschiedener Anbieter geben wird, kann auf eine Plugin-Repo Struktur mit weiteren umfangreichen Services zunächst verzichtet werden.

5.4.2 Erfahrungen aus anderen Projekten

Aus dem Projekt D-MARLA[15], welches ich zusammen mit Kommilitonen als verteiltes System für Lernende Agenten entwickelt habe, traten im Zusammenhang mit Java diverse Probleme beim Laden von Plugins auf. So ist insbesondere bei angestrebter Serialisierung von Klassen aus den Plugins darauf zu achten, die Plugins alle mit dem gleichen Classloader zu laden, da es ansonsten zu komplexen Problemen mit der JVM kommt. Als zu diesem Zeitpunkt unlösbares Problem, stellte sich das dynamische Laden und vor allem Entfernen von Plugins zur Laufzeit heraus.

[14] Separation of Concerns
[15] D-MARLA auf Github

Ein Problem tritt hierbei auf, wenn man zwei Plugins A und B lädt, wobei B ein Interface von A benutzt. Ruft B nun eine Methode auf A auf und A wird anschließend von einem anderen Prozess gegen eine alternative Implementierung A' ausgetauscht, werden alle von B referenzierten Zustände in A verloren gehen. B denkt nun es habe die Methode auf A bereits aufgerufen und erwartet auch das Verhalten des alten A, tatsächlich existiert aber ein A' mit neuem Zustand. Daher ist ein Hot-Pluggen in der aktuellen JVM nicht möglich.

Dieses Problem besteht weiterhin und wird vermutlich frühestens durch die Umsetzung des Projektes JIGSAW[16] voraussichtlich im JDK 9 gelöst werden.

5.4.3 Frameworks im Vergleich

Die Internetrecherche ergab drei derzeit verbreitete und genutzte Plugin-Frameworks für Java.

OSGi

OSGi ist eine von der OSGi Alliance[17] spezifizierte und referenzimplementierte dynamische Softwareplattform, die es erlaubt, Softwarekomponenten, sog. Bundles, dynamisch zur Laufzeit in ein und derselben JVM zu laden und wieder zu entfernen. Ein Teil der Plattform ist eine Registrierung, über welche sich die Komponenten finden lassen. Anwendungsbereiche finden sich u.a. in der Fahrzeugtechnik, im Mobilfunk, in der Netzwerktechnik (auch Heimvernetzung) oder auch zur Unterstützung von J2EE Architekturen.

JPF

JPF ist das Java Plugin Framework[18]. Es bietet Features wie Lazy-Loading für Plugins, d.h. Plugins verbrauchen nur minimale Ressourcen bis sie tatsächlich gebraucht werden, Plugin-Dependency Checks um Abhängigkeiten zwischen Plugins entsprechend auflösen zu können und JPF soll sogar das Problem des Hot-Pluggings lösen können. Leider gab es im JPF Projekt seit 2007 keine Aktivitäten mehr, so dass das Projekt kein besonders attraktiver Kandidat für eine auf längere Sicht ausgelegte Architektur ist. Dazu kommt, dass die Entwicklung von Plugins mit JPF zwingend Dateien mit YAML Definition zu den Plugins voraussetzt, was einer der Anforderungen an die Einfachheit der Benutzung widerspricht.

[16] Projekt JIGSAW auf java.net
[17] http://www.osgi.org
[18] http://jpf.sourceforge.net/

JSPF

JSPF[19] (Java Simple Plugin Framework) ist ein von Ralf Biedert am DKFI[20] erstelltes Plugin-Framework, um die Handhabung von Plugins in Java Projekten zu vereinfachen.

Herausragende Merkmale von JSPF sind die Nutzung von Java-Annotationen, Typsicherheit durch die Benutzung von Generics sowie das Laden von Plugins mit nur zwei Zeilen Code. Ein kleiner Prototyp des Autoren bestätigt diese Merkmale und lässt erkennen, wie einfach sich eine Plugin-Architektur mit JSPF umsetzen lässt. Das weiter oben beschriebene Problem der Notwendigkeit den gleichen Classloader zu nutzen, wird durch JSPF bereits gelöst. Tatsächlich lässt sich das Verhalten an dieser Stelle auch über Parameter beeinflussen, sodass man den Plugin-Container auch nach eigenen Bedürfnissen einstellen kann.

Zwar wurde JSPF zuletzt im August 2011 geupdatet, doch scheint es weitestgehend feature-ready zu sein und im Issue-Tracker finden sich auch nur - zumindest für dieses Projekt - unwesentliche Tickets. Aufgrund der Einfachheit bei der Benutzung und der Tatsache, dass das DFKI JSPF bereits in Projekten verwendet hat, ist es der Kandidat für die hier entwickelte Architektur.

5.5 Zeitmanagement

5.5.1 Anforderungen an Präzision und Auswertung

Eine hundertprozentige Synchronisation und die Sicherstellung einer kausal korrekten Ereignisabfolge in Verteilten Systemen, ist sehr rechen- und damit zeitintensiv. In einem System, welches diese Dinge sicherstellt, herrscht bezüglich der Nachverfolgbarkeit von Ereignissen sowie der späteren Zustandsbetrachtung zu einem Zeitpunkt X, eine höchstmögliche Präzision. Es ist nun zu entscheiden, in welchem Maße die präzise Auswertung des Simulationsverlaufes entscheidend ist.

Ein System, welches vor allem auf die Live-Visualisierung des Verlaufes bedacht ist - etwa eine Virtual Reality Simulation - ,braucht diesen Grad der Präzision eher nicht und kann ihn vermutlich auch gar nicht erreichen, da in ausreichend schneller Abfolge neue Zustände generiert werden müssen.

In einer systemischen Simulation wie der des Abdoulaye Naturschutzgebietes hingegen, ist eine präzise Auswertung sehr viel wichtiger als eine Ausführung in Echtzeit. Ereignisse und

[19] http://code.google.com/p/jspf/
[20] Deutsches Forschungszentrum für Künstliche Intelligenz: http://www.dfki.de/web

Entwicklungen im Gefüge aus Agenten und Umwelt wollen und müssen im Nachhinein präzise nachvollzogen werden können, um die essentiellen Rückschlüsse zu Validität und Kausalität dieser Ergebnisse zu ermöglichen.

Die immense Menge an Agenten und die damit notwendige Verteilung der Simulation, macht die Umsetzung einer vollständigen Synchronisation aus Performanzgründen dennoch unmöglich. Es muss also eine Balance aus Synchronizität und Geschwindigkeit gefunden werden. Das nächste Kapitel beschreibt einen neuartigen Hybridansatz, der diese Balance gewährleisten sollte.

5.5.2 Hybridansatz

Der hybride Ansatz von Pawlaszczyk & Timm (2007) setzt genau an dem Problem an, dass zu optimistische Synchronisierung zu zu häufigen Rollbacks führt und die strikte konservative Synchronisierung den Nutzen der Parallelisierung nahezu annulliert.

Um dieses Problem anzugehen, kombinieren Pawlaszczyk & Timm (2007) beide Verfahren miteinander in ein klares Kommunikationsprotokoll mit strikten Regeln für die möglichen Situationen.

Die Voraussetzung zur Nutzung des Protokolls besteht einzig darin, die Kommunikation der beteiligten Systemteile (z.B. Agenten) in Form eines klar definierten Nachrichtenprotokolls zu formalisieren und dem hybriden Algorithmus zur Verfügung zu stellen. Dies ist ein sehr nützlicher Umstand im Sinne der Fachdomänenneutralität, da man auf diese Weise nur wenig und auch eher technisch angehauchtes, domänenspezifisches Wissen in eine rein technische Komponente zu stecken braucht.

Das eigentlich Verfahren setzt sich aus fünf Regeln zusammen (nach Pawlaszczyk & Timm (2007)):

wait-for-rule Agent A schickt Agent B eine Nachricht und ausgehend davon, dass es wenigstens eine valide Antwort gibt, gilt, dass Agent A solange blockiert, bis die Antwort von Agent B eingetroffen ist.

execution condition Trifft die Antwort von Agent B auf die Nachricht von Agent A bei Agent A ein, so gilt, dass Agent A die wait-for Sperre aufheben und mit der Berechnung fortfahren kann.

delayed execution condition Wartet Agent A auf eine Nachricht m1 und trifft währenddessen eine Nachricht m2 ein, die nicht im derzeitigen Nachrichtenprotokoll enthalten ist,

so gilt, dass m2 in einer Queue gebuffert wird, bis das Nachrichtenprotokoll abgearbeitet wurde.

deadlock avoidance condition Wartet Agent A auf Nachricht m1, empfängt Nachricht m2 desselben Absenders, wobei m2 nicht zum derzeitigen Nachrichtenprotokoll gehört, und führt dies nicht dazu, dass Agent A ein Rollback durchführen muss, dann gilt: Agent A verarbeitet m2 ungeachtet etwaiger wait-for Sperren.

consideration of cyclic dependencies Wartet Agent A auf Nachricht m1, empfängt Nachricht m2 eines anderen Absenders, wobei m2 zum derzeitigen Nachrichtenprotokoll gehört, und führt dies nicht dazu, dass Agent A ein Rollback durchführen muss, dann gilt: Agent A verarbeitet Nachricht m2, da ansonsten das Nachrichtenprotokoll nicht fortgesetzt werden kann.

Die letzte Regel ist nach Pawlaszczyk & Timm (2007) für den eher unwahrscheinlichen Fall gedacht, dass der Kommunikationspartner innerhalb eines Dialoges plötzlich durch einen anderen ersetzt wird. Würde man die Nachricht m2 nicht verarbeiten, hinge das System fest.

Entsprechend ergibt sich für das Senden und Empfangen von Nachrichten ein Ablauf wie in Abbildungen 5.9 und 5.10 in Pseudocode gezeigt.

```
send(message msg) to aₙ{...
    if (msg.requiresReply()∉∅) then{
       waitForReply:=true;       //[def.4]
       MsgTemplate = msg.createReplyTemplate();
    }...
}
```

Abbildung 5.9: Senden einer Nachricht, Quelle:Pawlaszczyk & Timm (2007)

```
receive(message msg){...
      if  (LVT(a_i)>msg.Time)∨ msg.isANTIEVENT = true)
           then rollback();           //[def.3]
      else
      if(waitForReply = true) then
          if (MessageTemplate.match(msg)=true) then
          {
               waitForReply:=false;
               process(msg);          //[def.5a]
          }
          else
          if(msg.Sender = MsgTemplate.Sender)
               then process(msg);     //[def.6]
          else
          if(msg.ProtocolID = MsgTemplate.ProtocolID)
               then process(msg);     //[def.7]
          else
               bufferMessage(msg);    //[def.5b]
      }
      else process(msg); //no wait condition was set
}
```

Abbildung 5.10: Empfangen einer Nachricht, Quelle:Pawlaszczyk & Timm (2007)

5.5.3 Probleme bei der Auswertung

Der Hybridansatz von Pawlaszczyk & Timm (2007) erfüllt zwar die Anforderungen bezüglich kausaler Reihenfolge und Synchronizität in Verbindung mit ordentlicher Performanz, lässt aber ein Problem im Zusammenhang mit einer Auswertung im Rahmen der Information Integration außer Acht: Die zeitliche Skalierung. Im Abdoulaye Modell und ähnlichen anderen Modellen, werden oft auch historische Daten genutzt. Die so entstehenden Simulationen stellen dabei häufig eine Vergangenheit nach. Für eine Auswertung im Nachhinein und aus dem Kontext der Information Integration bedeutet dies, dass die Simulationsereignisse bei der Speicherung im Datacube in einer zeitlich distinkten und sinnvollen Anordnung abgelegt werden müssen. Diese Anordnung muss zudem noch zu den bereits vorhandenen historischen Daten passen.

Aber auch bei einer realzeitlich nicht einsortierbaren Simulation besteht das Problem der Linearisierung der in verschiedenen, verteilten Prozessen stattfindenden Systemereignisse. Dieses Problem tritt besonders dann zu Tage, sobald es um nicht voneinander abhängige Ereignisse in verschiedenen Knoten geht. Abhängige Ereignisse in verschiedenen Knoten sind bezüglich ihrer Reihenfolge klar sortierbar. Das stellt bereits der oben angesprochenende Hybridansatz sicher.

Zwischen diesen Ereignissen auftretende, lokale Ereignisse sind allerdings in einem späteren, globalen Linearisierungsprozess nicht ohne Weiteres einsortierbar. Das liegt an der möglicherweise unterschiedlich schnellen Fortschrittsgeschwindigkeit der einzelnen Prozesse.

Die Sinnhaftigkeit dieser Ordnung über die Simulationsereignisse stellt dabei ein Problem dar, welches nach nachvollziehbaren und deterministischen Regeln gelöst werden muss. Zudem sollte die Lösung dieses Problems nicht direkt in der Simulation, sondern davon gelöst in einem, möglicherweise verteiltem, Speicheradapter erfolgen, da der Simulationsablauf als solcher diese Informationen nicht benötigt und so eine weitere Verzögerung bei der Simulationsdurchführung verhindert werden kann.

6 Realisierung

Nachfolgend wird das neu entwickelte System mit Namen RUN beschrieben. Dabei wird zunächst der globale Systementwurf erläutert und dann auf einzelne, wichtige Bestandteile näher eingegangen.

6.1 Entscheidung für Java 8

Es wurde in der WALK Forschungsgruppe darüber diskutiert in welcher Sprache ein mögliches neues System idealerweise implementiert werden sollte. Dem bei WALK genutzten Java 6 standen C# / .NET sowie C++ gegenüber.

C++ war ein spannender Kandidat aufgrund des möglichen Performancevorteils bei gleichzeitiger Erhaltung der potentiellen Plattformunabhängigkeit, wurde aber aufgrund des Mangels an intensiver Erfahrung in der Gruppe sowie des nicht unerheblichen Reimplementierungsaufwandes für WALK wieder verworfen.

Mit C# / .NET hatten viele Diskussionsteilnehmer bereits gute und intensive Erfahrungen gesammelt und auch die speziellen Vorzüge der Sprache bzgl. Lambdas & Delegates gegenüber Java 6 schätzen gelernt. Die eingeschränkte Plattformunabhängigkeit (mittels Mono) fiel allerdings negativ ins Gewicht, weswegen ich mir schließlich die Early Access Preview von Java 8 angesehen habe.

Wie sich heraustellte, bietet Java 8 neben vielen kleinen Verbesserungen eine C# gegenüber quasi gleichwertige Implementierung des Lambdakalküls, eine Map / Reduce API für Collectionklassen sowie eine automatische Parallelisierung etlicher Operationen auf Collections an. Dabei sind die von mir getesteten Versionen (Builds 96 - 106) bereits Feature Ready und laufen absolut stabil.

Diese Vereinigung von wichtigen neuen Spracheigenschaften mit der Plattformunabhängigkeit sowie der bereits gegebenen IDE-Unterstützung durch IntelliJ, haben mich schließlich dazu bewogen RUN in Java 8 umzusetzen.

6.2 Erweiterung von JSPF

Um die verteilten Instanzen der Layer-Plugins per PluginInjection komfortabel nutzen zu können, habe ich die aktuelle Version von JSPF entsprechend erweitert. Der InjectHandler, der die Injektion von per Annotation verlangten Klassen übernimmt, kann nun zusätzlich zu lokal verfügbaren Plugins, auch die Möglichkeiten der RemoteAPI von RipeRMI zu nutzen, um entfernt instantiierte Layer ausfindig zu machen und deren Referenz in die lokale Plugininstanz zu übergeben. Das Verhalten ist so implementiert, dass zunächst nach lokalen Instanzen gesucht und anschließend die Remotesuche durchgeführt wird.

Die notwendigen Änderungen betrafen auch die Architektur von JSPF. Bislang war es so, dass es eine Basisimplementierung gab, die lediglich lokal arbeitete. Die Netzwerkfunktionalitäten aus dem remote-Package waren selbst als JSPF Plugin implementiert und mussten bislang vom Nutzer des Frameworks manuell geladen werden. Durch die Änderungen wird das remote-Plugin nun standardmäßig bei der Initialisierung des PluginManagers mitgeladen und aktiviert, zusätzlich implementiert der PluginManager intern das RemoteAPI Interface, da der PluginManager an alle Framework-Subklassen weitergegeben wird und diese somit dessen Remote-Fähigkeiten ohne weitere Abhängigkeiten nutzen können.

Dank dieser Änderungen müssen Layer-Entwickler nun nur noch die ihr zentrales Interface implementierende Klassen mit @PluginImplementation kennzeichnen und in das entsprechende Plugin-Verzeichnis des LayerContainers legen. Die Verteilung der einzelnen Layerinstanzen übernimmt der SimulationCore, die Kommunikation zwischen den einzelnen Layern wird automatisch durch JSPF und RipeRMI durchgeführt, so dass Layer-Entwickler bequem mit ihren eigenen Interfaces arbeiten können.

6.2.1 Schnittstellenlimitierungen

Wie sich gezeigt hat, beschränkt die Java Classloader Mechanik die Möglichkeit der Schnittstellen zwischen den Layern. So ist es in der derzeitigen Lösung über JSPF mit beliebig verteilten Layern nur möglich Typ-0 Schnittstellen zu benutzen. Schnittstellen eines höheren Typs lassen sich zwar ebenfalls umsetzen, funktionieren dann aber nur in Form eines sog. Multiplugins, welches voraussetzt, dass alle Layer, die derartige Schnittstellen untereinander haben, lokal auf demselben LayerContainer ausgeführt werden.

Wird später eine eigene, JSPF-unabhängige Kommunikationslösung zwischen den Layern

implementiert, gilt diese Limitation natürlich nicht mehr, da die Layer gegenüber JSPF dann als voneinander unabhängige Plugins in Erscheinung treten.

6.3 Systementwurf

RUN ist ein verteiltes System und besteht aus drei Grundbausteinen, die, zumindest theoretisch, beliebig zusammengesetzt werden können.

SimulationController Der SimulationController ist die Steuerzentrale des Simulationssystems. Er verschafft in einer GUI einen Überblick aller vorhandener Knoten und deren Zustände (etwa Systemlast, ausgeführte Simulationen etc.). Der Benutzer kann die auf den verbundenen SimCore's vorhandenen Simulationsmodelle abrufen und deren Layer einsehen. Von hier aus lassen sich die Layer auf verbundene LayerContainer verteilen und die Gesamtsimulation starten.

SimCore Der SimCore ist die Steuerungszentrale einer Simulation. Er enthält einen ModelContainer und stellt darüber Metainformationen zu verfügbaren Modellen sowie die ganz konkreten Implementierungen der den Modellen zugehörigen Layern bereit. Auf Anweisung durch den SimulationController steuert der SimCore die Verteilung, Initialisierung und Ausführung der Simulationen auf den LayerContainern. Derzeit kann ein SimCore lediglich ein Simulationsszenario zur Zeit verwalten.

LayerContainer Ein LayerContainer führt einen oder mehrere Simulationslayer aus, nachdem ihm diese vom SimCore zugewiesen worden sind. Auch enthält der LayerContainer die Serviceschnittstellen zu den eventuell von den Layern genutzten Nachbarsystemen wie etwa dem GIS, dem Datacube etc.

Die nachfolgende Grafik veranschaulicht den Aufbau eines exemplarischen Systems mit je einer Instanz des SimulationControllers und des SimCores sowie drei LayerContainern. Diese Komponenten sind über entsprechende Adapter mit der Datenhaltung (im Bild violett) verbunden.

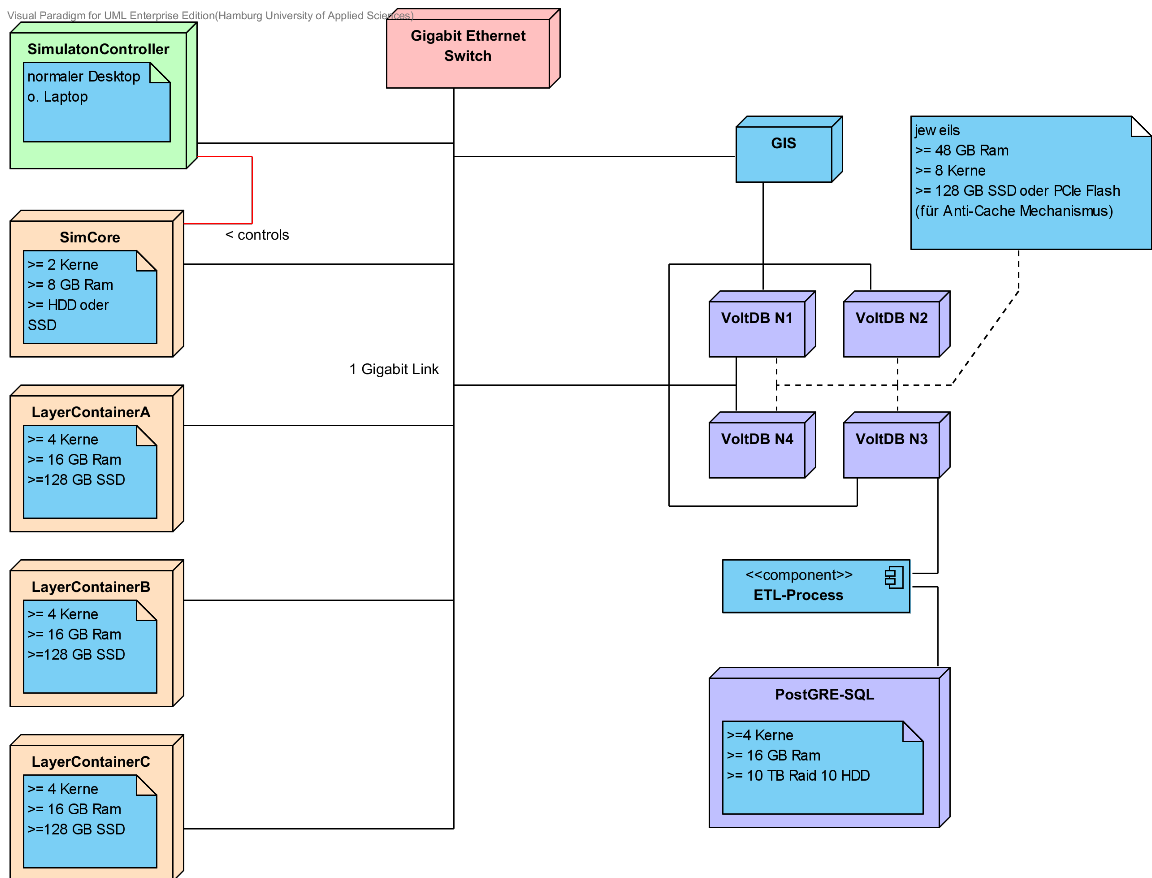

Abbildung 6.1: RUN Systemverteilung

6.4 Gemeinsame Komponenten aller Systemteile

Folgende Komponenten werden in allen Systemteilen als T- bzw. R-Software genutzt.

6.4.1 Kommunikation

Der NetworkAdaptor ist das Produkt einer Projektarbeit aus dem 4. Semester und wurde zu weiten Teilen von meinem Kommilitonen Jason Wilmans implementiert. Er stellt asynchrone, nachrichtenbasierte Kommunikationskanäle über TCP/IP bereit und unterstützt eine automatische (De)Serialisierung der zu übermittelnden Daten. Zum Empfangen kommt ein Publish / Subscribe Pattern zum Einsatz, so dass Nachrichten auch bei mehreren Empfängern stets nur einmal übertragen werden müssen. Zur weiteren Performancesteigerung unterstützt der

Adapter eine je Nachricht selbst anpassbare Serialisierung, so dass man möglicherweise große Datenmengen durch entsprechende Codierung oder Kompression verkleinern kann.

6.4.2 Dezentraler Namensdienst

Die NodeRegistry ist ein lokal laufendes Verzeichnis aller RUN-Knoten im Netzwerk. Dabei wird zu jedem Knoten sein Namen, sein Typ sowie seine Verbindungsinformationen gespeichert. Zudem enthält die NodeRegistry ein MulticastDiscoveryBeacon, mit welchem sich die verschiedenen Knoten im Netzwerk automatisch auffinden und gegenseitig registrieren können. Ist das verwendete Netzwerk nicht multicast-fähig lässt sich der DiscoveryProcess entweder über den NetworkAdapter durchführen oder auch von Hand in die NodeRegistry - etwa aus einer Configdatei - einfügen.

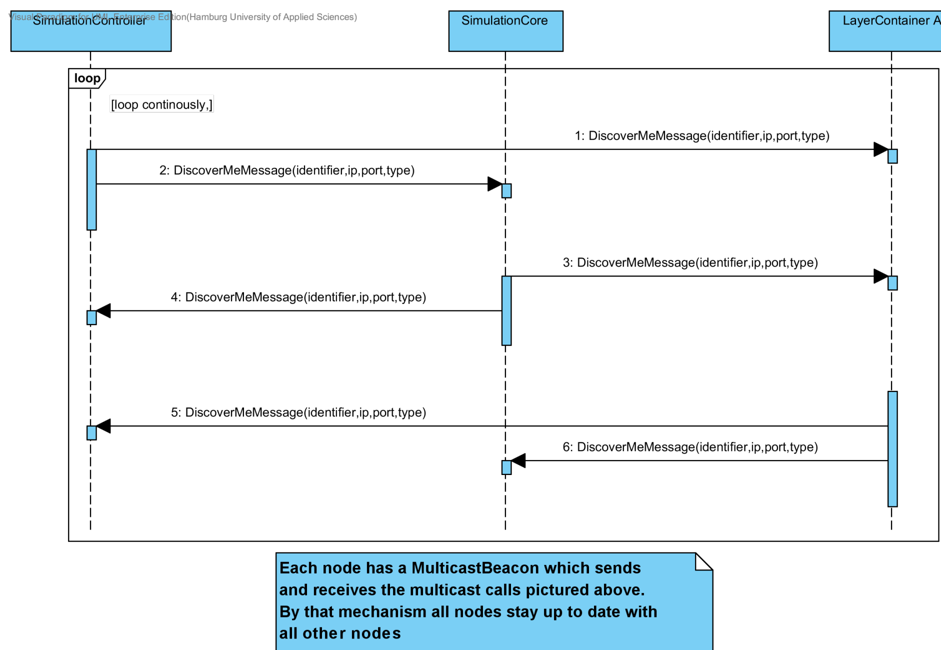

Abbildung 6.2: NodeRegistry Discovery

6.5 SimulationCore Interna

Zentrale Komponente im SimulationCore ist der SimulationManager. Dieser realisiert das Nachrichtenprotokoll mit den anderen Systemkomponenten (siehe 6.9) und nutzt den ModelContainer um Modelle bereitzustellen oder Metainformationen zu diesen abzufragen. Der Modelcontainer selbst enthält das ModelRepository zum Speichern der Modelle sowie einen

Embedded-Webserver über welchen die Layerimplementierungen in Form von JAR Dateien im späteren Verlauf des Initialisierungsprozess durch den jeweiligen Layercontainer abgerufen werden.

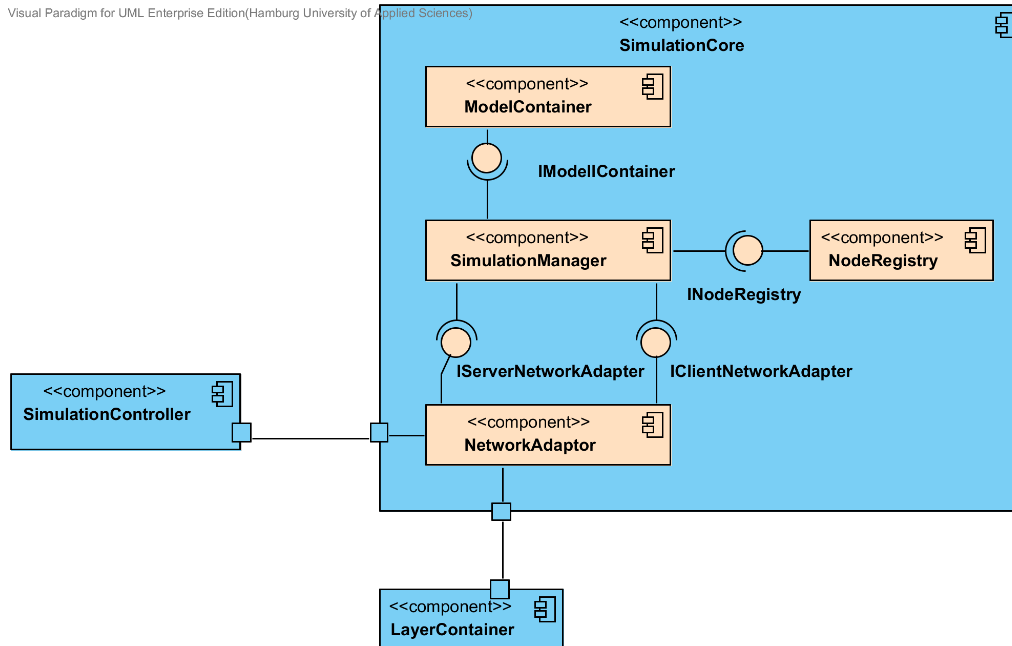

Abbildung 6.3: SimulationCore Komponenten

6.6 LayerContainer Interna

Herzstück des LayerContainers ist der LayerContainerController. Hier ist das Nachrichtenprotokoll für Initialisierungs- und Ausführungssteuerung implementiert. Der LayerContainerController nutzt seinerseits den NetworkAdapter sowie die NodeRegistry um mit anderen RUN-Knoten zu kommunizieren.

Weiterhin nutzt der LayerContainerController das modifzierte JSPF mit Namen JSPF-Remote. Über dieses lädt er die ihm vom SimCore als URL's übergegebenen Layer.

Das Package LayerAPI enthält die beiden Interfaces ILayer sowie IServices. ILayer erweitert dabei das Interface Plugin aus JSPF-Remote, was die einzige Bedingung seitens JSPF an ein Plugin ist. IServices aggregiert die angebotenen Dienste aus der Services Komponente und ermöglicht damit z.B. den Zugriff auf den Datacube, die SAPD-Uhr, das GIS sowie später ggf. auf ein eigenes Kommunikationssystem.

Die Layer Komponente in der nachfolgenden Grafik verdeutlich wie ein Layer-Plugin ins das System integriert ist. In diesem Fall handelt es sich um WALK als Beispiel, welches durch die Implementierung des Interfaces ILayer zu einem validen Plugin werden würde.

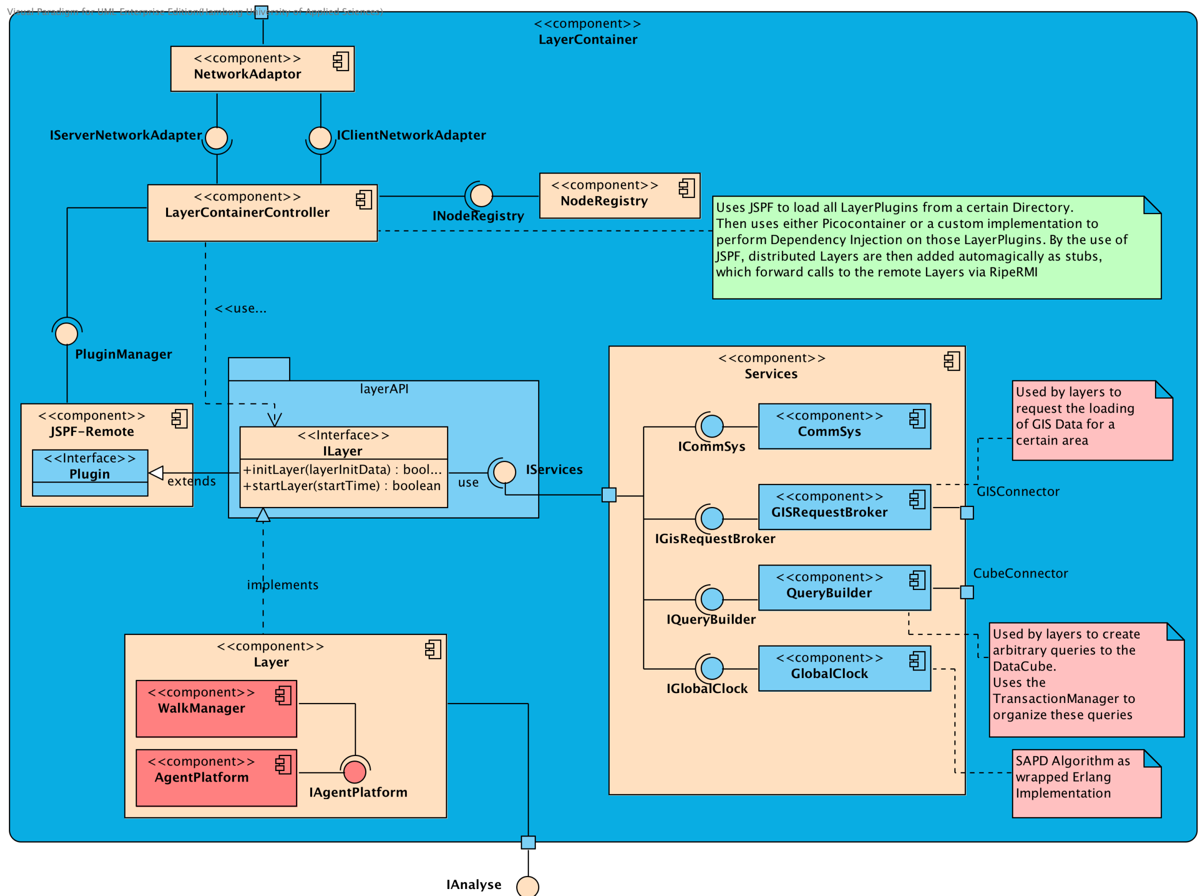

Abbildung 6.4: LayerContainer Komponenten

6.6.1 Layer Verteilung

Eine grundlegende Frage bei der Verteilung ist, ob Layer lediglich innerhalb eines Layer Containers oder auch über mehrere Container verteilt laufen können sollen. Aufgrund der bei einer Verteilung über LayerContainer hinweg entstehenden hohen Komplexität bzgl. Partitionierung und Kommunikation, werde ich zunächst nur die einfachere Variante betrachten.

Über den SimulationController kann die Zuweisung der Layer zu den verfügbaren Containern

vorgenommen werden. Die Layer werden dann jeweils in genau einem LayerContainer geladen und instanziiert. Über das JSPF Framework stellt der Container dann die Layerimplementierung anderen Containern im Netzwerk bereit.

6.7 Schnittstellen für Entwickler

Um einen Layer als Plugin für das System zu entwickeln, muss lediglich das Interface IPlugin aus dem Package de.haw.run.LayerAPI implementiert werden. Sollen mehrere Layer implementiert und auch miteinander verbunden werden, so müssen die Entwickler das Interface ILayer erweitern und ihre eigenen Interfaces zusammen mit den Interfaces der LayerAPI in einer JAR Datei bereitstellen. Über die technische Verbindung der Layer, muss sich der Entwickler weiter keine Gedanken machen. Dies wird durch das JSPF Framework automatisch erledigt. (siehe 6.2, 6.2.1 & JSPF - Plugin Design Practices[1]).

Die Methode initLayer(TInitializationType layerInitData) dient für Schritt 2 der Initialisierungsphase (6.9.1) als kontrollierbarer Konstruktorersatz und wird entsprechend zentral vom SimCore aus aufgerufen. Hier sollen Entwickler ihren Layer in einen Zustand bringen, der das Starten der Simulation erlaubt. Über den Parameter vom Typ TInitializationType kann diese Initialisierung ggf. durch eine Szenariodefinition individuell angepasst werden. In initLayer() darf ebenfalls auf andere Layer zugegriffen werden, zu denen dieser Layer eine Abhängigkeit hat. RUN garantiert, dass diese Layer zu diesem Zeitpunkt bereits initialisiert und verfügbar sind.

Die Methode advanceOneTick() wird im Rahmen der Ausführungssteuerung (6.9.2) verwendet. Was genau ein Tick im Einzelfall aus Sicht der Simulationszeit und der wallclock-Zeit 2.1.2 bedeutet, muss vom Modellierer in der Modelldefinition festgelegt werden. Generell gilt aber, dass ein Tick die kleinstmögliche, diskrete Ausführungseinheit in der Simulation sein soll.

RUN stellt keinerlei Vorgaben an die Interna eines Layers oder gar die Implementierungssprache, solange das Java-Interface ILayer bedient wird. Dem Entwickler steht das Interface IService zur Verfügung, über welches er sich der Zusatzfunktionen des LayerContainers - wie etwa GIS-Zugriff, Datenhaltung etc. - bedienen kann. Die Kommunikation zwischen den Layern einer Simulation braucht der Entwickler nicht zu implementieren. Über das

[1] http://code.google.com/p/jspf/wiki/UsageGuide#Plugin_Design_Practices

```java
package de.haw.run.layerAPI;

import net.xeoh.plugins.base.Plugin;
import java.util.UUID;

/**
 * This interface must be implemented by anyone who
 * wants to create a new Layer runnable in a LayerContainer.
 * The interface IServices may be used inside of that
 * implementation to access the provided services
 * from the LayerContainer.
 */
public interface ILayer extends Plugin {

    /**
     * Initializes the layer with layerInitData.
     * Use this instead of the constructor, as it is
     * guaranteed to be called in the correct load order.
     * @caution MUST NOT USE IGlobalClock (!!!), since
     * this will be in sync only after initialization
     * of all layers.
     * @pre This layer was successfully added to
     * its container.
     * @post This layer is in a state which allows
     * it to start the simulation.
     * @param layerInitData A datatype holding the
     * information of how to initialize a layer.
     * @return A boolean indicating whether or not
     * initialization was successful.
     */
    public boolean initLayer(TLayerInitializationDataType
        layerInitData);

    /**
     * Advances the layer by one tick. A tick is
     * the smallest unit of time possible in the current model.
     * If this layer is a static one, advanceOneTick
```

```
     * may be an empty implementation which simply
     * increases the tickCount.
     * @pre initLayer() was called and returned true.
     * @post getCurrentTick() returns a by 1
     * increased value
     */
    public void advanceOneTick();

    /**
     * The current tick the layer is in.
     * @return Positive long value, if in active
     * simulation or if simulation has ended, -1 otherwise
     */
    public long getCurrentTick();

    /**
     * The unique ID of this layer as a UUID.
     * @return UUID representing the unique ID
     * of this layer.
     */
    public UUID getID();

}
```

6.8 Zeitmanagement & Synchronisation

Die Synchronisierung kausal abhängiger Systemereignisse auf verteilten Knoten kann sehr zu Lasten der (System)Performance gehen, wie ich im Kapitel 5.5 dargelegt habe. Da die Synchronisierung unabdingbar für die korrekte Funktionsweise des MAS ist, habe ich nach einer Verbesserung der rein optimistischen bzw. rein konservativen Synchronisierung gesucht und den Hybrid Ansatz von Pawlaszczyk & Timm (2007) gefunden (siehe 5.5.2, S. 56). Der Hybridansatz verspricht eine Verminderung der notwendigen Rollbacks etwa um den Faktor 3.

Wie im vorherigen Kapitel aufgezeigt, ist es nötig Simulationsereignisse bei der Speicherung zu Linearisieren. Zur Unterstützung dieser Anforderung, ist es sinnvoll eine globale Systemuhr zu benutzen. Globale Systemuhren haben allerdings den Nachteil, dass sie für gewöhnlich

zentral gehalten sind und dadurch tendenziell ein Flaschenhals bei der Berechnung neuer Systemzustände werden.

Aus diesem Grunde habe ich mich nach entsprechender Recherche für einen neuartigen Ansatz mit Namen SAPD (Synchronization Algorithm of Parametric Difference) von Fan *et al.* (2013) entschieden.

Das Verfahren stellt eine Situation her, in der auf jedem Knoten des Simulationssystems eine eigene, virtuelle Uhr läuft. Diese Uhren werden in einem zweistufigen Verfahren synchronisiert. Das Verfahren ist dabei gemäß des Fazits von Fan *et al.* (2013) derart stabil, dass eine Synchronisation nur initial von Nöten ist. Einzig das Hinzufügen eines neuen Knotens oder gravierende Netzwerkveränderungen würden eine erneute Synchronisierung erforderlich machen. Das Verfahren ist auf die Millisekunde präzise. Der Präzisionsbereich liegt zwischen 0.63 und 7.28 Mikrosekunden.

Um unnötigen Verwaltungsaufwand zu vermeiden und eine möglichst schnelle und einfache Netzwerkübertragung zu gewährleisten, wird der SAPD-Algorithmus in Erlang umgesetzt. Architektonisch wird dabei jede LayerContainer Instanz eine SAPD Komponente enthalten, die über das Erlang Modul "Jinterface" in einen Java-Wrapper eingepackt wird. Der Algorithmus läuft für das System transparent in Erlang ab. Die einzige Schnittstelle für den Modellentwickler stellt eine Methode, um sich die aktuelle Uhrzeit zu holen sowie eine weitere Methode, um im Sinne eines Timers, einen Callback zu registrieren zur Verfügung. Wie nachfolgend dargestellt, verfügt jeder LayerContainer und jeder SimCore über eine eigene Instanz dieser Uhr.

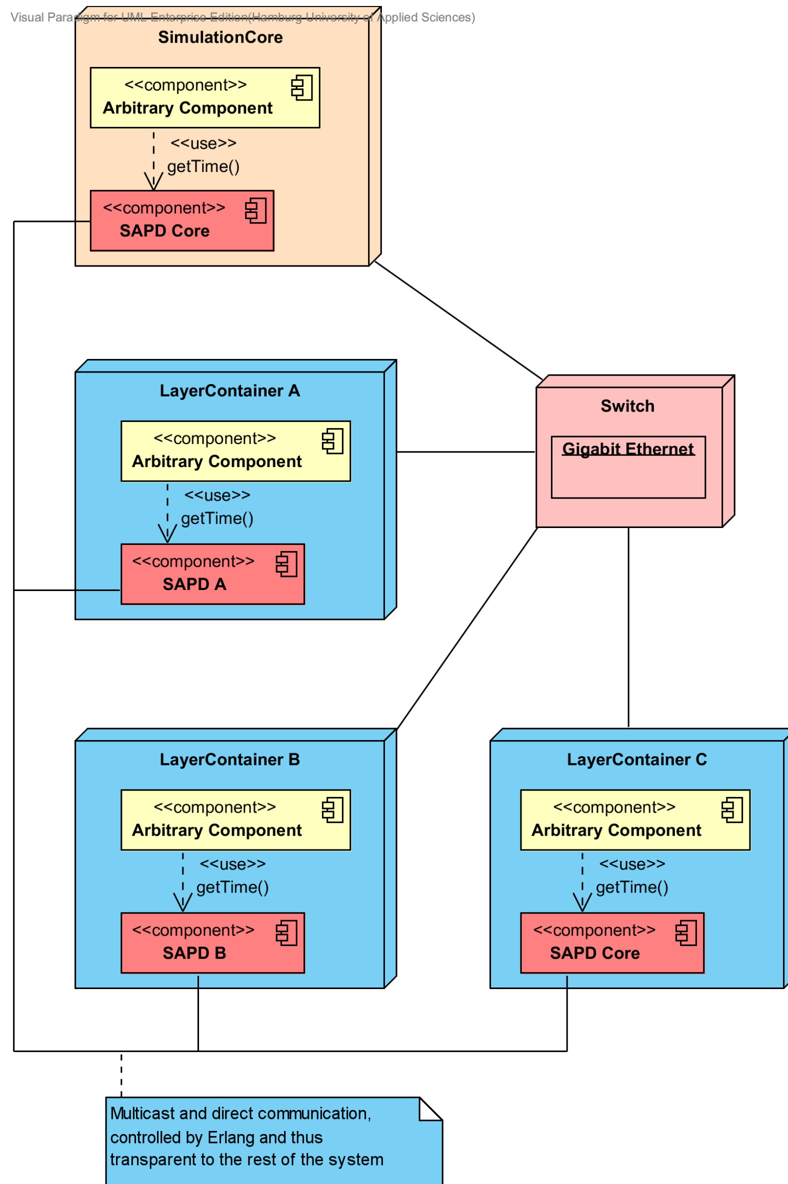

Abbildung 6.5: SAPD Komponentenverteilung

6.9 Startvorgang des Systems

6.9.1 Initialisierung

Voraussetzung für die Initialisierung ist die erfolgreiche Durchführung des DiscoveryProcess insofern, als dass wenigstens eine Instanz von jedem Systemteil präsent ist. Die Initialisierung

des Gesamtsystems beginnt nun mit der Abfrage alle verfügbaren Simulationsmodelle durch den SimulationController. Der Benutzer des SimulationControllers legt fest, welche Layer auf welchem LayerContainer ausgeführt werden sollen und schickt diese Verteilungszuordnung an den SimCore. Dieser errechnet die korrekte Initialisierungsreihenfolge der einzelnen Layer unter Berücksichtigung ihrer Abhängigkeiten untereinander und fügt die Layer anschließend in dieser Reihenfolge den jeweiligen LayerContainern hinzu.

Der nächste Schritt in der Initialisierung wäre das Herstellen eines Startzustandes in den Layern. Da hierzu möglicherweise bereits auf den Datacube zugegriffen werden könnte, muss dieser zunächst bereitgestellt werden. Wie in Kapitel 5.3.7 begründet, soll als DB-Lösung VoltDB eingesetzt werden. Daher ist es notwendig die Datenbankschemata inklusiver ihrer Partitionierungslogik (siehe 5.3.6) dynamisch aus den hinzugefügten Layern zu erstellen. Diese Generierung erfordert einen eigenen Prozess, der seinerseits direkt die VoltDB Instanz(en) ansteuert. Ein solcher Prozess ist absolut im Rahmen des Möglichen und die VoltDB Doku zeigt stellenweise bereits auf, dass derartige Mechaniken unterstützt werden, jedoch würde die Konzeption und Umsetzung den Rahmen dieser Bachelorarbeit sprengen. Aus diesem Grund wird die automatisierte Initialisierung der Datenbank hier zunächst ignoriert und ich geh davon aus, dass dieser Schritt manuell erledigt wird.

Sind alle Layer auf den LayerContainern geladen, versendet der SimCore an alle LayerContainer entsprechende Initialisierungsnachrichten, die das Erzeugen des Startzustandes in allen Layern triggern. Nach erfolgreicher Erzeugung, wird auf den LayerContainern der SAPD-Uhrenprozess ohne Sponsoring gestartet. Haben alle LayerContainer dem SimCore einen Erfolg zurück gemeldet, startet dieser den SAPD-Uhrenprozess als Sponsor und löst dadurch die Synchronisation aus. Ist auch das erfolgreich abgeschlossen, meldet der SimCore die erfolgreiche Beendigung der Initialisierung an den SimulationController. Der Benutzer kann von dort nun die Ausführung der Simulation starten. Der Prozess ist in Abbildung 6.6 veranschaulicht.

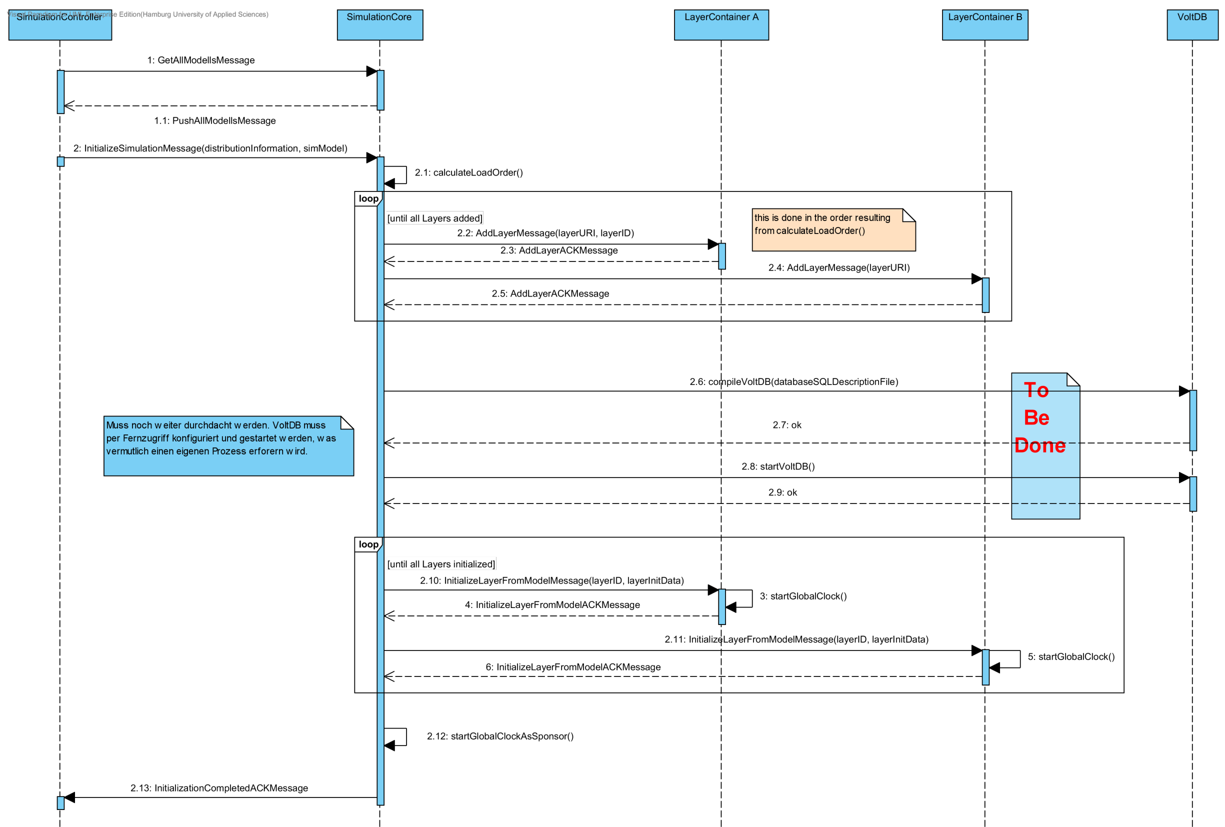

Abbildung 6.6: RUN Initialisierungsprozess

6.9.2 Ausführungsablauf

Um die Performanceprobleme einer zentralisierten Ausführungssteuerung zu umgehen, verwende ich die Möglichkeiten der SAPD-Uhr, um in einem 2-stufigen Verfahren von einer zentralisierten Ausführung (CSE: Centralized Simulation Execution) zu einer de-zentralisierten Ausführung (DCSE) zu gelangen. Dabei wird grundsätzlich von einer Ausführung in diskreten Schritten (Tick) ausgegangen.

Nach der erfolgreichen Initialisierung des Systems, initiiert der SimulationController den Simulationsstart. Das veranlasst den SimCore auf allen LayerContainers die Ausführung von genau einem Simulationsschritt durchzuführen. Was genau ein Simulationsschritt ist, bleibt dabei im Grunde dem jeweiligen Layer überlassen. Im Falle von statischen Layern, bewirkt dieser Aufruf theoretisch nichts. Jeder LayerContainer liefert als Ergebnis die Dauer dieser Ausführung zurück. Der SimCore berechnet aus diesen Werten, die maximale Dauer der Aus-

führung eines Ticks. Das Verfahren sieht nun vor, diesen Schritt so oft zu wiederholen, bis die maximale Ausführungsdauer eines Ticks innerhalb einstellbarer Grenzen, für eine ebenfalls einstellbare Anzahl von Wiederholungen stabil bleibt. Ist dieser Zustand erreicht, delegiert der SimCore die Ausführung unter Angabe einer sich auf die SAPD-Uhr beziehenden Startzeit sowie der Ausführungsdauer eines Ticks an die LayerContainer. Da die SAPD-Uhr auf allen Knoten gleichmäßig läuft, ist so eine de-zentrale, schrittweise Ausführung ohne weitere Kommunikation erreicht.

Die LayerContainer überwachen die Ausführungsdauer ihrerseits. Sollten sie feststellen, dass die vom SimCore ermittelte maximale Dauer überschritten wird, halten sie die Ausführung an und fordern alle anderen Knoten auf dasselbe zu tun. Anschließend benachrichtigt der betroffene LayerContainer, den SimCore, dass er die Ausführung wieder übernehmen soll und teilt ihm den letzten, korrekt ausgeführten Tick mit. Der SimCore organisiert nun einen Rollback des System auf diesen Tick und beginnt abermals mit der CSE. Das Verfahren ist in Abbildung 6.7 ohne den Rollback dargestellt.
Die Organisation eines Rollbacks wie oben beschrieben, ist nicht trivial und muss ebenfalls gesondert behandelt werden, da auch sie den Rahmen dieser Arbeit gesprengt hätte.

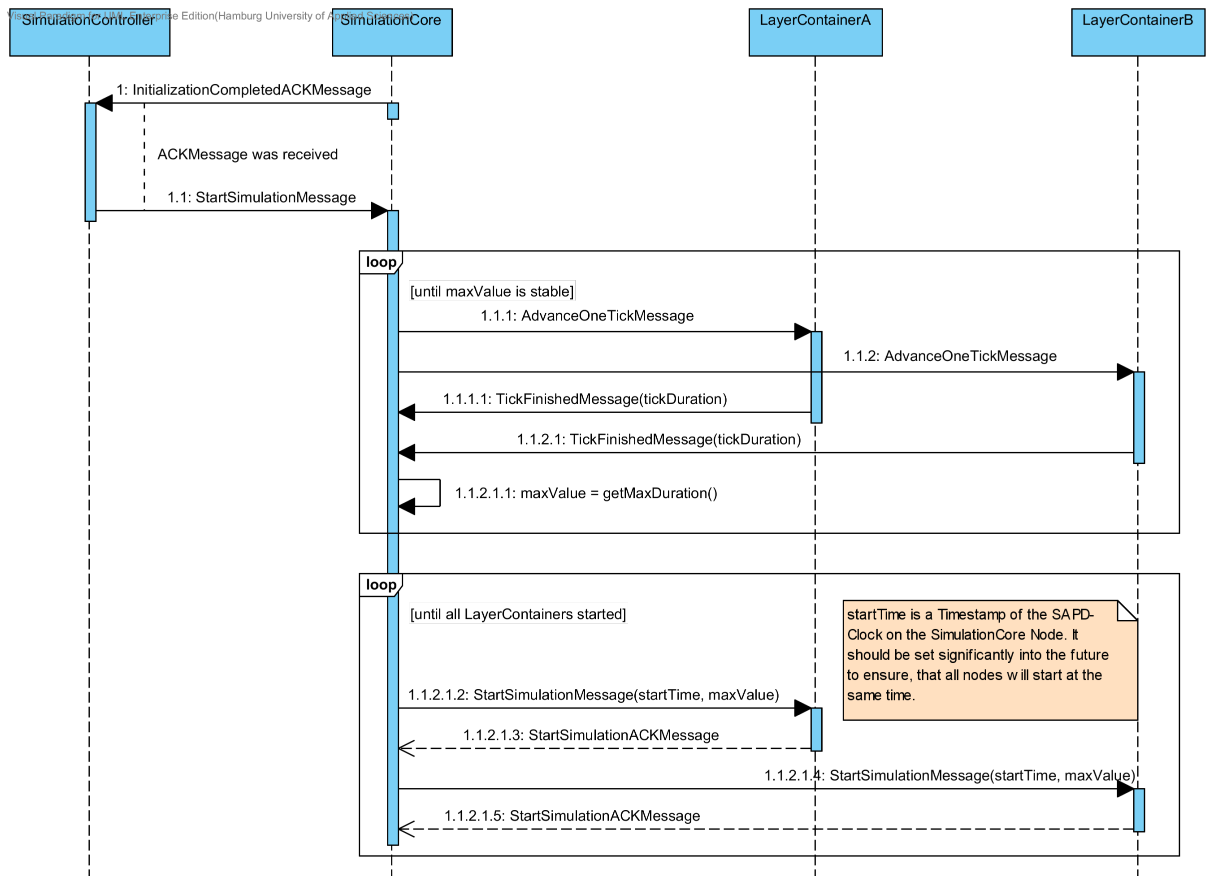

Abbildung 6.7: RUN Ausführungsbeginn CSE / DCSE

6.10 Implementierung des SAPD-Algorithmus

Der interne Aufbau der Erlang Implementierung gliedert sich in sechs verschiedene Prozesse und eine Utility-Komponente. Die Prozesse eines Erlang Programmes können in UML-Notation als Komponenten interpretiert werden, woraus sich der Aufbau wie in Abbildung 6.8 ergibt.

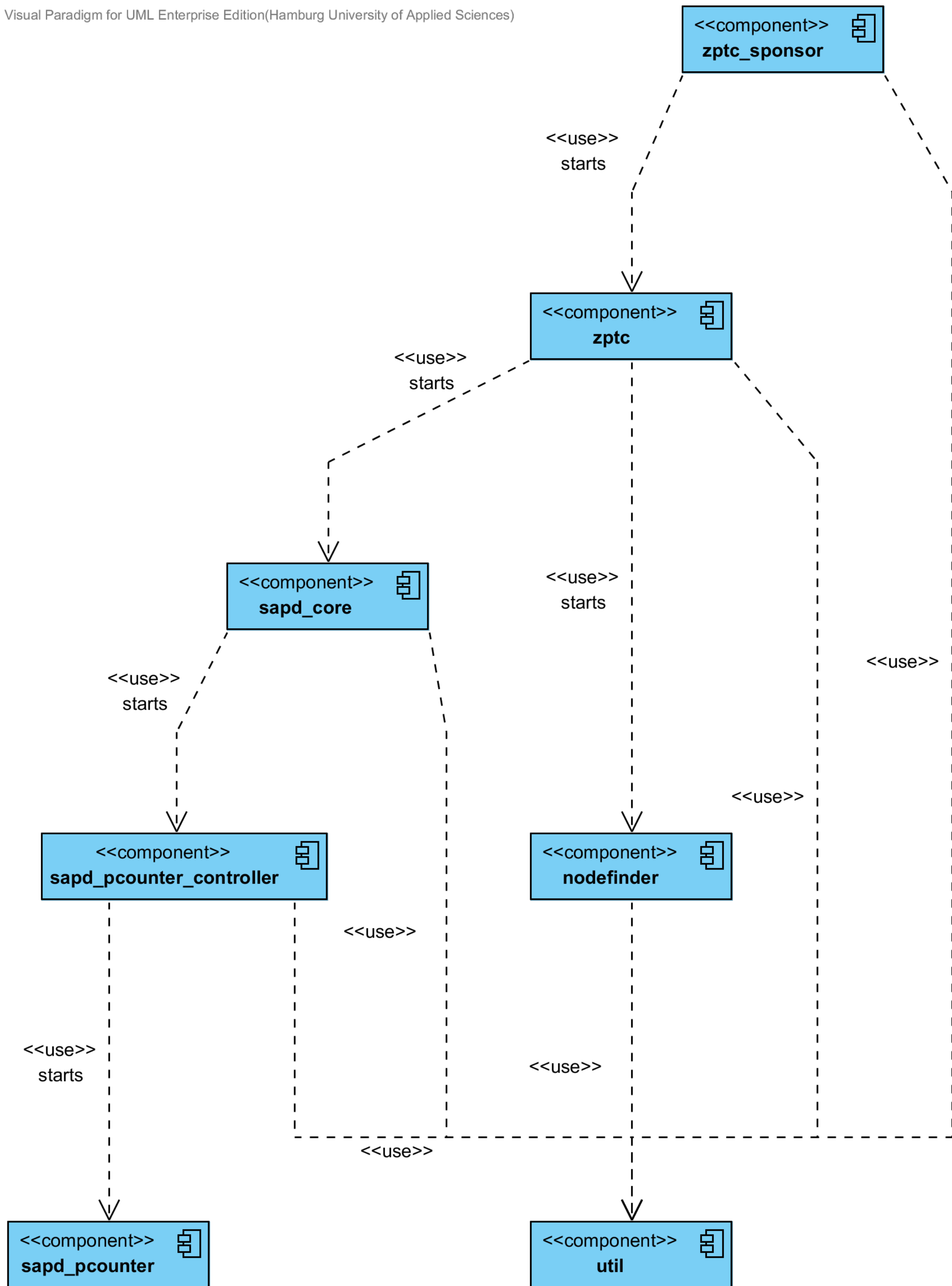

Abbildung 6.8: SAPD Komponentenverteilung

util Die Util-Komponente enthält lediglich Hilfsfunktionen, die in fast allen anderen Komponenten benötigt werden.

nodefinder Bei der Nodefinder-Komponente handelt es sich um eine Erlang-App, die dem automatisierten Auffinden anderer Erlangknoten im Netzwerk dient. Die Implementierung stammt von den Dukes of Erl: http://code.google.com/p/nodefinder/

sapd_pcounter Der SAPD_PCounter ist der tatsächliche Zählprozess welcher in regelmäßigen Zeitintervallen den Zeitzähler erhöht.

sapd_pcounter_controller Der Controller verwaltet Zeitabfragen externer Programme sowie die dem Algorithmus inhärenten Frequenzänderungen und stellt den PCounter entsprechend ein.

sapd_core Der Kern des Algorithmus. Hier ist der eigentliche Mechanismus zur Synchronisierung implementiert.

zptc Die Clientvariante der Zero Point Time Calibration stellt die Basis für eine von einem als Sponsor ausgewählten Knoten angestoßene Nullzeitkalibration dar. Dabei wird versucht den grundsätzlichen Latenzfaktor zwischen den Knoten zu bestimmen und auszugleichen, so dass alle Knoten etwa zur selben Zeit die Initialisierung starten.

zptc_sponsor Ergänzt zptc um die Sponsoringfähigkeit. Ein Aufruf von zptc_sponsor:start(). löst den Beginn des gesamten verteilten Uhrensynchronisationsverfahrens aus.

7 Diskussion & Ausblick

7.1 Bewertung von RUN

Die vorgeschlagene Architektur wird wie die eingangs untersuchten MAS anhand der gleichen Kriterien abgeprüft:

7.1.1 Agenten & Umwelt

Durch den Layeransatz werden keinerlei Bedingungen an Agenten oder Umwelt gestellt. Der Modellentwickler ist völlig frei in seiner Entscheidung wie er beides umsetzt, solange es sich in einer raumzeitlich spatial expliziten Simulation verwenden lässt. Die Pluginstruktur erlaubt es Layer konfigurierbar zu machen und sie wiederzuverwenden, so dass man sich nach und nach einen Baukasten zusammenstellen kann, aus dem spätere Entwickler sich wiederum bedienen können. Über die IServices Schnittstelle kann den Layern nach und nach eine ebenfalls wiederverwendbare

7.1.2 Visualisierung

Eine Visualisierung ist bislang weder umgesetzt noch komplett konzipiert. Das Thema hätte den Rahmen dieser Arbeit deutlich überschritten. Eine erste Idee geht davon aus ähnlich wie beim Verfahren von Vigueras (Vigueras *et al.*, 2013), virtuelle Kameras zu definieren, die nur bestimmte Teile der Umwelt abbilden. Dabei wäre es auch möglich lediglich bestimmte Layer abzufragen und so die Visualisierung an die jeweiligen Bedürfnisse präzise anzupassen.

7.1.3 Performanz & Skalierbarkeit

Die Ausführung der Layer erfolgt jeweils in einem eigenen Thread, so dass jeder LayerContainer in sich auf Mehrkernprozessoren mit ausreichendem RAM skaliert. Erste Tests mit drei einfachen Layern angelehnt an das Abdoulaye Modell legen dies Nahe. Dabei wurden 1000 Bauern, 100.000 Bäume sowie ein einfacher, zufälliger Wetterlayer implementiert. Bei entsprechendem Wetter fällen die Bauern Bäume. Alle drei Systemteile wurden auf einem 6

Kern AMD Phenom X2 mit 2,8 Ghz und 8GB Ram unter Windows 8 64bit initialisiert und ausgeführt. Die Systemressourcen wurden dabei komplett genutzt, also alle 6 Kerne ausgelastet. Weiterführende Lasttests stehen noch aus und erfordern die Implementierung eines realen Simulationsmodels oder aber eines speziellen Benchmarkmodels.

Durch die einfache Möglichkeit Layer auf verschiedene LayerContainer zu verteilen, kann das System problemlos horizontal skalieren. Dabei wird derzeit vorausgesetzt, dass der jeweilige LayerContainer sein/e Layer entsprechend schnell berechnen kann. Das Gesamtsystem kann nur so schnell sein, wie der langsamste Layercontainer. Durch die DCSE wird zumindest der Kommunikationsoverhead einer reinen zentralen Ausführung gespart. Eine dynamische Umverteilung der Layer ist derzeit nicht implementiert, dürfte unter Berücksichtigung der Ergebnisse von Thiel (2013) aber absolut möglich sein. Eine weitere, leider keineswegs triviale Verbesserung wäre die Verteilung eines Layers auf mehrere Layercontainer. Erste Ideen und vergleichbare Arbeiten geben aber Anlass zur Hoffnung, dass dies in einer zukünftigen Version des Systems möglich sein wird.

Die Speicherung von Simulationsdaten konnte ebenfalls noch nicht in einem realen Simulationsszenario erprobt werden. Allerdings lassen die Ergebnisse aus 5.3.7 darauf schließen, dass sich ein Datenbankadapter derart effizient implementieren lässt, dass er das Gesamtsystem nicht besonders aufhält. Es wäre möglicherweise anzuraten, einen verzögerten Schreibmechanismus mit einer Queue zu implementieren, um das Laufzeitverhalten insgesamt weiter zu entlasten.

7.1.4 GIS Unterstützung

Baldowski (2012) hat in seiner Master Projektarbeit ein GIS Nachbarsystem zu RUN geschaffen. Erste Tests mit dem RUN Prototypen und dem System von Baldowski (2012) laufen gerade erst an, so dass über dessen Leistungsfähigkeit noch keine Aussage getroffen werden kann. Konzeptionell macht es allerdings einen guten Eindruck. So sollen die Geodaten von einem zentralen Geoserver aus auf Anfrage durch einen Layer (über eine Serviceschnittstelle) gebündelt und in den Datacube übertragen werden. Von dort aus kann sich der Layer dann der Daten bedienen. In Zukunft ist hier auch denkbar diese Daten mit Daten aus anderen Quellen im Datacube oder einer gesonderten Servicekomponente zu aggregieren und daraus komplette Terrains zu generieren.

Zusätzlich hat Baldowski (2012) ein Webtool implementiert, welches bei der Ansicht und Auswahl der GIS-Daten behilflich sein wird. Auch hier steht ein Verbundtest noch aus.

7.1.5 Erweiterbarkeit

Bei den Layern stellt sich die Frage nach einer Erweiterbarkeit gar nicht, da hier keinerlei Einschränkungen vorliegen. Da die Anbindung externer Systeme über die Serviceschnittstelle stattfinden soll, ist es naheliegend, diese ebenfalls als Pluginstruktur zu konzipieren um die gleichen Vorteile, wie schon bei den Layern zu erhalten. Funktionen, die in mehr als einem Layer Relevanz haben, diverse Datenbankanbindungen, Support für Physikberechnungen durch Nutzung einer Physikengine als Service etc., könnten als Serviceplugin entwickelt werden und würden somit direkt den Umfang des Simulationsframeworks erweitern. Für den SimulationController und eine Visualisierungskomponente sind sehr ähnliche Konzepte denkbar, die die von den Layern genutzten Erweiterungen entsprechend auch in ihren Visualisieruns- bzw. Steuerungspendants zugänglich machen.

7.2 Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung einer Architektur, die insbesondere aber nicht ausschließlich auf die Bedürfnisse der ökologischen Systemmodellierung eingeht. Bei der Betrachtung komplexer sozio-ökologischer Systeme, wie zum Beispiel dem Abdoulaye Naturpark, ist es intuitiv in verschiedenen (Abstraktions-)Ebenen zu denken. Die Komplexität und damit die benötigte Rechenleistung dieser Modelle ist tendenziell sehr groß, was sich am Beispiel Abdoulaye schon in der benötigten Anzahl der simulierten Bäume (4 Millionen) zeigt. Diesen Umständen und dem Ziel der Arbeit ist mit dem Konzept der layerbasierten Simulation sowie den Ideen zur Skalierbarkeit und Performanzsteigerung Rechnung getragen worden. Die Kombination aus Datacube Datenhaltung und dem erweiterbaren Layerkonzept bedienen außerdem den Information Integration Prozess von Thiel-Clemen (2013). Das in dieser Arbeit implementierte System zeigt allerdings lediglich die Machbarkeit dieser Konzepte auf und stellt bestenfalls eine prototypische Implementierung dar, die es in der Zukunft mit weiteren Konzepten zu verbessern und weiterzuentwickeln gilt.

7.3 Ausblick

Während der Arbeit an RUN sind mir diverse notwendige oder auch einfach nur interessante Aspekte und/oder Erweiterungsmöglichkeiten untergekommen, die ich hier noch einmal abschließend kurz beschreiben möchte:

7.3.1 Linearisierung & Zeit

Ein sehr fundamentales Thema ist das der Zeitverwaltung. Hier ist in Zukunft noch einiges an Arbeit zu investieren, damit der Umgang mit den verschiedenen Zeitarten (vgl. 2.1.2) im System möglichst einfach und effektiv wird. Besonders spannend wird dies bei der Linearisierung, welche bei der Speicherung der Layerdaten im Datacube nötig sein wird. Da die Daten mit SAPD-Zeitstempel versehen sein werden, müssen diese in eine Reihenfolge gebracht und dementsprechend im Cube abgelegt werden. Aber auch der SAPD-Algorithmus muss noch erheblich verbessert werden. Der Pseudocode im Paper von Fan *et al.* (2013) ist leider sehr undurchsichtig und teilweise offenbar sogar fehlerhaft, so dass meine Implementierung eine Interpretation der Grundidee des Algorithmus ist. Zwar funktioniert diese unter Laborbedingungen, ein wirklicher Test auf Robustheit etc. dürfte aber noch einige Schwachstellen offenbaren.

7.3.2 Globales 3D-Koordinatensystem für Agenten & Umwelt

Es wird notwendig sein ein gemeinsames 3D Bezugssystem bzgl. Koordinaten in der Simulationswelt zu haben, damit in der Interlayerkommunikation eine Eindeutigkeit herrscht, wenn es um Lokationsbestimmungen etc. geht. Aus diesem Grund schlage ich vor jedem Layer automatisch ein in Größe und Skalierung je Model einmalig global konfigurierbares 3D-Koordinatensystem zu unterlegen. Benötigt man lediglich zwei Dimensionen, kann man die Z-Achse einfach ignorieren.

7.3.3 Datacube Adapter

Für die Ansteuerung des Datacubes und technisch gesehen der VoltDB, benötigt RUN einen vernünftigen Adapter, der während der Initialisierungsphase die Erzeugung und Initialisierung der korrekten Tabellen mitsamt ihrer Partitionierungen sowie der Stored Java Procedures übernimmt. Ausserdem sollte er den Layer-Entwicklern den Zugriff auf Daten aus dem Cube erleichtern.

7.3.4 Snapshot Adapter

Um regelmäßige Snapshots des Simulationszustandes anzufertigen, empfiehlt es sich eine NoSQL Datenbank wie Couchbase in Verbindung mit einem JSON Generatpr wie z.B. Googles GSON zu verwenden. In Kapiel 5.3.6 habe ich bereits gezeigt, dass Couchbase in Punkto Geschwindigkeit insbesondere beim Schreiben alle anderen Datenbanken weit hinter sich lässt. Die einfache und problemfreie Abbildung von Objekte auf JSON Dokumente und andersherum tut ihr Übriges um Couchbase in Verbindung mit GSON als Lösung für eine Snapshot-Lösung zu empfehlen. Diese Snapshots könnten auch in einer Rollback-Mechanik für die Rückführung auf DCSE (siehe 6.9.2) genutzt werden.

7.3.5 Beschreibungssprache für Model- & Szenariodefinition

Zur Erzeugung von Modellen und Szenarien bietet es sich an - zumindest als technischen Unterbau - eine Domain Specific Language zu erstellen, mit deren Hilfe man entweder einen grafischen Editor auf einer höheren Abstraktionsebene aufsetzen oder aber direkt neue Modelle und Szenarien beschreiben kann.

Literaturverzeichnis

Amouroux, Edouard, Thanh-quang, C H U, Boucher, Alain, & Drogoul, Alexis. 2007. GAMA : an environment for implementing and running spatially explicit multi-agent simulations.

Baldowski, Mariusz. 2012. Geoservices für WALK Usecases.

Fan, Linjun, Ling, Yunxiang, Wang, Tao, Zhu, Xiaomin, & Tang, Xiaoyong. 2013. Novel clock synchronization algorithm of parametric difference for parallel and distributed simulations. *Computer Networks*, **57**(6), 1474–1487.

Fujimoto, RM. 1999. Parallel and distributed simulation. *Simulation Conference Proceedings, 1999 . . .*, 147–157.

Henderson, Tristan, & Bhatti, Saleem. 2003. Networked games — a QoS-sensitive application for QoS-insensitive users. 141–147.

Jarschel, Michael, Schlosser, Daniel, Scheuring, Sven, & Hoß feld, Tobias. 2013. Gaming in the clouds: QoE and the users' perspective. *Mathematical and Computer Modelling*, **57**(11-12), 2883–2894.

Jefferson, D, & Sowizral, H. 1983. Fast Concurrent Simulation Using the Time Warp Mechanism, Part II.

Kallman, Robert, Natkins, Jonathan, Jones, Evan P C, Hugg, John, & Abadi, Daniel J. 2008. H-Store : A High-Performance , Distributed Main Memory Transaction Processing System.

Parunak, HVD, Savit, Robert, & Riolo, RL. 1998. Agent-based modeling vs. equation-based modeling: A case study and users' guide. *Multi-Agent Systems and Agent-Based . . .*, 10–25.

Pawlaszczyk, Dirk, & Timm, IJ. 2007. A hybrid time management approach to agent-based simulation. *KI 2006: Advances in Artificial Intelligence*, **4314**(Ki).

Pereki, H. 2013. Tropical dry forests above-ground biomass measurements in West Africa, Abdoulaye Wild Reserve (Togo).

Sadalge, J. Pramod, & Fowler, Martin. 2013. *NoSQL Distilled.*

Seidl, Rupert, Rammer, Werner, Scheller, Robert M, & Spies, Thomas a. 2012. An individual-based process model to simulate landscape-scale forest ecosystem dynamics. *Ecological Modelling*, **231**(Apr.), 87–100.

Stonebraker, Michael, Abadi, Daniel J, Harizopoulos, Stavros, & Helland, Pat. 2007. The End of an Architectural Era (It ' s Time for a Complete Rewrite).

Thiel, Christian. 2013. *Analyse von Partitionierungen und partieller Synchronisation in stark verteilten multiagentenbasierten Fußgängersimulationen.* Master Thesis, Hamburg University of Applied Sciences.

Thiel-Clemen, Th. 2013. Information Integration in Ecological Informatics and Modelling.

Vigueras, G., Lozano, M., Orduña, J.M., & Grimaldo, F. 2010. A comparative study of partitioning methods for crowd simulations. *Applied Soft Computing*, **10**(1), 225–235.

Vigueras, Guillermo, Orduña, Juan M., Lozano, Miguel, & Jégou, Yvon. 2013. A scalable multi-agent system architecture for interactive applications. *Science of Computer Programming*, **78**(6), 715–724.

Yang, Hung-chih, Dasdan, Ali, Hsiao, Ruey-lung, & Parker, D Stott. 2007. Map-Reduce-Merge : Simplified Relational Data Processing on Large Clusters. 1029–1040.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 20. September 2013 Christian Hüning