Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Christian Thiel

Analyse von Partitionierungen und partieller Synchronisation in stark verteilten multiagentenbasierten Fußgängersimulationen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Thiel

Analyse von Partitionierungen und partieller Synchronisation in stark verteilten multiagentenbasierten Fußgängersimulationen

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

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

Abgegeben am 13. Februar 2013

Christian Thiel

Titel der Arbeit

Analyse von Partitionierungen und partieller Synchronisation in stark verteilten multiagentenbasierten Fußgängersimulationen

Stichworte

Multiagentensystem, Fußgängersimulation, Verteiltes System, Synchronisation, Partitionierung, Lastverteilung, verteilte virtuelle Umgebungen

Kurzzusammenfassung

Ziel dieser Arbeit ist die Entwicklung einer stark skalierbaren Simulationsplattform für Fußgängersimulationen.
Kernaspekte der Arbeit sind die Entwicklung eines partiellen Synchronisationsmechanismus in einer stark verteilten virtuellen Simulationsumgebung und der Vergleich dieser dabei entwickelten Lösung mit einer vollständigen Synchronisation.
Im Zuge der Entwicklung der partiellen Synchronisation werden mehrere Partitionierungsalgorithmen sowohl theoretisch als auch im praktischen Einsatz miteinander verglichen.

Christian Thiel

Title of the paper

Analysis of techniques for partitioning and partial synchronization in heavily distributed multiagent based crowd simulations

Keywords

Multi-Agent-System, Crowd Simulation, Distributed System, Synchronization, Partitioning, Load-Balancing, Distributed Virtual Environments

Abstract

This work deals with the development of a highly scalable simulation platform for crowd simulation.
Core aspects of the work are the development of a partial synchronization mechanism in a highly distributed virtual simulation environment and the comparison of this method with a full synchronisation mechanism.
During the development of the partial synchronization several partitioning algorithms will be compared both theoretically and in practical.

Inhaltsverzeichnis

1. Einführung **1**
1.1. Ziele der Arbeit . . . 1
1.1.1. Synchronisation . . . 1
1.1.2. Partitionierung . . . 2
1.2. Nicht Teil der Arbeit . . . 2
1.3. Aufbau der Arbeit . . . 3

2. Grundlagen **4**
2.1. Fußgängersimulationen . . . 4
2.1.1. Klassifikation nach Anwendungsgebiet . . . 4
2.1.2. Grundlegende Ansätze für Fußgängersimulationen . . . 5
2.1.3. Verteilte Multiagentensysteme . . . 7
2.2. Verteilte Systeme . . . 9
2.2.1. CAP-Theorem . . . 9
2.2.2. Konsistenz in verteilten Systemen . . . 10
2.2.3. Synchronisierung und Zeit am Beispiel von HLA . . . 13
2.2.4. Verteilte Datenspeicher . . . 15
2.2.5. Agenten-Sichtbereiche . . . 21
2.3. Partitionierung . . . 23
2.3.1. Auswirkung verschiedener Partitionierungen . . . 23
2.3.2. Statische und dynamische Partitionierung . . . 24
2.3.3. Vergleichbare Arbeiten . . . 25

3. Konzept **34**
3.1. Architektur von WalkSim . . . 34
3.1.1. WalkManager . . . 34
3.1.2. Geoinformationssystem . . . 36
3.1.3. AgentPlatform . . . 36
3.1.4. WalkCS . . . 36
3.2. Distributed Environment . . . 37
3.2.1. Sektoren . . . 38
3.2.2. Verteilung . . . 39

3.3. Simulationsausführung . . . 41
3.3.1. Zeit . . . 42
3.4. Die acht Irrtümer der verteilten Datenverarbeitung . . . 42

4. Implementierung **45**
4.1. Distributed Virtual Environment . . . 45
4.1.1. Datenmodell . . . 46
4.1.2. Synchronisation . . . 47
4.1.3. Spatiale Hashdatenbank (In-Memory) . . . 49
4.1.4. Persistenz . . . 52
4.2. Partitionierung . . . 53
4.2.1. Partitionierungsdaten . . . 54
4.2.2. Anpassung der Referenzimplementierungen für WalkSim . . . 56
4.3. Kommunikationssystem . . . 58
4.3.1. Implementierung . . . 60
4.3.2. Nachteile . . . 62
4.4. Agenten und KI . . . 62
4.4.1. Verwendete Agenten-KI . . . 64
4.5. Szenarios . . . 65
4.6. Simulationsaufführung . . . 66

5. Benchmarks und Experimente **68**
5.1. Testszenarien . . . 68
5.1.1. Messgrößen . . . 69
5.1.2. Synchronisation . . . 70
5.1.3. Skalierbarkeit . . . 71
5.1.4. Partitionierung . . . 71
5.2. Nicht Teil der Tests . . . 76
5.2.1. Spatiale Hashdatenbank . . . 76
5.2.2. Simulationsschrittweite . . . 77

6. Ergebnisse **78**
6.1. Vergleich der Synchronisation . . . 78
6.1.1. Synchronisationsdauer . . . 79
6.1.2. Datenmengen . . . 80
6.2. Skalierbarkeit . . . 81
6.3. Partitionierung . . . 82
6.3.1. Test 1 - Einzelgruppen . . . 82
6.3.2. Test 2 - Chaotische Menge . . . 84
6.3.3. Test 3 - Evakuierung mit einem Notausgang . . . 85
6.3.4. Test 4 - Homogene Menge . . . 87

6.3.5. Test 5 - Bahnhof 88
6.3.6. Zusammenfassung 89

7. Diskussion **90**
7.1. Synchronisationsperformance 90
7.2. Skalierbarkeit 91
7.2.1. Skalierung bei konstanter Agentenmenge 91
7.2.2. Skalierung bei linear steigender Simulationsgröße 92
7.2.3. Zusammenfassung 93
7.3. Partitionierung 93
7.3.1. KMeans 94
7.3.2. QHull 95
7.3.3. Peschlow 96
7.3.4. Bisection 97
7.3.5. Peschlow + Bisection 99
7.3.6. Zusammenfassung 100

8. Ausblick **102**

A. Weitere Bestandteile der WalkSim-Plattform **103**
A.1. Visualisierung 103
A.1.1. Benutzeroberfläche 104
A.1.2. Architektur 105
A.2. Runtime-Plattform 106
A.3. Kollisionserkennung 107

B. DVD-Anlage **108**

Literaturverzeichnis **109**

Abbildungsverzeichnis **113**

1. Einführung

Im Rahmen des Forschungsschwerpunktes *Bevölkerungsschutz* der HAW Hamburg werden innerhalb des WALK-Projektes [44] Methodiken untersucht und entwickelt, mit deren Hilfe Fußgängersimulationen durchgeführt werden können. Für diese Aufgabe soll die verteilte Multi-Agenten-Plattform *WalkSim* entstehen.

In dieser Arbeit sollen für diese verteilte Simulationsplattform Möglichkeiten der effizienten Synchronisation gefunden werden. Bisherige Arbeiten verwendeten meist eine vollständige Replikation sämtlicher Daten der Simulation auf alle Hosts des Systems. Für WalkSim sollen Wege untersucht werden, wie anstatt dieser vollständigen Synchronisation auch eine partielle Synchronisation benutzt werden kann, bei der jeder Host nur die für ihn relevanten Daten synchronisiert. Dabei ist zu untersuchen, wie sich der höhere Verwaltungsaufwand einer solchen partiellen Synchronisation gegenüber der Kommunikationsersparnis verhält.

Die Gesamtleistung eines verteilten Systems hängt nicht nur von der Synchronisation der Teilnehmer ab, sondern auch davon, wie effizient die einzelnen Arbeitspakete auf die verfügbaren physikalischen Hosts verteilt werden können. Im zweiten Themenbereich dieser Arbeit sollen deshalb bereits vorhandene Algorithmen zur Partitionierung von verteilten Fußgängersimulationen im Bezug auf deren Verwendung in WalkSim miteinander verglichen und gegebenenfalls für WalkSim optimiert werden.

1.1. Ziele der Arbeit

1.1.1. Synchronisation

Im Rahmen dieser Arbeit sollen zwei Synchronisationsparadigmen miteinander verglichen werden:

- Die **vollständige Synchronisation** gleicht zu jeder Zeit den kompletten Datenbestand der Simulation zwischen allen Servern ab, sodass jeder Server ein komplettes lokales Abbild der Simulation besitzt

- Bei der **partiellen Synchronisation** gleichen die Server nur die Teile der Simulationsdaten miteinander ab, die auf dem jeweiligen Server auch tatsächlich gebraucht werden. So hat jeder Server nur einen Teil der Simulationsdaten lokal verfügbar und die Kommunikationskosten werden drastisch reduziert. Im Gegensatz zur pauschalen vollständigen Synchronisation ist jedoch ein deutlich höherer Verwaltungsaufwand nötig.

In dieser Arbeit soll eine Methodik für die partielle Synchronisierung entwickelt und implementiert werden und diese anschließend in einer Performanceanalyse im Hinblick auf Gesamtleistung und Skalierbarkeit mit der vollständigen Synchronisation verglichen werden.

1.1.2. Partitionierung

Eine partielle Synchronisation setzt voraus, dass die Teile der Simulation, welche auf die gleichen oder ähnliche Datenbereiche zugreifen müssen, auch auf dem selben Server sind, damit der Datenbestand gering gehalten wird.
Dafür ist ein guter Partitonierungsalgorithmus erforderlich, welcher die einzelnen Agenten oder andere Simulationsbestandteile möglichst optimal auf die vorhandenen Server aufteilt. In dieser Arbeit sollen verschiedene, bereits vorhandene Partitionierungsalgorithmen miteinander verglichen werden. Dafür werden mehrere Testfälle erstellt, welche die Algorithmen vor Extremsituationen stellen. Es soll untersucht werden, wie sich die einzelnen Algorithmen in diesen Extremfällen verhalten und ob es möglich ist, einen universalen Partitionierungsalgorithmus für WalkSim zu entwickeln, welcher die partielle Synchronisation optimal unterstützt.

1.2. Nicht Teil der Arbeit

Eine Fußgängersimulation besteht neben der eigentlichen Multiagentenplattform aus vielen anderen Komponenten:

- Agenten-KI für Planung und Wegfindung
- Fachliches Datenmodell für Agenten, Hindernisse und Gebäudepläne
- Szenariodefinition
- Physik und Kollisionserkennung
- Visualisierung und Validierung von Simulationsinhalten

Diese Bereiche werden im Weiteren nicht ausführlich behandelt. Soweit nötig, wird eine prototypische Implementierung angefertigt, welche den für diese Arbeit erforderlichen Ansprüchen gerecht wird. Es wird jedoch explizit kein Wert auf ein realistisches und authentisches Verhalten der Agenten gelegt, da dies für die Kernfragen dieser Arbeit nicht weiter von Belang ist.

1.3. Aufbau der Arbeit

Die Arbeit gliedert sich in folgende Kapitel:
Kapitel 2 behandelt die Grundlagen der Arbeit. Es werden bisher entwickelte Fußgängersimulationen vorgestellt, Grundlagen der verteilten Datenspeicherung vermittelt und entsprechende vorhandene Arbeiten und Methodiken vorgestellt.
Kapitel 3 beschäftigt sich mit dem für WalkSim erstellten Konzept und der grundlegenden Funktionsweise des Programms und der Verteilung.
Kapitel 4 widmet sich Details zur Implementierung von Kernkomponenten und der Lösung konkreter Probleme.
Im Folgenden beschäftigt sich Kapitel 5 mit Benchmarks und Testfällen, mit denen die implementierten Verfahren experimentell analysiert werden.
In Kapitel 6 werden die Ergebnisse dieser Tests in Tabellen und Grafiken veranschaulicht und signifikante Merkmale hervorgehoben.
In Kapitel 7 werden die zuvor erhaltenen Daten aus den Testfällen analysiert.
Abschließend folgen in Kapitel 8 ein Fazit und ein Ausblick für folgende Arbeiten.

2. Grundlagen

In diesem Kapitel werden zunächst essentielle Grundlagen zum Verständnis der Arbeit erläutert und bereits vorhandene Lösungen zu wichtigen Problemen vorgestellt.

2.1. Fußgängersimulationen

Die Simulation von Menschenmassen ist nicht nur für Gebäudearchitekten und Rettungs- oder Ordnungskräfte interessant. Auch im Bereich der Spielentwicklung ist mit der steigenden Rechenleistung von Spielkonsolen und -PCs immer authentischeres Verhalten von computergesteuerten Spielfiguren möglich.
In Zeiten, wo militärische Konflikte immer mehr in besiedelte Regionen verlegt werden oder gar im Häuserkampf enden, beschäftigt sich sogar das Militär mit der Simulation des Verhaltens von Zivilisten im Rahmen von Kampfhandlungen. Dabei sind eine Vielzahl von akademischen aber auch kommerziellen Simulationssystemen entstanden, von denen im Folgenden einige vorgestellt werden.

2.1.1. Klassifikation nach Anwendungsgebiet

Um diese Systeme zu beschreiben und zu bewerten, ist es sinnvoll, vorher eine Klassifikation anhand des Anwendungsfalls der Simulationen zu erstellen. Für Fußgängersimulationen bietet sich dafür eine Unterteilung nach ihrer Größe an:

- In kleinen Simulationen mit bis zu 100 Individuen sollen meist nur einzelne Räume oder kleine Gebäude (Wohnhäuser, Restaurants etc.) simuliert werden. Dabei ist das realistische Verhalten der einzelnen Personen sehr wichtig.
- In mittelgroßen Simulationen mit Hunderten bis mehreren Tausend Personen (z.B. Bahnhöfe, Hochhäuser, Konzerthallen) stehen die individuellen Bewegungen einzelner Personen nicht an vorderster Stelle, viel mehr kommt es auf die Interaktion von ganzen Personengruppen an.

- In großen Simulationen mit mehreren Zehn- oder gar Hunderttausenden von Personen (z.B. Musik-Festivals, Massendemonstrationen, Mekka) spielt die individuelle Bewegung von Personen so gut wie keine Rolle mehr. Hauptsächlich werden dort Flussbewegungen und Dichteverteilungen innerhalb der Menschenmassen untersucht.

Die Individualität einzelner Personen wird bei solchen Simulationen im Allgemeinen immer unwichtiger, je größer die Simulation ist.

2.1.2. Grundlegende Ansätze für Fußgängersimulationen

Für die konkrete Umsetzung einer Fußgängersimulation wurden diverse, teilweise grundlegend verschiedene Algorithmen entwickelt, welche alle ihre jeweiligen Vor- und Nachteile haben (vgl. [49]).

2.1.2.1. Flussbasierte Simulationen

Flussbasierte Fußgängersimulationen (z.B. [19]) basieren auf einer extrem einfachen Modellierung. Dabei werden die Personen oft nicht einmal mehr als einzelne Objekte dargestellt, sondern es gibt lediglich Werte zur relativen Personendichte in einem abgegrenzten Areal. Die Bewegung der Personen bzw. der Personendichte wird mit Hilfe einer mathematischen Übergangsfunktion beschrieben, wie man sie auch in der Regelungstechnik antrifft:

$$s_{t+\triangle t} = F(s_t, \triangle t)$$

Dabei wird eine Art Übergangswahrscheinlichkeit formuliert (z.B. "50% der Personen in Bereich A gehen im nächsten Rechenschritt nach links"). Die Berechnung des nächsten Zustandes erfolgt meist mit Differentialgleichungen oder Potentialfeldern.

Diese flussbasierten Systeme bieten dementsprechend keine Individualität der einzelnen Personen. So ist es z.B. weder möglich, den Weg einer Person exakt über den Verlauf einer Simulation zu verfolgen, noch können den Personen individuelle Eigenschaften zugewiesen werden.
Der Vorteil dieses Ansatzes ist es jedoch, dass er aufgrund der großen Abstraktion extrem performant ist. Es ist ohne weiteres möglich, Hunderttausende Personen zu simulieren.

Flussbasierte Simulationen eignen sich deshalb sehr gut dafür, extrem große Personenmengen zu simulieren, bei denen die Individualität der Personen nicht wichtig ist.

2.1.2.2. Objektbasierte Systeme

Im objekt- oder auch entitätsbasierten Ansatz werden die Personen als gleichartige Objekte modelliert, deren Bewegungen durch globale und lokale Regelsätze modelliert werden. Dadurch können bereits einige der üblichen global auftretenden Effekte wie Staubildung und *Flocking*[1] nachgebildet werden.

Ein sehr bekanntes Anwendungsbeispiel für objektbasierte Systeme kommt von Helbing et al. mit dem *Social Forces Model* [26]. Dort werden die einzelnen Objekte als Partikel mit einer Masse und einer Geschwindigkeit dargestellt. Die Modifikation der Partikel geschieht mit einer Mischung aus physikalischen und soziophysiologischen Kräften, welche meist als Potentialfeld realisiert werden.

Objektbasierte Systeme haben gegenüber den flussbasierten Systemen den Vorteil, dass sie jede Person als individuelles Objekt modellieren können. So kann die Bewegung einer Person über die gesamte Simulation hinweg beobachtet werden und in begrenztem Umfang auch bereits individuelle Regeln für Personengruppen festgelegt werden.
Dies geschieht jedoch zu Lasten der Performance, da solche individuellen Operationen mehr Rechenleistung benötigen.

2.1.2.3. Multiagentensysteme

Multiagentensysteme [48] bieten im Gegensatz zu den bisher vorgestellten Modellierungen eine maximale Individualität der einzelnen Personen. Jede Person wird als eigenständiger Agent modelliert, welcher autonom denken und handeln kann. Dieser hat seine eigene, von den anderen Agenten unabhängige Modellierung und kann so eine beliebig detaillierte künstliche Intelligenz darstellen.
Multiagentensysteme arbeiten meist ereignisorientiert. Dementsprechend agiert jeder Agent in diesem System wie ein endlicher Automat, welcher mit einem eintreffenden Ereignis $e \in \sum$ mittels einer Übergangsfunktion $\delta : Q \times \sum \rightarrow Q$ von seinem aktuellen Zustand $q_0 \in Q$ in einen neuen Zustand $q_1 \in Q$ übergeht. Für die meisten Multiagentensysteme sind sowohl die Menge der möglichen Zustände eines Agenten Q als auch die Menge der möglichen Ereignisse $\sum$ derart groß, dass die viele Agenten-Implementierungen von der reinen Modellierung eines Zustandsautomaten abweichen und alternative Modellierungen verwenden.

Die Eigenständigkeit der Agenten stellt jedoch andere Anforderungen an die Simulationsumwelt der Agenten, als dies bei fluss- und objektbasierten Systemen der Fall ist. Dies liegt vor allem an ihrer internen Arbeitsweise, die meist in drei Schritte zu unterteilen ist:

[1] Herdenverhalten größerer Gruppen

- **Sense**: Der Agent nimmt seine Umwelt wahr und gleicht sie mit seinem internen Speicher ab. Dabei muss der Agent nicht zwingend die gesamte Umwelt wahrnehmen und der interne Speicher kann durchaus veraltetes Wissen enthalten, wenn das Objekt bereits aus dem Sichtbereich verschwunden ist. Jeder Agent greift dabei auf seine individuelle Wissensbasis zurück.
- **Reason**: In diesem Schritt "denkt" der Agent, indem er auf Grundlage seiner aktuellen Wissensbasis Ziele bestimmt, die er erreichen möchte und Handlungspläne erstellt, mit deren Hilfe er diese Ziele erreichen möchte. Dazu gibt es diverse, teils gänzlich verschiedene Konzepte (z.B. STRIPS, GOAP [27], BDI [48]).
- **Act**: Der Agent wandelt den zuvor erdachten Plan in eine Reihe von möglichen ausführbaren Aktionen um, die von der Simulationsumgebung zur Verfügung gestellt werden und führt diese aus.

Multiagentensysteme bieten eine nahezu beliebige Komplexität innerhalb des Agenten, weshalb die Anforderungen an die Hardware generell sehr hoch sind. Deshalb eignen sich Multiagentensysteme oft nur für kleinere Simulationen, in denen die Individualität der Personen eine unabdingbare Anforderung ist oder die Anforderungen an die Intelligenz der darzustellenden Personen sehr hoch sind.

Beispiele für bereits vorhandene Arbeiten sind das Projekt von Shao et al. [38], wo eine Evakuierung der Pennsylvania Station simuliert wurde, oder auch Klügl et al. [28] mit der Simulation einer Zugevakuierung in einem Tunnel.
Im kommerziellen Bereich finden sich die beiden Projekte AI.Implant [35] und MASSIVE [3], welche ebenfalls mit Agentensystemen arbeiten, und im Fall von MASSIVE z.B. auch in der Filmindustrie bei Blockbustern wie *Der Herr der Ringe* eingesetzt wurden.

2.1.3. Verteilte Multiagentensysteme

Sollen mit einem Multiagentensystem größere Simulationen durchgeführt werden, stößt man aufgrund der hohen Performanceansprüche der Agenten sehr schnell an die Kapazitätsgrenzen eines einzelnen Servers, weshalb man nicht umhin kommt, eine solche Simulation auf mehrere Server zu verteilen.
Weil Agenten grundsätzlich autonom handeln, kann die Agentenmenge verhältnismäßig einfach auf viele Hosts aufgeteilt werden. Jeder Agent muss dafür lediglich Zugriff auf den gemeinsamen *World State* der Simulation haben, welcher den eindeutigen globalen Zustand der Simulation repräsentiert. Die Verteilung dieses Simulationszustandes ist dabei die größte Schwierigkeit. Jeder Agent muss auf dem gleichen Datenbestand arbeiten, egal auf welchem Server er ausgeführt wird. Nur so kann garantiert werden, dass die Simulation verteilt das gleiche Ergebnis erzeugt, als wäre sie nicht verteilt ausgeführt worden.

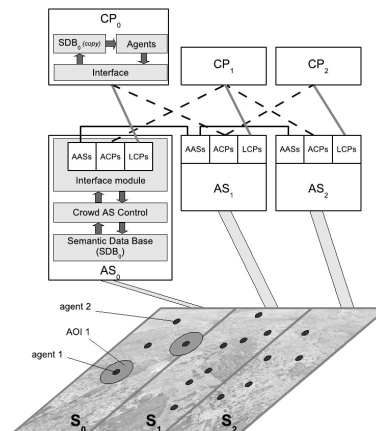


Abbildung 2.1.: Verteilte Architektur beim System der University of Valencia [45]

2.1.3.1. COSMOS - Nanyang Universität

Das Projekt COSMOS [30][33] von der Universität Nanyang entwickelte ein System, bei dem die Agenten mittels der verteilten Simulationsarchitektur HLA [21][20] kommunizieren. Dort gibt es für jeden Server einen sogenannten *Federate* (die zentrale Simulationskomponente von HLA), auf dem die Agenten ausgeführt werden plus einen Federate für die Simulationsumgebung, in dem Daten und Objekte vorgehalten werden, die nicht den Agenten gehören.
Die Synchronisation der Federates übernimmt dabei HLA selbst. Die eigentliche Ausführung und Migration der Agenten wird mit dem JADE Framework [6] innerhalb der Federates bewerkstelligt, die lokale Wissensbasis der Federates und Agenten über das Agentenframework *RePast* realisiert.
Der Simulationszustand wird dabei nicht in einer zentralen Komponente verwaltet, sondern in den einzelnen Federates selbst. Die Federates bzw. Agenten kontrollieren ihre eigenen Aktionen selbst und kommunizieren diese über HLA.

2.1.3.2. University of Valencia

Ein weiteres Projekt entstand an der University of Valencia [31] [45].
Dort wurde die gesamte Simulation in zwei Bereiche aufgeteilt. Zum einen verwalten sogenannte *Action-Server* (AS) die Simulationsumgebung. Von diesen Servern kann es mehrere geben, welche die Simulationsumwelt regionsbasiert unter sich aufteilen. Dabei liegt auf jedem Server eine vollständige Kopie der Simulationsumgebung, jedoch ist jeder AS nur für einen regionalen Bereich zuständig. Die AS sind dafür zuständig, die Aktionen der Agenten zu

verifizieren und die entsprechenden Änderungen in die Simulationsumgebung zu schreiben. Die *Client-Server* (CS) beherbergen die eigentlichen Agenten, welche die Simulationsarbeit verrichten.
Jeder CS ist mit einem oder mehreren AS verbunden, sodass seine Agenten die Aktionsausführungen beantragen können. Die Aktionen der Agenten werden vor ihrer eigentlichen Ausführung vom jeweiligen AS überprüft und erst danach per Broadcast an alle anderen Mitglieder des Verbundes publiziert. So wird z.B. bereits im Vorfeld verhindert, dass Agenten miteinander kollidieren. Diese Verifikation einer Aktion dauert jedoch entsprechend länger, da jedes Mal ein externer Server befragt werden muss. Getestet wurde diese Architektur mit 12 Servern (4 AS + 8 CS) und 20.000 Agenten. Die verwendeten Agenten waren jedoch sehr primitiv gehalten, sodass in einer praktischen Simulation mit deutlich weniger Agenten zu rechnen ist bzw. mehr CS nötig wären.

2.2. Verteilte Systeme

Ein verteiltes System besteht aus mehreren physikalisch voneinander getrennten Rechenknoten, die ihre Arbeit zunächst unabhängig voneinander verrichten und dafür ihre eigenen Rechenwerke und Speichersysteme besitzen. Um diese Einzelsysteme trotzdem in einem Verbund rechnen zu lassen, müssen diese koordiniert und synchronisiert werden.
Im Folgenden werden die Hauptprobleme verteilter Systeme erklärt und gängige Lösungen vorgestellt.

2.2.1. CAP-Theorem

Verteilte Systeme werden hauptsächlich eingesetzt, weil entweder ein einziger Host technisch nicht genügend Rechenleistung aufbringen kann, um eine Aufgabe zu lösen, oder um sicherzustellen, dass trotz des technischen Ausfalles eines Teilsystems das Gesamtsystem weiterhin verfügbar bleibt. Da die Funktion eines solchen Rechnernetzes jedoch eine Kommunikation unter den einzelnen Hosts erfordert, führt das zu einem Problem, welches auch als CAP-Theorem bekannt ist (vgl. [25]).

Dieses besagt, dass es in einem verteilten System nicht möglich ist, alle der drei Eigenschaften zu erfüllen:

- **Consistency**: Alle Teilnehmer sehen zur gleichen Zeit immer den gleichen Datenbestand
- **Availability**: Eine Anfrage wird garantiert in vertretbarer Zeit beantwortet (mit Erfolg oder Fehler)

- **Partition Tolerance**: Das System bleibt weiterhin funktionsfähig, selbst wenn ein Teil ausfällt

Diese Eigenschaften sind dabei nicht binär zu betrachten, sondern eher als graduelle Größen. Ein System kann eine geringe Verfügbarkeit bieten, wenn es nur langsam, aber garantiert antwortet oder eine schlechte Konsistenz besitzen, wenn ein Abgleich der einzelnen Systeme nur periodisch erfolgt.

Kann eine unbedingte Konsistenz vernachlässigt werden, so spricht man im Allgemeinen vom *BASE*-Prinzip[2] [36], welches die traditionellen Anforderungen des ACID-Prinzips[3] aufweicht, indem zugunsten höherer Performance auf hohe Konsistenz verzichtet wird.

Für ein verteiltes System muss deshalb die Priorität dieser drei Eigenschaften für eine konkrete Aufgabe festgelegt werden. Z.B. ist das Domain-Name-System des Internets ein System der AP-Kategorie, weil es eine extrem hohe Verfügbarkeit bei gleichzeitigem Ausfall von Teilkomponenten benötigt, die Konsistenz der Daten jedoch eine untergeordnete Rolle spielt. Das Buchungssystem einer Bank sollte im Gegenzug eine unbedingte Konsistenz wahren, selbst wenn dafür die Verfügbarkeit im Fehlerfall eingeschränkt ist.

2.2.2. Konsistenz in verteilten Systemen

In einem verteilten System, welches Daten repliziert, gibt es immer ein Konsistenzproblem. Wird ein Datum an einer Stelle im System geändert, so unterscheidet sich diese Replik des Datums in diesem Moment von anderen Repliken. Das kann nicht nur dazu führen, dass andere Prozesse auf einen nicht mehr aktuellen Wert zugreifen, es kann außerdem passieren, dass ein Prozess eines anderen Systems, welcher auf eine andere Replik des gleichen Datums zugreift, diese ebenso verändert. Das kann dazu führen, dass zwei Prozesse das gleiche Datum auf verschiedenen Hosts ändern, die Schreibzugriffe aber wegen falscher Zeitsynchronisation vertauscht werden. Für die Klassifizierung dieses Konsistenzproblems wurden mehrere Stufen definiert, welche beschreiben, wie konsistent sich ein bestimmtes System verhält (vgl. [15, S. 340 ff]):

- Die **strenge Konsistenz** besagt, dass eine Leseoperation auf ein Datum immer den zuletzt geschriebenen Wert zurück gibt, egal auf welche Replik des Datums konkret geschrieben und gelesen wurde. Da es in verteilten Systemen grundsätzlich keine eindeutige globale Zeit gibt, ist die Implementierung der strengen Konsistenz jedoch sowohl theoretisch als auch praktisch unmöglich.

[2] **B**asically **A**vailable, **S**oft state, **E**ventual consistency
[3] **A**tomicity, **C**onsistency, **I**solation, **D**urability

- Die **sequentielle Konsistenz** bedeutet, dass alle in einem System teilnehmenden Prozesse alle ausgeführten Schreiboperationen in der gleichen Reihenfolge sehen. Das muss nicht zwingend bedeuten, dass diese auch zeitlich sequentiell ausgeführt wurden. Es gibt keine Auskunft über die “letzte” Schreiboperation.
- Die **linearen Konsistenz** ist eine Verschärfung der sequentiellen Konsistenz, bei der die Schreiboperationen von allen Prozessen in der gleichen Reihenfolge gesehen werden und zudem in einer zeitlichen Reihenfolge geordnet werden. Diese zeitliche Reihenfolge wird durch synchronisierte Uhren (z.B. eine Vektoruhr) realisiert.
- Die **kausale Konsistenz** stellt eine Abschwächung der sequentiellen Konsistenz dar. Hier wird eine kausale Abhängigkeit von Schreiboperationen erstellt. In dieser Konsistenz wird nur garantiert, dass Schreiboperationen, die kausal voneinander abhängig sind, von allen Prozessen in der gleichen Reihenfolge gesehen werden. Bei Schreiboperationen, die völlig unabhängig voneinander sind, weil z.B. zwei unabhängige Objekte geschrieben werden, wird keine konkrete Reihenfolge garantiert.
- Die **FIFO-Konsistenz** ist eine weitere Abschwächung der kausalen Konsistenz, bei der auch der kausale Zusammenhang der Schreiboperationen verworfen wird. Hier wird nur garantiert, dass Schreiboperationen, die von einem Prozess ausgeführt werden, von allen anderen Prozessen in der gleichen Reihenfolge gesehen werden, wie der schreibende Prozess sie ausgeführt hat.
- Die **Freigabekonsistenz und Eintrittskonsistenz** sind die losesten Konsistenzen für gemeinsame Variablen. Diese benötigen zusätzliche Programmierkonstrukte wie *Sperren* oder *Mutexe*. Hier wird garantiert, dass jede Variable durch einen Prozess im System für eine Schreiboperation gesperrt werden kann. Die Synchronisierung aller Repliken erfolgt dabei entweder direkt vor (Eintritt) oder direkt nach (Freigabe) der Sperre.

Neben diesen Konsistenzen gibt es noch weitere deutlich schwächere Konsistenzen, die nur noch auf einen Prozess begrenzt sind. So garantiert z.B. die monotone Lese-Konsistenz, dass alle Leseoperationen eines Prozesses immer den gleichen oder einen aktuelleren Wert eines Objektes sehen, als die vorige Operation, aber nie einen älteren. Die monotone Schreib-Konsistenz besagt, dass sich Schreiboperationen auf eine lokale Replik eines Objektes von einem Prozess nicht überschneiden dürfen und nacheinander ausgeführt werden müssen. Das erinnert sehr an die FIFO-Konsistenz, mit dem Unterschied, dass hier nur lokale Prozesse FIFO-Konsistenz sind, nicht das gesamte verteilte System.

Grundsätzlich ist zu sagen, dass hohe Konsistenzanforderungen die Leistung des Gesamtsystems stark beeinträchtigen können und generell immer das schwächste Konsistenzmodell zu wählen ist, welches die von der jeweiligen Anwendung geforderten Eigenschaften gerade so noch erfüllt.

Bei Anwendungen wie Simulationen kann es manchmal auch sinnvoll sein, auf bestimmte Konsistenzanforderungen zu verzichten, weil die Inkonsistenz von Daten nur eine sehr geringe Auswirkung auf das Gesamtergebnis hat, aber dafür die Komplexität des Systems deutlich geringer und die Leistung somit deutlich höher ist. So spielt es meist keine Rolle, ob ein Agent nun die tatsächlich aktuelle Position seines Nachbarn sieht, oder die vorige, um ein paar Zentimeter verschobene Position.

2.2.2.1. Verteilte Uhren

Ein wichtiger Punkt bei der Synchronisierung und Konsistenz in verteilten Systemen sind korrekte Zeitstempel. Nur wenn sich alle Systeme über die Zeit in einem System einig sind, können Lese- und Schreibzugriffe entsprechend der bereits beschriebenen Konsistenzen gehalten werden.

Die Zeit in einem verteilten System gleich der realen Zeit zu halten, ist dabei ein sehr kompliziertes Problem, da alleine schon die technischen Übertragungverzögerungen einen perfekten Uhrenvergleich unmöglich machen.
Simulationsprogramme, wie sie in dieser Arbeit behandelt werden, sind jedoch nicht darauf angewiesen, dass die lokalen Zeiten mit der realen Zeit übereinstimmen. Hier ist es lediglich wichtig, dass die Ereignisse, die in der Simulation auftreten, in einer kausal chronologische Reihenfolge geordnet werden können.

Lamport-Zeitstempel

Uhren, für die nicht die reale Zeit wichtig ist, sondern lediglich die Reihenfolge von Ereignissen, heißen logische Uhren. Eine einfache logische Uhr ist die Lamport-Uhr (vgl. [15, S. 289 ff]). Diese beschreibt die aktuelle Zeit nicht mit Hilfe von Sekunden sondern mit Hilfe eines Ganzzahlwertes, welcher bei jedem auftretenden Ereignis um einen Schritt erhöht wird. Dabei gibt es keinerlei Zusammenhang zum realen Zeitgeber des rechnenden Hosts.
Zu Beginn einer Simulation startet jeder Host seinen eigenen Lamport-Zähler bei null. Dieser Zähler wird bei jedem lokal auftretenden Ereignis um eins erhöht, sodass die Ereignisse lokal in eine chronologische Reihenfolge gebracht werden können. Empfängt ein Host eine Nachricht von einem anderen Host, so schickt dieser in der Nachricht den Zeitstempel dieser Nachricht mit. Der Empfang der Nachricht erhöht beim lokalen Host ebenfalls die eigene Uhr. Ist der Zeitstempel der Nachricht jedoch höher als der aktuelle lokale Zeitstempel, so wird der lokale zuerst mit dem empfangenen gleich gesetzt. Alle folgenden Ereignisse finden dementsprechend chronologisch nach dem Empfang der Nachricht statt.

Mit dieser Uhr können alle Nachrichten, die zwischen zwei oder mehreren Hosts ausgetauscht werden, in eine chronologische Reihenfolge gebracht werden. Fand ein Ereignis a vor dem Ereignis b statt, so stehen diese in einer *passiert-vor* Relation, entsprechend $a \rightarrow b$. Dementsprechend gilt für die Uhr C ebenso $C(a) < C(b)$.
Zwei Ereignisse x und y, welche auf verschiedenen Hosts auftreten, die keinerlei Kontakt zueinander haben, können jedoch in keinen Zusammenhang gebracht werden, da die lokalen Uhren niemals miteinander synchronisiert wurden.

Vektoruhr

Der Nachteil der Lamport-Zeit besteht jedoch darin, dass man nicht ablesen kann, ob zwei Ereignisse kausal voneinander abhängig sind, da aus den jeweiligen Zeitstempeln $C(a)$ und $C(b)$ von zwei Ereignissen verschiedener Hosts nicht zwingend auf eine reale Reihenfolge geschlossen werden kann.

Die Vektoruhr erweitert deshalb die Lamport-Uhr (vgl. [15, S. 293 ff]). Im Unterschied zur Lamport-Uhr besteht ein Zeitstempel nicht mehr nur aus einer einzigen Uhr sondern aus den einzelnen Uhren der Hosts. Dabei erhöht jeder Host stets nur seine eigene Uhr.

Nun kann für zwei Ereignisse die kausale Relation ermittelt werden, indem alle Teiluhren des Zeitstempels verglichen werden. Sind alle Teilzeitstempel von $VT(a)$ kleiner oder gleich dem entsprechenden Zähler in $VT(b)$ und mindestens ein Zähler strikt kleiner, gilt $a \rightarrow b$. Kann durch diesen Vergleich weder auf $a \rightarrow b$, noch auf $b \rightarrow a$ geschlossen werden, so sind die Ereignisse nebenläufig und vollständig voneinander unabhängig.

2.2.3. Synchronisierung und Zeit am Beispiel von HLA

Die Synchronisierung der verteilten Simulation bezeichnet den Vorgang, bei dem der Datenbestand der Simulation zwischen den einzelnen Hosts abgeglichen wird. Dafür kann entweder eine vollständige Synchronisation durchgeführt werden, bei der jeder Host den Gesamtzustand abbildet oder eine partielle Synchronisation, bei der jeder Host nur Teile des Zustandes abbildet. Ein Datum kann jedoch weiterhin auf mehreren Hosts als Kopie existieren. Bei dieser Synchronisation müssen zuvor bestimmte zeitliche Konsistenzregeln eingehalten werden.

Eine Möglichkeit, diese Synchronisierung durchzuführen, wurde mit der *High-Level-Architecture* HLA [21] definiert. HLA wurde dafür entworfen, mehrere einzelne Simulationen in einem verteilten Netzwerk miteinander zu verbinden. Dafür wurden zwei Hauptkomponenten definiert:

Federates sind Schnittstellen zu den eigentlichen Simulationen, welche miteinander verbunden werden sollen. Diese können intern voneinander unabhängig sein, müssen nach außen hin jedoch ein einheitliches Datenmodell, das *Simulation Object Model*, implementieren.

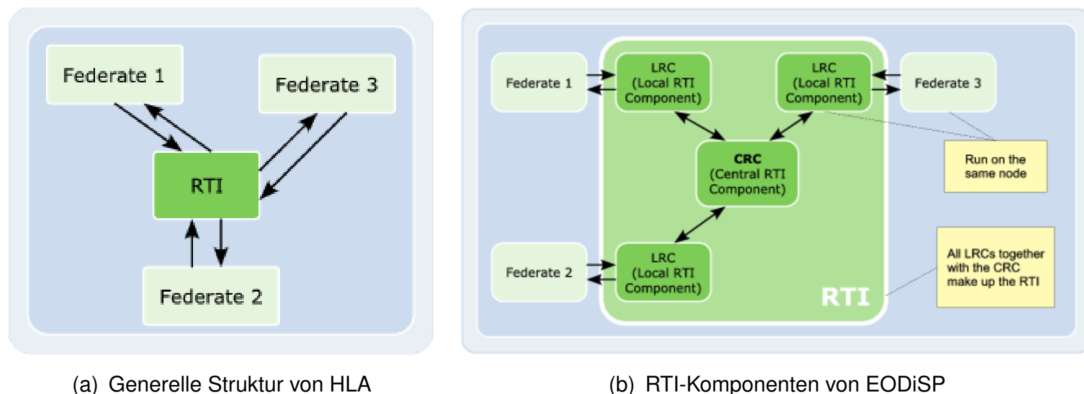
RTI Die **R**un**T**ime-**I**nfrastructure ist die Middleware, die zwischen den einzelnen Federates vermittelt und sowohl Zeit- als auch Datenmanagement übernimmt und dementsprechend die verteilte Simulation steuert.

Die einzelnen Federates in HLA kommunizieren über Nachrichten miteinander. Jede Nachricht hat einen logischen Zeitstempel. Dieser ist ein skalarer Wert ähnlich der Lamport-Zeit. Die RTI stellt diese Nachrichten nun den entsprechenden Federates abhängig von der individuell gewählten Ereignissortierung zu (vgl. [22]):

- **Empfangsreihenfolge:** Ereignisse werden schlicht in eine FIFO-Warteschlange eingefügt und dem Federate in der Reihenfolge zugestellt, wie sie empfangen wurden.
- **Prioritäts-Reihenfolge** Die Nachrichten werden geordnet nach dem Zeitstempel zugestellt. Dies verhindert nicht, dass auch Nachrichten aus der Vergangenheit zugestellt werden können.
- **Kausale Reihenfolge:** Die Nachrichten werden in einer kausalen Reihenfolge zugestellt. Die kausale Reihenfolge wird entsprechend der Lamport-Zeit ermittelt.
- **Kausale und totale Reihenfolge:** Nachrichten werden in einer kausalen Reihenfolge zugestellt. Zusätzlich dazu werden kausal unabhängige Ereignisse an alle Federates in der gleichen Reihenfolge zugestellt.
- **Zeitstempelreihenfolge (TSO):** Ereignisse werden in Reihenfolge des jeweiligen Zeitstempels zugestellt. Es wird von Grund auf verhindert, dass für einen Federate Ereignisse auftreten können, die im Bezug auf dessen aktuelle Zeit in der Vergangenheit liegen. Zusätzlich wird die kausale und totale Reihenfolge gewahrt.

Die einzelnen Federates können ihren Zeitfortschritt mit den *Time Management Services* separat voneinander steuern. Dazu kann jeder Federate einen Zeitfortschritt bei der RTI beantragen. Dieser Zeitfortschritt steuert, welche Ereignisse dem Federate zugestellt werden. Dies ist zudem abhängig von der gewählten Zustellreihenfolge. Im Fall von TSO werden z.B. keine Ereignisse mehr zugestellt, die in der Vergangenheit des Federates liegen. Bei TSO muss jeder Federate zusätzlich ein Lookahead-Fenster in die Zukunft definieren. Alle Nachrichten, deren Zeitstempel in diesem Fenster liegen, können dem Federate trotzdem zugestellt werden.
Abhängig von der Zustellreihenfolge gewährt die RTI dementsprechend einen Zeitfortschritt nur, wenn sie garantieren kann, dass alle vor dem Zeitpunkt t des Zeitfortschritts aufgetretenen Nachrichten zugestellt worden sind. So liegen z.B. bei TSO die einzelnen Federates auf der Zeitachse sehr dicht zusammen.

2.2.3.1. Datensynchronisierung



(a) Generelle Struktur von HLA

(b) RTI-Komponenten von EODiSP

Abbildung 2.2.: Architektur in der HLA [2]

Der Datenaustausch findet in HLA als partielle Synchronisation statt [21, S. 146]. Dafür wird ein zweidimensionaler *Routing Space* definiert, der sich über die normalisierten Koordinaten [0.0|0.0] bis [1.0|1.0] erstreckt. Ein Federate kann nun einen beliebigen Bereich in diesen Routing Space abonnieren und empfängt alle Änderungen, die diesem Bereich zugeordnet werden können. Zusätzlich können die Daten lokal auf dem Host des Federates weiter nach bestimmten Attributen gefiltert werden.

Die eigentliche Implementierung der RTI ist nicht in der Spezifikation vorgegeben, sodass mit der Zeit viele teils freie, teils kommerzielle RTI-Implementierungen entstanden sind.
Eine Implementierung ist die *EODiSP HLA Architecture* [2]. Diese definiert die RTI mit zwei Grundkomponenten. Die *Local RTI Component* (LRC) läuft auf den Hosts der einzelnen Federates. Die *Central RTI Component* (CRC) läuft auf einem zentralen Server und dient als zentrales Bindeglied der einzelnen LRCs. Jede Nachricht oder Aktualisierung eines Objektes wird zuerst an das CRC geschickt. Dort wird geprüft, welche LRCs Federates verwalten, die den entsprechenden Teil des Routing Spaces abonniert haben, und die Nachricht wird an die entsprechenden LRCs gesendet.

2.2.4. Verteilte Datenspeicher

In verteilten Simulationen muss der gesamte Simulationszustand ebenso verteilt werden wie die einzelnen Arbeitsaufgaben bzw. in unserem Fall die Agenten. Für die Organisation von Informationen in einem verteilten System gibt es drei grundsätzliche Möglichkeiten:

- Bei der **zentralen Datenhaltung** wird der gesamte Simulationszustand nur auf einem Host verwaltet. Alle anderen Hosts, die diese Daten lesen oder verändern wollen, müssen auf diesen Host zugreifen. Diese Variante ist sehr einfach, weil sie Synchronisation unnötig macht und sich Nebenläufigkeit leicht verhindern lässt. Bei der zentralen Datenhaltung kann der zentrale Datenserver bei größer werdenden Systemen schnell zu einem Flaschenhals werden.
- Die **replizierte Datenhaltung** kopiert den Gesamtzustand auf alle teilnehmenden Hosts. Jeder Host hat so direkten Lese- und Schreibzugriff auf alle Daten. Dadurch sind einerseits die Wege zu den Daten sehr kurz, allerdings gestaltet sich die Änderung von Daten sehr komplex, da eine Änderung mit allen Replikationen synchronisiert werden muss.
- Bei der **verteilten Datenhaltung** werden die Daten echt im System verteilt, sodass ein Datum nur auf wenigen oder gar nur einem Host verwaltet wird. Hier ist der Synchronisationsaufwand am geringsten, jedoch erfordert diese Datenverteilung ebenfalls einen hohen Verwaltungsaufwand. Zum einen muss ein Verzeichnis unterhalten werden, welches auf den konkreten Ort eines Datums verweisen kann, und zum anderen ist eine intelligente Verteilung nötig, die dafür sorgen muss, dass die Daten am besten dort verwaltet werden, wo sie auch gebraucht werden.

In den letzten Jahren und Jahrzehnten wurden viele verschiedene Arten von verteilten Datenspeichern entwickelt, welche sich neben der verwendeten Methodik und Technik auch in ihrem Einsatzgebiet unterscheiden und im Bezug auf das zuvor beschriebene CAP-Theorem verschiedene Prioritäten setzen.

Neben den fachlichen Anforderungen an eine solche Datenbank, die von Simulationsmodell zu Simulationsmodell verschieden sein können, sind mehrere technische Anforderungen extrem wichtig:

- Die Zugriffszeit auf ein Datum muss so gering wie möglich sein, da mehrere zehntausend Agenten gleichzeitig auf die Datenbank zugreifen werden.
- Aus der geringen Zugriffszeit leitet sich die Anforderung ab, trotz der Verteilung des Datenspeichers die Netzkommunikation möglichst gering zu halten, da jeder Netzwerkzugriff extrem viel Zeit kostet.

Gleichzeitig können einige andere klassische Datenbank-Anforderungen vernachlässigt werden:

- In erster Instanz ist es nicht nötig, die Daten stets persistent zu halten. Entsprechend reicht es, wenn die Daten im RAM des Hosts gehalten werden.
- Eine Replikation der Daten ist nicht nötig, da bei einem Ausfall eines Hosts höchstwahrscheinlich auch die Agenten des Hosts ausgefallen sind und so ein Neustart der Simulation nötig ist.

Aus den Anforderungen lässt sich ableiten, dass für Simulationen ein hochperformanter verteilter Datenspeicher gesucht wird, bei welchem die Ausfallsicherheit eine sehr geringe Rolle spielt.

Im Folgenden werden nun mehrere Architekturen solch verteilter Datenspeicher vorgestellt.

2.2.4.1. Relationale Datenbank-Management-Systeme

Der klassische komplexe Datenspeicher ist ein RDBMS[4]. In solchen Datenbanken können komplexe Datenstrukturen in miteinander verknüpften Relationen definiert werden und mit der Abfragesprache SQL in einer nahezu beliebigen Komplexität abgefragt werden. Der Zugriff wird mit verschiedensten Arten von Indizes beschleunigt.

In Bezug auf das CAP-Theorem stellen RDBMS hochverfügbare Datenspeicher dar, welche versuchen, möglichst alle drei Eigenschaften auf einmal zu erfüllen. Die verschiedenen Verteilungstechniken der verfügbaren RDBMS bieten unterschiedliche Gewichtungen in Bezug auf Verfügbarkeit und Partitionstoleranz, wobei die Konsistenz der Daten bei allen die höchste Priorität hat.

Die Softwarelandschaft bei SQL-Datenbanken ist aufgrund der großen Verbreitung weit gefächert. Big-Player sind hier die kommerziellen Produkte *Oracle 11g* [12] und Microsoft *MS SQL Server* [10] sowie die Open-Source Datenbanken *MySQL* [11] und *PostgreSQL* [13]. In Java-Anwendungen sind zudem *HSQL* [9] und *H2* [5] verbreitet, da es bei diesen Datenbanken möglich ist, sie in einer Embedded-Variante innerhalb der JVM zu verwenden, was die Zugriffszeit deutlich verringert.

Verteilung
Alle großen DBMS unterstützen die Funktionalität der Replikation. Dabei unterstützen alle Systeme viele Arten der Verteilung (Auszug, vgl. [8], [7]):

- **Master/Slave-Replikation**: Bei der Master-Slave-Replikation werden alle Daten auf alle teilnehmenden Server verteilt. Dabei dürfen schreibende Anfragen nur vom einzigen Master bearbeitet werden, Leseoperationen jedoch auch von allen Slaves. Fällt der Master aus, ist das System in dieser Zeit nur eingeschränkt einsatzfähig.
- **Cluster**: Bei einem SQL-Cluster sind alle teilnehmenden Server synchron geschaltet, d.h. alle Server haben das Recht auf ein Datum sowohl lesend als auch schreibend zuzugreifen. Transaktionen werden auf allen Servern gleichzeitig ausgeführt. Einige Konfigurationen erlauben zudem die Teilung der Daten in mehrere Partitionen, bei denen

[4] Relational Database-Management-System

die Daten nur auf ausgewählte Server gespiegelt werden. Diese Aufteilung kann jedoch nicht dynamisch geändert werden.

Das DBMS H2 unterstützt lediglich die Konfiguration des Clusters mit einer vollständigen Spiegelung, HSQL fehlt diese Unterstützung vollständig.

2.2.4.2. Key/Value-Stores

Neben den SQL-Datenbanken haben sich seit Ende der 90er Jahre sogenannte NoSQL-Datenbanken etabliert. Diese weichen vom bis dahin weit verbreiteten Schema der Relationen ab und speichern die Daten in Strukturen, welche jeweils für den konkreten Anwendungszweck passender sind als das klassische relationale Datenbankschema.

Eines dieser Konzepte ist der sogenannte Key/Value-Store. Key/Value-Stores speichern lediglich einen bestimmten Wert unter einem bestimmten, eindeutigen Schlüssel ab. Dabei ist der Schlüssel meist eine Zeichenkette, der Wert kann jedes beliebige Objekt sein. Im einfachsten Fall stellt die aus vielen Programmiersprachen bekannte *Map<String, Object>*[5] bereits einen vollständigen Key/Value-Store dar.
Wie der Name bereits vermuten lässt, unterstützen Key/Value-Stores meist nur die Abfrage über den einzigen Schlüssel. Komplexe Anfragen wie bei SQL-Datenbanken sind nicht möglich.

In vielen Key/Value-Stores besitzen die gespeicherten Objekte ebenfalls eine festgelegte Struktur, sodass sie effizient weiterverarbeitet werden können. Meist besteht ein Eintrag eines Key/Value-Stores ebenfalls aus einem kleinen Key/Value-Store, wo in per Zeichenkette adressierten Feldern die eigentlichen Daten abgelegt werden, sodass letztlich eine große zweidimensionale Tabelle entsteht.

Key/Value-Stores besitzen in der Regel nicht die Möglichkeit, Transaktionen über mehrere Objekte durchzuführen, da die für diesen Fall notwendigen verteilten Transaktionen die Komplexität des Systems unnötig erhöhen würden.

K/V-Stores mit Verzeichnisdienst

Eine Möglichkeit der Verteilung eines Key/Value-Stores ist mit Hilfe eines Verzeichnisdienstes. Hier werden die Daten nach einem vorgegebenen Prinzip auf die teilnehmenden Hosts verteilt und ein zentraler Host unterhält ein Verzeichnis, welches den Ort eines jeden Schlüssels speichert.

[5] In anderen Programmiersprachen auch *Dictionary* oder *Assoziatives Array*

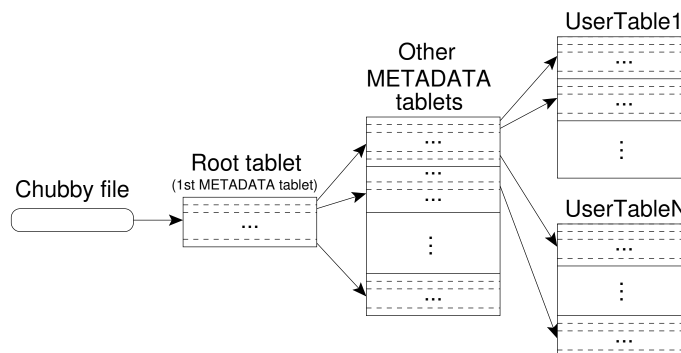


Abbildung 2.3.: Bigtable's Verteilungshierarchie [18, S. 4]

Ein Beispiel für einen solchen K/V-Store ist das von Google entwickelte *Bigtable* [18].
Hier wird die Gesamtmenge aller Einträge alphabetisch nach ihrem Key sortiert und in viele kleine Bereiche zerteilt. Diese *Tablets* genannten Bereiche können nun beliebig auf die Hosts des Systems verteilt werden. Dabei speichert ein mehrstufiger Verzeichnisdienst den konkreten Ort jedes Bereiches (siehe Abb. 2.3). Durch die alphabetische Sortierung kann so selbst für noch nicht vergebene Schlüssel schnell der zuständige Server ermittelt werden. Ebenso weist diese Verteilung bei passenden Schlüsseln ein sehr gutes Lokalitätsverhalten auf.

Über den Verzeichnisdienst ist ein Bereich zu einem Zeitpunkt lediglich nur einem einzigen Host zugewiesen, sodass auch nur ein Host die Daten verändern kann. In Googles Implementierung wird die Datensicherheit durch ein unterliegendes *Google Filesystem* [23] realisiert, welches selbst ein verteiltes, repliziertes Dateisystem ist. Der Tablet-Host agiert dabei lediglich als Cache. Fällt ein Host des K/V-Stores aus, kann dieser so durch einen anderen ersetzt werden.

Bigtable ist darauf ausgelegt, extrem viele parallele Lesezugriffe auf extrem große Datenmengen (viele Terabyte bis hin zu Petabyte) zu verkraften, die sich selten ändern. Transaktionen werden dabei lediglich für einzelne Einträge unterstützt. Dabei wird ein Versionierungssystem verwendet, mit welchem es möglich ist, ältere Versionen eines Datum wiederherzustellen.

Apache HBase [16] ist eine quelloffene Implementierung, die auf Basis von Bigtable von Facebook entwickelt wurde (vgl. [17]). HBase basiert nicht auf dem proprietären GFS, sondern auf dem verteilten Dateisystem *Hadoop* [39].

K/V-Stores als DHT

Eine weitere Verteilungsmöglichkeit von K/V-Stores ist die *Distributed Hash Table*. Im Gegensatz zu der vorigen Verteilung benutzt die DHT nicht den jeweiligen Schlüssel direkt zur Verteilung der Daten sondern einen Hashwert des Schlüssels. Dieser Hashwert ist ein möglichst chaotischer Wert, d.h. der kleinste Unterschied zwischen zwei Keys ergibt bereits einen völlig anderen Hashwert.

Für jeden Key wird dabei ein konstanter Hashwert errechnet, wobei die Menge aller möglichen Hashwerte als Ring betrachtet wird, d.h. der letzte mögliche Hashwert hat den ersten als Nachfolger. Jedem am Ring teilnehmenden Host wird zufällig ein Hashwert zugeordnet. Die zu speichernden Objekte werden nun jeweils dem Server zugeteilt, welchem der nächstgrößere Hashwert zugeordnet wurde. So ist ein Server immer für den Bereich an Keys verantwortlich, deren Hashwerte zwischen ihm und dem zugewiesenen Vorgängerhashwert liegen. Jeder Host kennt dabei stets seinen Nachfolger, sodass Anfragen entlang des Rings zum jeweilig zuständigen Host geroutet werden. Durch die verwendete Zufallsverteilung kann es passieren, dass einige Server mehr Keys verwalten müssen als andere. Dem wirkt die DHT entgegen, indem die Ring-Bereiche dynamisch verkleinert und vergrößert werden können. Dieser Algorithmus wird auch als Chord-Algorithmus bezeichnet [41].
Die Hosts tauschen sich regelmäßig asynchron untereinander über die Positionen aller bekannten Koordinator-Server aus. So weiß jeder Host, wo ein bestimmter Key verwaltet wird und es ist für jeden Zugriff in den meisten Fällen lediglich ein Routing-Schritt erforderlich.

Eine Implementierung eines solchen DHT-basierten K/V-Stores ist das von Facebook entwickelte *Cassandra* [29]. Es wurde von Facebook als Nachfolger von HBase entworfen und bis 2010 als Datenbank für das Facebook-Messaging benutzt.

Zusätzlich zum Hashring verwendet Cassandra eine zweite Ebene, welche sich um die Replikation von Daten kümmert. Der Hashring wird dabei nicht unter den eigentlichen Datenservern aufgeteilt, sondern lediglich unter Koordinator-Servern. Diese sind selbst dafür zuständig, die Daten entsprechend auf ihre Repliken zu duplizieren. Dafür stehen in Cassandra mehrere Replikationsstrategien zur Verfügung, welche sich jedoch lediglich darin unterscheiden, an welchen topologischen bzw. geografischen Orten sich die Repliken befinden (im gleichen oder anderen Server-Rack, im gleichen oder anderen Rechenzentrum).

2.2.4.3. Tuple-Spaces

Eine Erweiterung der Key/Value-Stores sind die Tuple-Spaces. Bei ihnen ist es nicht nur möglich, ein Schlüssel->Wert Paar zu speichern, sondern gleich komplette Tupel. Im Gegensatz zu

Key/Value-Stores können diese Tupelmengen nicht nur mit Hilfe des Primärschlüssels durchsucht werden, sondern anhand all ihrer Werte.
Tuple-Spaces können keine Primärschlüssel auf Tupel verwalten, sodass durchaus mehrere wertgleiche Tupel im Tuple-Space existieren können. In den meisten Implementierungen können einzelne Tupel jedoch anhand einer Tupel-ID identifiziert werden.

Die Verteilung findet dabei separat für jeden Tupel-Typ statt. Dafür wird explizit ein Feld des Tupels als Routing-Key definiert. Die eigentliche Verteilung der Tupel kann dann analog zu K/V-Stores mit verschiedenen Strategien erfolgen.

Eine verbreitete Implementierung für Tuple-Spaces ist *Gigaspaces* [24].

2.2.4.4. Weitere NoSQL-Systeme

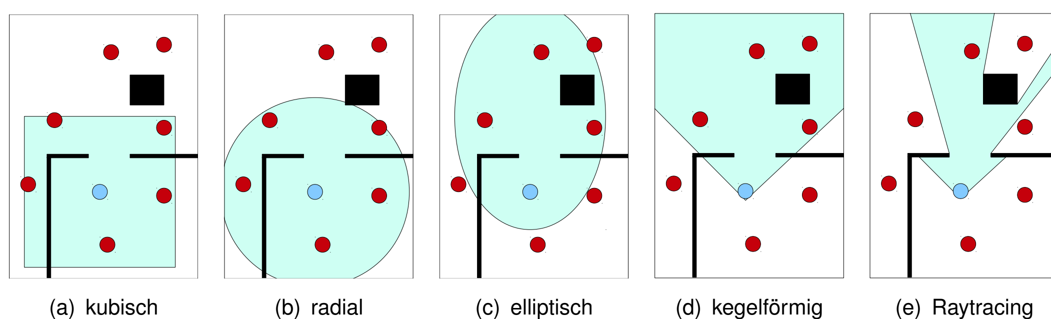
Neben den weit verbreiteten Key/Value-Stores existieren noch weitere NoSQL-Datenbankkonzepte, dessen Eignung jedoch im Vorfeld ausgeschlossen wurde und sie nur der Vollständigkeit halber erwähnt werden (vgl. [46]):

Document-Store Speichert ganze Dokumente ab und indiziert sie u.a. mit Tags. Geeignet z.B. für Wikis oder Literatursammlungen.

Graph-Datenbank Die Objekte werden als verknüpfter Graph gespeichert, sodass sehr schnell Verbindungen unter einzelnen Objekten gefunden werden können.

Objekt-Datenbank Strukturiert die Daten nach den Paradigmen der objektorientierten Programmierung und ähnelt noch am ehesten den SQL-Datenbanken.

2.2.5. Agenten-Sichtbereiche



(a) kubisch (b) radial (c) elliptisch (d) kegelförmig (e) Raytracing

Abbildung 2.4.: Verschiedene geometrische Formen von Sichtbereichen

Auch wenn die Sichtbereiche der Agenten auf den ersten Blick nur wenig mit einem verteilten System zu tun haben, besitzen sie einen großen Einfluss auf die nötige Datenverteilung.

Der Nutzen der partiellen Synchronisation besteht darin, dass jeder Knoten nur den Teil des Gesamtzustandes kennt und konsistent halten muss, den die lokalen Prozesse auch benötigen, um ihre Arbeit zu erledigen. Die lokalen Prozesse in WalkSim sind hauptsächlich die einzelnen Agenten und die benötigten Daten definieren sich aus den jeweiligen Sichtbereichen der Agenten.

Abb. 2.4 stellt mehrere gängige Formen dar, wie Sichtbereiche modelliert werden können.

kubisch Der kubische Sichtbereich ist mathematisch am einfachsten zu berechnen und war vor einigen Jahren auch noch häufig in 3D-Spiele-Engines anzutreffen. Auffallend ist seine untypische Form und dass Objekte hinter Wänden fälschlicherweise sichtbar sind.

radial Der kreisförmige Sichtbereich unterscheidet sich vom kubischen dadurch, dass er in alle Richtungen hin die gleiche Ausdehnung hat und sich somit unabhängig von der Blickrichtung in seiner Reichweite ändert.

elliptisch Der elliptische Sichtbereich ist in Blickrichtung in die Länge gezogen. Zudem ist sein hinterer Mittelpunkt vor den Agenten gelegt, damit die Sichtweite im Rücken eingeschränkt ist. Auch dieser Sichtbereich ist nicht durch Hindernisse begrenzt.

kegelförmig Der kegelförmige Sichtbereich ist eine Verschärfung des elliptischen Sichtbereiches, indem er die Sicht nach hinten komplett verhindert. Jedoch ist der Berechnungsaufwand bereits bedeutend höher als bei kubischen oder elliptischen Sichtbereichen.

Raytracing Der "perfekte" Sichtbereich ist der kegelförmige Sichtbereich, welcher Objekte, bei denen sich zwischen Agent und Objekt ein undurchsichtiges Hindernis befindet, ausschließt und so verhindert, dass der Agent durch Wände gucken kann. Allerdings ist das dafür notwendige *Raytracing* eine sehr aufwändige Operation, die nur in geringer Zahl durchgeführt werden sollte.

Die Form und Größe dieser Sichtbereiche beeinflusst nicht nur den nötigen Rechenaufwand, um die Menge der von einem einzelnen Punkt aus sichtbaren Objekte zu ermitteln. Vor allem die Größe beeinflusst den Nutzen der partiellen Synchronisation erheblich. Je weiter der Agent sehen kann, desto größer ist auch der Anteil am Gesamtzustand der Simulation, der auf der Agentenplattform lokal verfügbar sein muss.

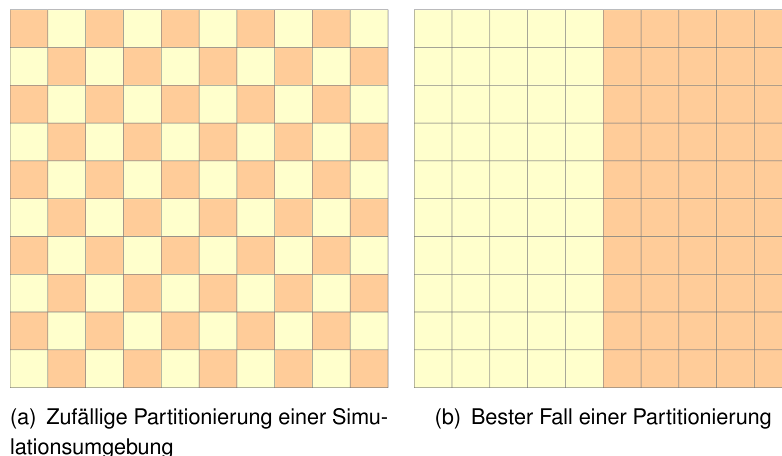
2.3. Partitionierung

Sobald in einem verteilten System Arbeit auf mehrere Teilnehmer verteilt wird, ist immer eine Partitionierung dieser Arbeit in kleinere Arbeitspakete erforderlich, welche auf die einzelnen Teilnehmer verteilt wird.

Bei Multiagentensystemen werden meist die Agenten verteilt. Bei der Verteilung der Agenten auf die teilnehmenden Rechner sind zwei Aspekte besonders wichtig:

- Die Kommunikation der einzelnen Teilnehmer untereinander soll so gering wie möglich gehalten werden, damit keine unnötigen Wartezeiten auf Netzwerkdatenverkehr entstehen.
- Die Auslastung aller Server sollte möglichst gleich groß sein (Load-Balancing).

2.3.1. Auswirkung verschiedener Partitionierungen



(a) Zufällige Partitionierung einer Simulationsumgebung

(b) Bester Fall einer Partitionierung

Abbildung 2.5.: Zwei mögliche räumliche Partitionierungen einer gleichmäßig ausgelasteten Simulation

Welche Auswirkungen eine schlechte Partitionierung auf eine Simulation hat, wird in Abb. 2.5 verdeutlicht. Angenommen, es befinden sich in jedem der dargestellten Kästchen 100 Agenten und der Sichtbereich der Agenten umfasst neben dem eigenen Kästchen auch alle 8 angrenzenden Kästchen.
Abb. 2.5(a) stellt nun eine Partitionierung dar, bei der die Bereiche der Reihe nach auf zwei Server verteilt wurden. Dabei befindet sich in der Nachbarschaft jedes Kästchen mindestens

ein Nachbar-Kästchen des jeweils anderen Servers, womit in Summe alle 5.000 Agenten des einen Servers von irgendeinem Agenten des anderen Servers gesehen werden. In Abb. 2.5(b) wurden die Bereiche als große Partitionen zusammenhängend auf zwei Server verteilt. Bei dieser Paritionierung besitzen lediglich die Kästchen in der Mitte Nachbarn des anderen Servers, was in Summe nur eine Überschneidung von jeweils 500 Agenten bedeutet.
Hier wird deutlich, dass die Partitionierung in Abb. 2.5(b) die deutlich bessere ist als in Abb. 2.5(a), weil der potentielle Kommunikationsaufwand um den Faktor zehn geringer ist.

Das Problem ist nun, dass es m^n Möglichkeiten gibt, n Agenten auf m Hosts zu verteilen. Die Berechnung einer perfekten Verteilung aller Agenten auf die verfügbaren Server ist folglich ein NP-vollständiges Problem und die tatsächliche Berechnung somit praktisch unmöglich (für Beweis vgl. [32]). Um diesen Aufwand auf einen polynomiellen Aufwand zu reduzieren, sind somit immer Abstriche nötig, da man sich der perfekten Verteilung entweder durch Heuristiken oder durch Modellvereinfachungen annähern muss.

2.3.2. Statische und dynamische Partitionierung

Bei der Partitionierung von Fußgängersimulationen besteht zusätzlich die Schwierigkeit, dass sich die Agenten während der Simulation frei bewegen können. Die Bewegung kann dazu führen, dass sich die erzeugte Last des Agenten mit der Zeit ändert, weil sich seine Wahrnehmungen im Laufe der Simulation ändern. Weiterhin ändern sich mit dem veränderten Sichtbereich auch die Kommunikationspartner, was zu einer Verschiebung der Kommunikationsstruktur innerhalb des Systems führt.

Die Partitionierung besteht deshalb grundsätzlich aus zwei Phasen:

- Bei der **initialen Partitionierung** muss der Partitionierer lediglich aufgrund eines statischen Bildes entscheiden, wie er die Last verteilt. Da zu diesem Zeitpunkt aber noch gar keine Arbeit verteilt und ausgeführt wurde, sind keine konkreten Daten über Prozessor- und Netzwerklast bekannt, sodass sämtliche Lastdaten reine Schätzwerte sind.

- Die **dynamische Partitionierung** kann während der Simulation hingegen auf konkrete Laufzeitdaten des Systems zurückgreifen und die Simulation entsprechend besser partitionieren.

Es ist möglich statische Partitionierungsalgorithmen auch für die dynamische Partitionierung einzusetzen. Abhängig vom Algorithmus besteht dort jedoch die Gefahr, dass jede Neupartitionierung eine gänzlich andere Verteilung erzeugt.

2.3.3. Vergleichbare Arbeiten

Die bisherigen Arbeiten im Bereich der Partitionierung von Fußgängersimulationen lassen sich in zwei Gruppen einteilen:

- Die **regionsbasierte Verteilung** benutzt als Grundlage den Raum aus dem die Simulation besteht und teilt ihn in kleinere Unterbereiche auf, welche auf die Server aufgeteilt werden. Alle Rechenarbeit, die innerhalb eines dieser Unterbereiche anfällt, wird auf dem Server ausgeführt, dem dieser Bereich zugeordnet ist.
- Die **prozessbasierte Verteilung** basiert rein auf den einzelnen logischen Prozessen, die innerhalb der Simulation ausgeführt werden (z.B. Agenten). Hierbei wird jeder Prozess aufgrund unterschiedlicher Lastparameter separat analysiert und einem entsprechenden Server zugeordnet. Die räumliche Aufteilung spielt dabei keine Rolle bzw. wird nur indirekt mit einbezogen.

2.3.3.1. k-D-Tree

Eine Variante der regionsbasierten Partitionierung ist das rekursive Teilen der Umwelt in immer kleinere Teilbereiche. Bei diesem Algorithmus wird der Gesamtraum zuerst in mehrere Teilräume unterteilt. Diese Teilräume werden daraufhin weiter geteilt, wenn die Last der einzelnen Bereiche weiterhin einen Grenzwert überschreitet und die Anzahl der Teilbereiche nicht die Anzahl der Server überschreitet.
In 2D-Räumen sind Quad-Trees und k-D-Trees die gängigsten Algorithmen (für weitere Informationen siehe [40]):

- Bei einem *Quadtree* wird jeder Teilbereich (meist ein Quadrat) in vier gleichgroße Bereiche geteilt. Der Vorteil des Quadtrees ist, dass er sich durch die gleichgroßen Unterbereiche sehr schnell und einfach aufbauen lässt. Nachteilig ist jedoch, dass bei jeder Spaltung zwingend drei weitere Partitionen entstehen. In einem Quadtree ist es z.B. nicht möglich, eine Anzahl von drei Partitionen zu erreichen, da bereits nach der ersten Teilung vier Partitionen existieren. Dies macht den Quadtree für Agenten-Partitionierungen ungeeignet, da hier die Zahl der Server beliebig sein muss.
- der k-D-Tree teilt einen Bereich nicht durch vier, sondern teilt ihn an einem Median so in zwei Bereiche, dass danach zwei gleich bewertete Teilbereiche entstehen. Dabei wird alternierend je nach Tiefe horizontal und vertikal geteilt. So kann jede Anzahl an Partitionen erreicht werden und diese zugleich durch die individuellen Grenzen ausgeglichen partitioniert werden.

```
1  Agent[] agents // Liste aller Agenten im System
2  Partition[] partitions // liste aller Partitionen
3
4  for ( i = 0 ; i < maxPartitions ; i++ ) {
5    // waehle zufaellig einen Agenten aus und erstelle eine Partition mit der
         Agentenposition als Zentrum
6    Agent randomAgent = chooseRandomAgent(agents)
7    partitions.add(new Partition(randomAgent.position))
8  }
9
10 i = 0
11 while (i < maxIterations && something changed in previous run) {
12   i++;
13   foreach ( agent : agents )
14     reassign agent to nearest partition by euklidian distance to partition center
15
16   recalculate all partition centers by average position of all its agents
17 }
```

Abbildung 2.6.: Pseudocode des KMeans-Algorithmus

Während der laufenden Simulation wird in diesem k-D-Baum nun nach Partitionen gesucht, die zuvor definierte Grenzen in ihrer verursachten Last unter- bzw überschreiten. Unterlastete Partitionen sollten dabei zusammengefasst, überlastete geteilt werden. Dabei muss die Anzahl der Partitionen konstant bleiben. So wird im Baum zuerst nach zwei Blatt-Knoten gesucht, die dem gleichen Eltern-Knoten angehören und zusammen eine Unterlast-Grenze unterschreiten und zusammengefasst werden sollen. Daneben wird ein weiterer Blattknoten gesucht, welcher einzeln bereits die Überlastgrenze überschreitet und geteilt werden sollte. Nur wenn beide Suchen mindestens einen Treffer ergaben, wird eine Partitionierung durchgeführt.

2.3.3.2. KMeans

Der KMeans-Algorithmus ist hauptsächlich im Datamining anzutreffen, wenn es darum geht Objekte ihrer Ähnlichkeit nach in Gruppen einzuteilen. Übertragen auf Fußgängersimulationen ist der Algorithmus eine prozessbasierte Partitionierung, die nur die Position des Agenten benutzt, um Agenten in Gruppen zu teilen. Er basiert auf der Annahme, dass die Inter-Server-Kommunikation minimiert wird, wenn Agenten, die geografisch dicht beieinander liegen, auf dem gleichen Server ausgeführt werden.

Der Algorithmus ist rein positions- bzw. entfernungsbasiert. Zu Beginn des Algorithmus werden aus der Menge der Agenten n Agenten zufällig ausgewählt und für jeden Agenten eine Partition mit der Position des Agenten als räumliches Zentrum erstellt (siehe Abb. 2.6). Danach erfolgt die eigentliche Verteilung der Agenten auf die Partitionen. In einer Schleife wird wiederholt jeder Agent der räumlich nächsten Partition zugewiesen. Am Ende jeder Iteration werden die

Zentren der Partitionen auf Basis der durchschnittlichen Position der Agenten der Partition neu berechnet. Diese Schleife wird solange wiederholt, bis sich das System eingeschwungen hat und in einer Iteration keine Agenten mehr in eine andere Partition verschoben werden oder bis eine maximale Zahl von Iterationen erreicht wurde.

Der Vorteil dieses iterativen Verfahrens ist, dass die Anzahl der Durchläufe leicht auf die Dauer der Simulation gestreckt werden kann. So kann der KMeans z.B. jede Sekunde auf Basis der dann aktuellen Positionen weitere Iterationen ausführen und kann so stets auf dem vorigen Partitionierungszustand aufbauen.

Der KMeans-Algorithmus bezieht weder die erzeugte Last der Agenten noch Kommunikationsstrukturen mit ein. Zudem ist das Ergebnis sehr von den initial zufällig ausgewählten Agenten abhängig und kann bei wiederholten Simulationen des gleichen Szenarios unterschiedliche Ergebnisse erzeugen. Der Algorithmus ist verhältnismäßig schnell, da der Aufwand pro Iteration lediglich $O(n)$ beträgt (n = Anzahl der Agenten). Wie viele Iterationen nötig sind, hängt neben der Agentenanzahl sehr stark vom Zufall und der Verteilung der Agenten in der Umgebung ab.

2.3.3.3. QHull

Im QHull-Algorithmus, entwickelt an der Universität von Valencia [45], wird davon ausgegangen, dass eine optimale Partitionierung anhand einer folgenden Fitnessfunktion erkannt werden kann:

$$H(P) = w_1 * \alpha(P) + w_2 * \beta(P), w_1 + w_2 = 1$$

Der erste Teil $\alpha(P)$ berechnet die Anzahl von Agenten, deren Sichtweite über die aktuellen Partitionsgrenzen hinaus geht (im Idealfall 0). Der zweite Teil $\beta(P)$ stellt die Standardabweichung von der durchschnittlichen Anzahl von Agenten pro Partition dar und soll für ausgewogene Partitionen sorgen (im Idealfall ebenfalls 0). Die Parameter w_1 und w_2 sind anpassbare Gewichtungen, mit denen der Einfluss der einzelnen Kennzahlen reguliert werden kann. Im Paper wurden für w_1 0.6 und für w_2 0.4 angenommen.

Der eigentliche Algorithmus verwendet eine nicht näher beschriebene Variante des kMeans-Algorithmus für die initiale Partitionierung. In der dynamischen Partitionierung während der Simulation wird nun in jedem Schritt für jeden Server ausgerechnet, wie stark seine Auslastung von der Durchschnittslast aller Server abweicht und die Server entsprechend in unterlastete und überlastete Server geteilt (siehe Abb. 2.7). Nun wird für jeden unterlasteten Server solange ein Lastausgleich durchgeführt, bis dessen Abweichung von der durchschnittlichen Serverlast gleich null ist. Dafür wird zunächst unter den direkt benachbarten, überlasteten Servern der Agent mit der geringsten euklidschen Distanz gesucht und der unterlasteten Partition

```
1  Server[] uServers // unterlastete Server
2  Server[] oServers // ueberlastete Server
3
4  simulationCycle() {
5    QHullPartitioning()
6    calculateServersLoad(uServers, oServers)
7    loadBalance(uServers, oServers)
8  }
9
10 loadBalance(uServers, oServers) {
11   while ( serverload imbalance ) {
12     foreach ( s : uServers ) {
13       while ( s.imbalance != 0 ) {
14         bestAgent = searchBestAgent(s.oNeighbors) // suche dichtesten Agenten in
               ueberlasteten Nachbarn
15         if ( bestAgent == NULL )
16           bestAgent = searchBestAgent(s.uNeighbors) // suche dichtesten Agenten in
                 unterlasteten Nachbarn
17         if ( bestAgent != NULL )
18           migrateAgent(bestAgent)
19       }
20
21       if ( agentesMigrated )
22         calculateServersLoad(uServers, oServers)
23     }
24   }
25 }
```

Abbildung 2.7.: Pseudocode des QHull-Algorithmus

hinzugefügt. Wird auf diese Weise kein passender Agent gefunden (z.B. weil alle Nachbarn unterlastet sind), wird die Suche auf alle Nachbarn ausgedehnt, deren last höher ist, als die eigene. Dieser Vorgang wird solange wiederholt, wie einer der Server unterlastet ist und ein Agent für die Migration gefunden wurde.
Dadurch dass der Algorithmus solange läuft, bis für alle Server die Abweichung zur Durchschnittslast null beträgt, sind zum Ende des Algorithmus alle Server gleich ausgelastet.

Der Aufwand von QHull ist sehr stark abhängig von der aktuellen Simulationssituation und davon wie viele Migrationen nötig sind, um ein Gleichgewicht herzustellen. Abhängig von der Implementation von $searchBestAgent()$, beträgt der Aufwand für jede Agentenmigration im schlimmsten Fall $O(n)$, weil jeder Agent der nicht auf dem eigenen Server läuft, für eine Migration in Frage kommen kann.

```
1  priority queue M =  {}
2
3  foreach (ag : agents, h : hosts) { // erstelle Migrationsindikatoren
4  dcomm <= comm(ag,h) - comm(ag,ag.host)
5  if ( dcomm > 0 ) { // Verschiebung macht Sinn
6    m =  (ag, host(ag), h, dcomm) // Migrationsindikator
7    M.add(m)
8  }
9  foreach ( m : M ) {
10   src <= sourceHost(m)
11   dst <= destHost(m)
12   ag <= agent(m)
13
14   if ( migration is possible ) { // nur moeglich, wenn nicht schon Agenten in die
       andere Richtung migriert wurden und keine zu grosse Ueberlast erzeugt wird
15     Plane Agentenmigration
16     load_src <= load_src - ag.load
17     load_dst <= load_dst + ag.load
18   }
19 }
20 migrate agents
```

Abbildung 2.8.: Pseudocode des Peschlow-Algorithmus (Teilalgorithmus für Kommunikationsoptimierung

2.3.3.4. Partitionierung nach Peschlow

Peschlow et al. [34] teilt seinen Partitionierungsalgorithmus in zwei Teile, welche unabhängig voneinander alternierend ausgeführt werden:
Der erste Teilalgorithmus optimiert die Kommunikationskosten (siehe Abb. 2.8). Hier wird für jeden Agenten ausgerechnet, mit welchem Server er das größte Kommunikationsaufkommen hat und berechnet, wie sich die Gesamtkommunikationskosten des Agenten zu Fremdservern ändern, wenn er zu einem der anderen Server migriert wird. Diese Indikatoren werden der Größe nach sortiert. Nun werden die Agenten-Server-Paarungen mit der besten Kostenänderung gewählt und die Agenten entsprechend migriert, sofern eine Migration möglich ist. Eine Migration ist möglich, wenn dadurch kein zu großes Lastungleichgewicht entsteht und nicht zuvor bereits ein Agent zwischen den beiden Servern in die andere Richtung migriert wurde.

Der zweite Teilalgorithmus führt eine reine Lastverteilung der Agenten zwischen den verfügbaren Servern durch (siehe Abb. 2.9). Dabei wird nicht jeder Agent gleich gewichtet, sondern die Anzahl der verarbeiteten Ereignisse im Vergleich zur dafür benötigten Zeit als Indikator genommen. Ist das Verhältnis von Rechenzeit zu Ereignissen über dem Gesamtdurchschnitt der Simulation, so ist der Server überlastet, ist es darunter, hat er noch freie Kapazitäten. In der Optimierung werden nun überlastete Plattformen zu entlasten versucht. Diese Entlastung basiert dabei unter anderem auf den Ergebnissen der Kommunikationsoptimierung, d.h. es

```
1   priorityQueue M[][] =  {}
2
3   foreach ( h : hosts ) // berechne Lastdifferenz
4     h.dload = h.load - h.cap
5   foreach ( pairs of hosts (src, dst) )
6     priority queue M[src][dst] = {}
7   foreach ( ag : agents, h : hosts ) {
8     if ( ag.host.dload > 0 && h.dload < 0 ) {
9       dcomm = comm(ag,h) - comm(ag,ag.host)
10      m = (ag, ag.host, h, dcomm) // Migrationsindikator
11      M[ag.host][h].add(m)
12  }
13  while ( some host over- or underloaded) {
14    (src, dst) = (host(max(h.dload), host(min(h.dload)) // suche ueberlastetsten und
          unterlastetsten host
15    m = poll(M[src][dst])
16    ag = m.agent
17    if ( migration is possible ) {// nur moeglich, wenn nicht schon Agenten in die
          andere Richtung migriert wurden und keine zu grosse ueberlast erzeugt wird
18      plan agent migration
19      load_src <= load_src - ag.load
20      load_dst <= load_dst + ag.load
21    }
22  }
23  migrate agents
```

Abbildung 2.9.: Pseudocode des Peschlow-Algorithmus (Teilalgorithmus für Lastausgleich)

werden bevorzugt Agenten migriert, die häufiger mit anderen, freieren Plattformen kommunizieren. Im Unterschied zur reinen Kommunikationsoptimierung wird akzeptiert, dass Agenten auf einem kommunikationstechnisch ungünstigeren Server laufen, sodass die Herstellung eines Lastausgleichs eine höhere Priorität hat.

Von diesen beiden Algorithmen wird bei jedem Partitionierungsschritt alternierend jeweils einer ausgeführt. Die Algorithmen stehen ansonsten in keinem näheren Zusammenhang, sodass es dazu kommen kann, dass die Migration des einen Algorithmus direkt darauf vom jeweils anderen wieder rückgängig gemacht wird. Der Aufwand der Algorithmen beträgt jeweils $O(nm+o)$, (n = Anzahl der Agenten, m = Anzahl der Plattformen, o Anzahl der zu untersuchenden Migrationsindikatoren).

Auf die initiale Partitionierung wird in der Arbeit nicht näher eingegangen, es wird jedoch gesagt, dass die Partitionierung auf eine Lastverteilung achtet, die vermutlich rein auf Basis der Agentenanzahl basiert.

```
1  priorityQueue M[][] =  {}
2
3  foreach ( h : hosts ) // berechne Lastdifferenz
4    h.dload = h.load - h.cap
5  foreach ( pairs of hosts (src, dst) )
6    priority queue M[src][dst] = {}
7  foreach ( ag : agents, h : hosts ) {
8    if ( ag.host.dload > 0 && h.dload < 0 ) {
9      dcomm = comm(ag,h) - comm(ag,ag.host)
10     m = (ag, ag.host, h, dcomm) // Migrationsindikator
11     M[ag.host][h].add(m)
12 }
13 foreach ( pair (src,dest) : hosts ) {
14   // n = Maximalanzahl an Migrationen zw. Hosts
15   top_src = top(M[src][dest], n) // guenstigste n Migrationen von src nach dest
16   top_dest = topn(M[dest][src], n)// guenstigste n Migrationen von dest nach src
17   cut list length to the length of the shortest
18   plan migration of all agents in both lists
19 }
20 migrate agents
```

Abbildung 2.10.: Pseudocode des Takahashi-Algorithmus

2.3.3.5. Partitionierung nach Takahashi

Die Partitionierung von Takahashi et al. [42] basiert weitgehend auf der von Peschlow et al. Im Unterschied zum Algorithmus von Peschlow versucht Takahashi, einen Lastausgleich vom Design her nicht notwendig zu machen, da die Anzahl der Agenten auf allen Servern konstant bleibt. Dafür werden initial alle Agenten beliebig auf alle Server verteilt, wobei alle Server die gleiche Anzahl an Agenten ausführen müssen. Danach wird für jeden Partitionierungsschritt analog zur Kommunikationskostenoptimierung von Peschlow für jedes Server-Paar eine Liste von möglichen Migrationen erstellt und prozessiert. Anders als bei Peschlow muss es hier zwingend für beide Migrationsrichtungen eines Serverpaares Migrationskandidaten geben, damit eine Migration durchgeführt werden kann. Ein Agent kann nur von Server A nach Server B migriert werden, wenn parallel dazu auch ein Agent von Server B nach Server A umziehen kann und dabei die Gesamtkommunikationskosten des Systems senkt.

Durch diesen Austausch ist es nicht notwendig, einen zusätzlichen Lastausgleich durchzuführen, wenn man voraussetzt, dass alle Agenten jeweils die gleiche, konstante Rechenlast verursachen.

```
1  Cell[] cells // alle Tellen des Environments
2  Partition[] partitions // alle Partitionen
3
4  doPartitioning(cells) {
5    add all cells to single partition
6
7    while ( numPartitions < targetNum ) {
8      greatestPartition = max(partitions) // suche Partition mit den meisten Agenten
9      (cell1, cell2) = getMostDistantCells(greatesPartition)
10     newPartition(cell2) // erstelle leere neue Partition
11
12     // schiebe solange Zellen von A nach B, bis beide Partitionen gleich viele
          Agenten haben
13     while (greatestPartition.numAgents > newParttition.numAgents ) {
14       Cell best = chooseBestNeighborCell(greatesPartition, newPartition)
15       greatesPartition.remove(best)
16       newPartition.add(best)
17     }
18   }
19
20   while ( any partition.load < lowerThreshold ) { // Abbruch, wenn nichts mehr
        verschoben wird
21     uPartition = min(partitions) // suche unterlastete Partition
22
23     // schiebe solange, bis Partition nicht mehr unterlastet ist
24     while ( uPartition.load < lowerThreshold) {
25       Cell best = chooseBestNeighborCell(partitions, oPartition)
26       partitions(cell).remove(best)
27       newPartition.add(best)
28     }
29   }
30 }
```

Abbildung 2.11.: Pseudocode des Bisection-Algorithmus von Lui

2.3.3.6. Bisection

Der Bisection-Algorithmus von Lui et al. [32] ist ein regionsbasierter Algorithmus, welcher auf einem Graphen basiert.

Bei der initialen Partitionierung wird die Umgebung zunächst in viele Regionen, in diesem Fall Zellen, unterteilt (siehe Abb. 2.11). Jede Zelle besitzt ein Gewicht anhand der Anzahl der Agenten, die sich räumlich in ihr befinden. Die Kommunikation der Agenten untereinander wird auf die Zellen aggregiert und als gewichtete Kanten zwischen den Zellen dargestellt, die Anzahl aller Nachrichten zwischen zwei Zellen stellt dabei die Gewichtung dar.
Um den Raum zu partitionieren, werden nun alle Zellen einer Partition zugeordnet. Daraufhin werden die beiden Zellen der Partition ermittelt, welche den größten Abstand zueinander haben und mit einer der beiden eine neue Partition erstellt. Nun werden solange Zellen der

alten Partition in die neue Partition verschoben, bis bei beiden Partitionen das Knotengewicht ausgeglichen ist.
Diese Teilung wird fortlaufend mit der jeweils größten Partition durchgeführt, bis die gewünschte Anzahl an Partitionen erreicht ist. Es ist leider nicht spezifiziert, wie die Funktion $chooseBestNeighborCell()$ implementiert ist, sodass weder über die Qualität noch den Aufwand geurteilt werden kann. Die Implementierung der Funktion $getMostDistantCells()$ ist ebenfalls nicht spezifiziert, jedoch besitzt sie mindestens einen Aufwand von $O(n^2)$, weshalb die initiale Erstellung der Partitionierung bei vielen Zellen einen erheblichen Aufwand darstellt.

Nach der initialen Partitionierung folgen inkrementelle Aktualisierungen basierend auf dem aktuellen Simulationszustand. In diesen werden Zellen zwischen zwei Plattformen verschoben, wenn sich dadurch die Gesamtkosten beider Plattformen reduzieren. Die Kosten sind dabei durch folgende Funktion definiert:

$$C_P = w_1 C_P^W + w_2 C_P^L$$

Die Gesamtkosten C_P einer Partition ergeben sich aus der gewichteten Summe der Lastkosten C_P^W und der Kommunikationskosten C_P^L, wobei w_1 und w_2 Gewichtungsfaktoren mit der Summe 1 sind (in den in deren Arbeit aufgeführten Experimenten wird für beide Faktoren stets 0.5 als Wert verwendet). Wie auch bei vorigen Algorithmen gilt auch hier, dass für jede Iteration Agenten zwischen zwei Servern stets nur in eine Richtung migriert werden können, um zu verhindern, dass ein Agent unendlich zwischen zwei Partitionen hin und her verschoben wird.

3. Konzept

Für die Klärung der zu Beginn aufgezeigten Fragestellungen soll die Multiagenten-Simulationsplattform *WalkSim* entworfen werden. Zusätzlich soll mit WalkSim im weiteren Verlauf des Forschungsprojektes eine universelle Plattform für Fußgängersimulationen auf Basis von Agenten entwickelt werden. Hauptsächlich soll WalkSim später für die Ausführung von Entfluchtungsanalysen genutzt werden, um beim Entwurf von Gebäuden bzw. Notfallplänen zu helfen oder bei der Organisation von Großveranstaltungen zu unterstützen.
Dazu soll WalkSim auch als Forschungsplattform für verschiedenste agentenbasierte Simulationsmodelle dienen, auf der unterschiedlichste KI-Modellierungen getestet werden können.

In dieser Arbeit liegt die Aufmerksamkeit erst einmal auf der verteilten Architektur an sich und der Möglichkeit Simulationen auszuführen und zu steuern. Die Architektur soll auf spätere Erweiterungen vorbereitet werden, die im weiteren Verlauf des Forschungsprojektes entwickelt werden.

3.1. Architektur von WalkSim

WalkSim wurde von Grund auf dafür entworfen, dass seine Komponenten beliebig auf verschiedene Server verteilt werden können. Die unterliegende Kommunikations-Middleware abstrahiert dabei den Ort der Komponenten, sodass es keine Rolle spielt, auf welchem Server eine bestimmte Komponente nun konkret ausgeführt wird.

Im Folgenden werden nun die wichtigen Komponenten näher erläutert (vgl. Abb. 3.1).

3.1.1. WalkManager

Der *WalkManager* ist die zentrale, steuernde Komponente des Systems. Er weiß, welche Simulationen gerade ausgeführt werden und welche Simulations-Szenarien verfügbar sind. Intern gliedert er sich in zwei weitere Komponenten:

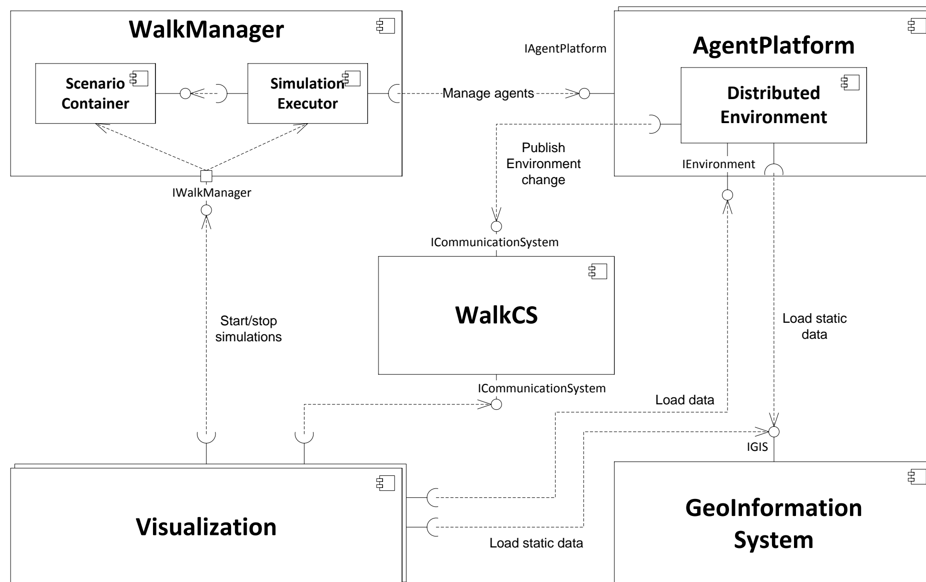


Abbildung 3.1.: Grundlegende Architektur von WalkSim

3.1.1.1. ScenarioContainer

Der ScenarioContainer verwaltet alle verfügbaren Szenarios, die WalkSim ausführen kann. Diese werden als XML-Datei auf dem Datenträger gespeichert und beim Programmstart geladen. Neben der Auswahl einer Simulationsumgebung, die den Gebäudeplan enthält, kann in diesen Szenario-Dateien der Ablauf einer Simulation detailliert festgelegt werden.

3.1.1.2. SimulationExecutor

Der *SimulationExecutor* ist für die eigentliche Ausführung einer Simulation zuständig. Zum einen gehört dazu die Initialisierung der Simulation, bei der die im Szenario definierten Agenten auf den beteiligten Servern gestartet werden, zum anderen wird hier der sogenannte *Simulation-Loop* ausgeführt.

Die Simulationsausführung in WalkSim arbeitet nicht kontinuierlich, sondern getaktet. Durch die zusätzlichen Synchronisationspunkte, die eine solche getaktete Ausführung mit sich bringt, wird zwar Performance verschenkt, weil bei jedem Teilschritt auf den langsamsten Teilnehmer

gewartet werden muss, jedoch gestaltet sich so der Abgleich unter den Servern deutlich einfacher. Dafür wird eine virtuelle Simulationszeit eingeführt, welche den aktuellen Fortschritt einer Simulation kennzeichnet.

3.1.2. Geoinformationssystem

Das Geoinformationssystem (GIS) verwaltet die statischen Daten der Simulationsumgebungen. In dieser Arbeit wird ein sehr rudimentärer Prototyp verwendet, welcher lediglich alle statischen Hindernisse mit ihren räumlichen Ausmaßen enthält.
Das GIS teilt die gespeicherten Daten fachlich in verschiedene Layer. Jeder dieser Layer stellt eine Klasse von Information dar (z.B. statisches Hindernis, Feuer, Wasser) und kann separat abgefragt werden.

Das GIS ist keine direkte Komponente von WalkSim, sondern stellt eher einen reinen Adapter zu einer externen Geodatenbank dar, welcher diese Daten für WalkSim aufbereitet und zur Verfügung stellt.

3.1.3. AgentPlatform

Auf jedem Server, der Agenten ausführen soll, läuft eine *AgentPlatform*, welche die Agenten starten, migrieren und stoppen kann.
Für jede Simulation existiert zudem ein lokaler Simulationsverwalter, welcher die konkreten Kommandos des SimulationExecutors ausführt und an entsprechende Unterkomponenten delegiert.

Jede AgentPlatform kann verschiedene Implementationen von Agenten starten. Für jeden Agententyp existiert eine *AgentFactory*, welche die konkreten Agenten erstellt. So ist es möglich, mehrere gänzlich verschiedene Typen von Agenten in einer einzigen Simulation auszuführen und miteinander zu vergleichen.

3.1.4. WalkCS

Das Kommunikationssystem *WalkCS* stellt die Middleware für die Verteilung zur Verfügung, welche dafür sorgt, dass die einzelnen Komponenten aufeinander zugreifen können, als würden sie auf dem selben Server ausgeführt werden. Dafür stellt das Kommunikationssystem zwei Arten der Kommunikation zur Verfügung:

- **RPC**: Unterstützte Komponenten können sich beim Kommunikationssystem registrieren. Danach kann auf die in der Komponente spezifizierten Methoden via RPC systemweit zugegriffen werden. Ein RPC blockiert solange, bis die entfernte Komponente eine Antwort sendet.
- **Channels**: Das Kommunikationssystem stellt mit *Channels* asynchrone Kommunikationskanäle zur Verfügung, mit denen sich beliebige Komponenten über den publish/subscribe-Mechanismus verbinden können. Die Channel-Kommunikation entkoppelt die direkte Sender->Empfänger-Verbindung, da der Sender keine Kenntnis über die potentiellen Empfänger einer veröffentlichten Nachricht hat. Channel-Nachrichten sind nicht blockierend und können keine Antwort-Nachricht erhalten.

3.2. Distributed Environment

Das *Distributed Virtual Environment* (im Folgenden DVE) stellt den verteilten Datenspeicher von WalkSim zur Verfügung.

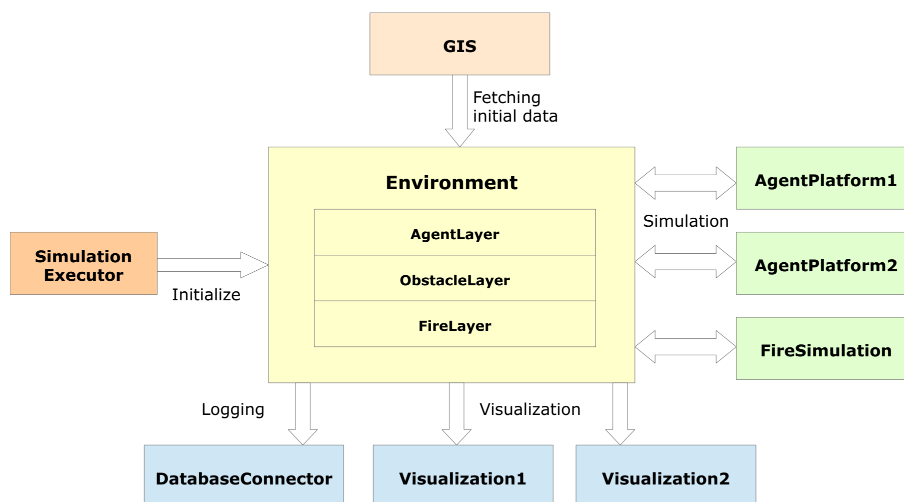


Abbildung 3.2.: Fachliche Integration des DVE in WalkSim

Abbildung 3.2 zeigt die fachliche Integration des Distributed Virtual Environments als verteilten Datenspeicher. Mit dem DVE sind mehrere Simulationskomponenten verbunden, welche sowohl lesend als auch schreibend auf dessen Daten zugreifen können. Diese Komponenten müssen nicht zwingend Agentenplattformen sein. Auch wenn solche Komponenten in dieser Arbeit nicht weiter betrachtet werden, ist es möglich, dass auch andere Simulationen für z.B.

Feuer, Rauch oder auch Verkehrssimulationen angegliedert werden.
Zusätzlich besteht die Möglichkeit, dass sich externe Komponenten wie Visualisierungen oder Database-Logger an das DVE andocken können. Diese können jedoch nur lesend auf das DVE zugreifen.

Das DVE abstrahiert die Verteilung und Replikation des Simulationszustandes für die zugreifenden Simulationskomponenten. Die Komponenten können beliebig auf diverse Hosts verteilt werden und haben trotzdem Zugriff auf den gemeinsamen Simulationszustand.

3.2.1. Sektoren

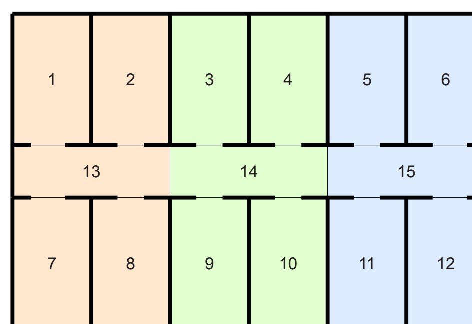


Abbildung 3.3.: Aufteilung eines DVE mit 15 Sektoren auf drei Instanzen

Das DVE wird in mehrere räumliche Sektoren aufgeteilt (siehe Abb. 3.3). Diese Sektoren müssen nicht zwingend alle die gleiche Größe besitzen und auch nicht homogen wie in einem Raster verteilt sein. So können sich die Sektoren auch an statischen Hindernissen wie Wänden und Räumen orientieren, was für die Partitionierung später Vorteile hat. Untereinander können die Sektoren über Nachbarschaftsverhältnisse miteinander verbunden werden. Damit wird angegeben, dass zwei Sektoren direkt aneinander grenzen und es für Agenten möglich ist, von einem zum anderen Sektor zu gelangen. Dies hat später ebenso Vorteile für die Partitionierung und kann ggf. auch für eine einfachere Wegfindung der Agenten genutzt werden.

Die Sektoren sind in einer Baumstruktur organisiert. Der Wurzelsektor umfasst dabei die gesamte Simulationsumgebung, die darunter liegenden Kindsektoren jeweils nur einen Teil dieser Umgebung, wobei Kindsektoren jeweils auch wieder Kindsektoren haben können und der Baum so theoretisch unendlich tief sein darf. Diese Strukturierung als Sektorbaum ermöglicht so zum einen eine logische Teilung der Umgebung in Gebäude, Stockwerke, Flure und Räume, zum anderen hat es den algorithmischen Vorteil, dass eine Suche in einem geordneten Baum deutlich effizienter ist als in einer flachen Liste.

3.2.2. Verteilung

Bei der Verteilung des DVE auf n Hosts werden nun diese Sektoren in n Partitionen aufgeteilt. Die Aufteilung übernimmt der zentrale SimulationExecutor mit Hilfe eines Partitionierungsalgorithmus (siehe Kapitel 4.2). Dieser errechnet in regelmäßigen Abständen eine neue Partitionierung des DVE und veröffentlicht diese im System, sodass alle Komponenten diese erfahren. Jeder Sektor wird einer Agentenplattform für die Zeitspanne bis zur nächsten Neupartitionierung eindeutig zugewiesen, sodass diese Plattform als einzige für diesen Sektor verantwortlich ist. Objektänderungen können dementsprechend immer nur von der Plattform vorgenommen werden, die für den Sektor, in dem sich das Objekt befindet, verantwortlich ist. Möchte eine Simulationskomponente ein Objekt einer anderen Plattform ändern, muss dies per RPC geschehen. Dadurch dass jede Plattform die vollständige Partitionierung vom SimulationExecutor gesendet bekommt, besitzt jede DVE-Instanz ein vollständiges Verzeichnis darüber, auf welcher Plattform welcher Sektor verwaltet wird. So ist es bei einem Zugriff auf fremde Sektoren nicht erforderlich, zuerst an einer zentralen Stelle nachzufragen, bei welcher DVE-Instanz ein bestimmter Sektor verwaltet wird.

Bei dieser Partitionierung ist es natürlich zweckmäßig, dass z.B. Agenten der Simulation auf der Instanz ausgeführt werden, die auch den Sektor verwaltet, in dem sich der Agent befindet, um unnötige Kommunikation zwischen den DVE-Instanzen zu vermeiden. Zwingend erforderlich ist dies jedoch nicht.

3.2.2.1. Interaktion

Die Interaktion anderer Komponenten mit dem DVE beschränkt sich auf die vier Operationen *get*, *add*, *update* und *remove*:

- *get(id)*: Es wird ein einzelnes Objekt mit Hilfe dessen ID abgefragt.
- *get(geometry)*: Es werden alle Objekte gesucht, die sich innerhalb des beschriebenen Bereiches befinden. Aktuell kann dieser Bereich ein Rechteck oder ein Kreis sein, jedoch sind theoretisch beliebige zweidimensionale Formen möglich.
- *add(object)*: Fügt dem DVE ein neues Objekt hinzu. Ist das Objekt bereits vorhanden, wird es überschrieben. Es kann jedes beliebige Objekt hinzugefügt werden, welches eine ID und eine Position besitzt. In den meisten Fällen besitzen die Objekte noch weitere Eigenschaften. Diese werden alle vom DVE gespeichert, jedoch nicht weiter behandelt. So können beliebig komplexe Objekte innerhalb des DVE gespeichert werden.
- *update(update)*: Aktualisiert ein im DVE vorhandenes Objekt mit Hilfe einer Update-Anweisung.

- *remove(id)*: Entfernt ein Objekt dauerhaft aus dem DVE. Nach dieser Operation kann das Objekt nicht mehr mit der *get*-Anweisung gefunden und keine Updates mehr auf das Objekt angewendet werden.

Das DVE unterstützt keinerlei Transaktionen. Jede dieser Operationen arbeitet atomar auf den einzelnen Objekten, jedoch ist es weder möglich, einzelne Objekte über die Dauer einer Operation hinaus zu sperren (z.B. in einer atomaren *Lese->Schreib*-Operation), noch können einmal vorgenommene Änderungen wieder rückgängig gemacht werden.

Alle Operationen, die gespeicherte Objekte verändern, können nur von der Instanz des DVE vorgenommen werden, die den Sektor verwaltet, in dem sich das Objekt befindet. Entsprechende Operationen werden, von der lokalen DVE-Instanz via RPC an die entsprechende Instanz auf dem entfernten Server weiter geleitet.

3.2.2.2. Synchronisation

Die Synchronisation der DVE-Instanzen findet in regelmäßigen Abständen statt, wobei die Größe der Abstände konfigurierbar ist. Für diesen Datenabgleich erstellt jede Instanz für ihre verwalteten Sektoren je eine Update-Nachricht der Form

$$environmentUpdate(newObjects, updates, removedObjects).$$

Diese Update-Nachricht enthält alle Aktualisierungen, die seit der letzten Synchronisation in diesem Sektor aufgetreten sind. Traten in einem Sektor keine Änderungen auf, wird eine leere Update-Nachricht erstellt.
Anschließend werden diese Nachrichten über das Kommunikationssystem als asynchrone Nachricht veröffentlicht, wobei jeder Sektor seinen eigenen Channel zugewiesen bekommt. Jede DVE-Instanz kann autonom entscheiden, welche Daten aus fremden Sektoren für die lokalen Agenten benötigt werden und nur die Channels für diese Sektoren werden auch abonniert. So empfängt jede DVE-Instanz nur Nachrichten für die Sektoren, deren Daten die Instanz auch wirklich benötigt.

3.2.2.3. Konsistenz

Die Konsistenz des in WalkSim verwendeten verteilten Datenspeichers basiert auf dem BASE-Prinzip. Auch wenn immer eine möglichst hohe Konsistenz der Daten angestrebt wird, ist es nicht möglich, dass eine Kopie eines Objektes immer auf dem aktuellen Stand ist.

Das ist darauf zurück zu führen, dass die Synchronisation eines Objektes nur in regelmäßigen Abständen durchgeführt wird und nicht direkt bei der Änderung des Objektes. Das hat den Vorteil, dass die einzelnen Objektaktualisierungen zu einer größeren Nachricht zusammengefasst werden können, welche schneller übertragen werden kann. Ein Nachteil ist jedoch, dass Daten, die in einer fremden DVE-Instanz verwaltet werden, bis zum nächsten Abgleich potentiell veraltet sind und der kopierte Zustand nicht mehr dem originalen Zustand entspricht. Dieser Nachteil hält sich allerdings in Grenzen, da der Großteil der Aktualisierungen lediglich die Position eines Objektes um wenige Zentimeter verändert. Diese Inkonsistenz hat zwar unbestritten Auswirkung auf den Wegfindungsalgorithmus eines Agenten, jedoch wird diese so klein sein, dass sie gegenüber dem immens größeren Aufwand einer sofortigen Publikation von Änderungen vernachlässigbar ist. Zudem greifen in einer Simulation potentiell nur wenige Agenten einer Plattform auf Agentendaten anderer Plattformen zu.
Die Auswirkungen dieser nur zyklischen Aktualisierung hängen teilweise vom konkreten Szenario ab und inwiefern sich große, dichte Gruppen von Agenten an Partitionsgrenzen aufhalten. Der wichtigste Faktor ist jedoch die Länge der Synchronisationsintervalle. Für diese Arbeit wurde ein Synchronisationsintervall von 40 ms Simulationszeit festgelegt, wodurch sich das DVE folglich 25 mal in der Sekunde synchronisiert. Inwiefern die Veränderung dieses Intervalls signifikante Auswirkungen auf die Simulation haben, wird in dieser Arbeit nicht weiter untersucht, da die Validierung von Simulationen nicht Teil dieser Arbeit ist.

3.3. Simulationsausführung

Die Ausführung der Simulation basiert in WalkSim auf einem getakteten Algorithmus. Der SimulationExecutor jeder Simulation agiert dabei als zentraler Taktgeber. Dieser startet den nächsten Simulationsschritt nur, wenn alle verteilten Komponenten den letzten Schritt beendet haben. Jeder Takt wird dabei in mehrere Unterschritte geteilt:

1. **Szenarioereignisse** Zu Beginn jedes Schrittes bearbeitet der SimulationExecutor alle Szenarioereignisse, die während des aktuellen Simulationsschrittes auftreten werden (z.B. Agentenerstellung).
2. **Agentenfortschritt** Der Agentenfortschritt bezeichnet die Abarbeitung aller Agentenereignisse, die bis zum Ende des aktuellen Simulationsschrittes auftreten.
3. **Synchronisation** Zu Beginn der Synchronisationsphase führt jede DVE-Instanz für ihre lokalen Objekte eine Kollisionserkennung und -lösung durch. Anschließend gleichen die Instanzen ihre Daten untereinander ab.
4. **Partitionierung** Der SimulationExecutor berechnet mit dem gewählten Partitionierungsalgorithmus eine neue Partitionierung und verteilt diese an die einzelnen Agentenplatt-

formen. Diese prüfen mit der neuen Partitionierung, ob lokale Agenten ggf. auf andere Plattformen migriert werden müssen.

3.3.1. Zeit

Wie bereits in den Grundlagen in Kapitel 2.2.2.1 erläutert, gibt es in verteilten Systemen verschiedene Möglichkeiten einer Zeitsynchronisation. Meist steht diese Zeit in Verbindung mit der sortierten Übermittlung von Ereignissen an den Agenten.

WalkSim verwendet eine logische Uhr, um die einzelnen Hosts zu steuern. Diese Uhr ist zwar abgekoppelt von der realen, externen Zeit, jedoch weist sie nicht zwingend die Konsistenzeigenschaften der Lamportzeit auf.
Die Zeitsynchronisation regelt dabei der zentrale SimulationExecutor, welcher zu Beginn jedes Simulationsschrittes einen Zeitstempel t_{end} an die einzelnen Hosts sendet. Dieser Zeitstempel markiert jeweils das Ende des aktuellen Taktes.
Während des Agentenschrittes eines Taktes leitet jeder Host nun alle Ereignisse an die Agenten weiter, für deren Zeitstempel $t < t_{end}$ gilt. Jeder Agent besitzt eine individuelle Warteschlange für eingehende Ereignisse, die nach dem logischen Zeitstempel geordnet sind. Dabei gibt es explizit keine Garantie, dass alle Ereignisse in der Warteschlange einen Zeitstempel aufweisen, der größer ist als der des zuletzt von diesem Agenten bearbeiteten Ereignisses. Agenten können also durchaus Ereignisse aus der Vergangenheit bekommen.

Während eines Simulationsschrittes werden die Zeiten nicht weiter zwischen den einzelnen Agenten synchronisiert. Durch die Taktung der Simulation können zwei Agenten trotzdem nur maximal so weit in ihrer logischen Zeit auseinander liegen, wie ein einzelner Simulationstakt lang ist — in dieser Arbeit 40 Millisekunden. Dementsprechend hält sich der Nachteil, welcher durch diese Vereinfachung der Zeitsynchronisierung entsteht, in Grenzen. Eine totale und zeitgeordnete Zustellung der Ereignisse, welche Ereignisse aus der Vergangenheit verbietet, würde erheblichen Einfluss auf die Skalierbarkeit des Systems haben, da der Synchronisierungsaufwand zwischen den Agenten um ein Vielfaches ansteigen würde.

3.4. Die acht Irrtümer der verteilten Datenverarbeitung

Bei der Entwicklung von verteilten Systemen spricht man oft von den *Fallacies of Distributed Computing*, den Irrtümern der verteilten Datenverarbeitung (vgl. [37]). Diese bezeichnen acht gängige Annahmen, die Entwickler für wahr erachten, obwohl sie falsch sind. Auch wenn diese hier aufgezählten Annahmen durchaus stimmen (bzw. nicht stimmen), kommt es immer auf den konkreten Kontext an, ob deren Missachtung auch Auswirkungen auf die Anwendung hat:

Das Netzwerk ist ausfallsicher Kein Netzwerk ist ausfallsicher, da keine Hardware eine unendliche Lebensdauer hat. Ausfallsicherheit kann nur durch Redundanzen erhöht werden. Da die Ausführung einer Simulation in WalkSim jedoch keine lebensbedrohliche Wichtigkeit erfordert, kann akzeptiert werden, dass das System bei Versagen einer einzelnen Komponente im Ganzen nicht mehr funktioniert, bis der Defekt behoben wurde. In WalkSim wird deshalb explizit nicht auf Ausfallsicherheit geachtet. Fällt eine Komponente aus, fällt das ganze System aus.

Die Latenzzeit ist gleich null In jedem Netzwerk gibt es eine Round-Trip-Time (RTT), die selbst das kleinste Paket benötigt, um von einem Rechner zum anderen und zurück gesendet zu werden. Bei einem Datenabgleich spielt diese Latenzzeit eine große Rolle. Je länger die RTT, desto mehr Zeit wird das DVE für den Datenabgleich benötigen und desto mehr Zeit verloren, wenn der SimulationExecutor Kommandos an die Agentenplattformen sendet.

Der Datendurchsatz ist unendlich Die limitierte Bandbreite eines Netzwerkes ist die Hauptmotiation für die Entwicklung einer partiellen Synchronisation, da die Übertragung großer Updates entsprechend mehr Zeit benötigt.

Das Netzwerk ist sicher Die Sicherheitsfrage stellt sich hauptsächlich in öffentlichen verteilten Systemen, die in großen Netzwerken arbeiten. Da WalkSim jedoch auch funktioniert, wenn es in einem kleinen isolierten LAN ausgeführt wird, ist der Schutz vor Hackerangriffen oder ähnlichem nicht erforderlich und wird auch nicht weiter betrachtet.

Die Netzwerktopologie wird sich nicht ändern Dieser Irrtum ist hauptsächlich an große Netzwerke adressiert, deren Topologie nach vielen Kriterien aufgebaut ist und sich zusammen mit den Kriterien ebenso ändern kann. Eine geänderte Topologie kann Auswirkungen auf Latenz, Datendurchsatz oder überhaupt die Erreichbarkeit von Hosts haben. Für WalkSim wird vorausgesetzt, dass die Topologie des Netzwerkes konstant bleibt. Es ist sogar davon auszugehen, dass WalkSim unabhängig davon ebenso funktionieren wird, solange sich alle teilnehmenden Hosts gegenseitig erreichen können.

Es gibt immer nur einen Netzwerkadministrator Je größer das verteilte System, desto mehr Personen werden benötigt, um es zu verwalten. Die Verwaltung von WalkSim kann jedoch vorerst von einer Person bewältigt werden, sodass diese Annahme irrelevant ist.

Die Kosten des Datentransports können mit null angesetzt werden Zusätzlich zur bereits erwähnten Latenz im Netzwerk treten bei der Netzwerkkommunikation noch weitere Kosten auf, die meist die betriebssysteminterne Verarbeitung oder die Serialisierung und Deserialisierung der Datenpakete betreffen. Diese Latenz wirkt sich ebenso auf die Ausführungszeit von WalkSim aus und muss in der Implementierung so gering wie möglich gehalten werden. Zusätzlich beschreibt dieser Irrtum die monetarischen Kosten der Netzwerkinfrastruktur, dieser Faktor wird für WalkSim jedoch nicht näher betrachtet.

Das Netzwerk ist homogen In vielen Netzwerken sind nicht alle Teilnehmer identisch. Sie unterscheiden sich in ihrer Hardware, dem Betriebssystem, benutzen unterschiedliche Netzwerkprotokolle zur Kommunikation. All das macht die Entwicklung eines verteilten Systems komplizierter. Für WalkSim wird jedoch in dieser Arbeit angenommen, dass das Netzwerk homogen ist und alle teilnehmenden Hosts die gleiche Konfiguration besitzen.

4. Implementierung

Für die Implementierung von WalkSim fiel die Wahl auf die Programmiersprache Java. Großer Vorteil dieser Sprache ist deren Plattformunabhängigkeit, sodass WalkSim komfortabler als universale Forschungsplattform für Fußgängersimulationen publiziert werden kann. Ausschlaggebend für die Programmierung in Java waren zudem die vorhandenen Kenntnisse der Entwickler. Eine Entwicklung von WalkSim in C/C++ hätte möglicherweise mehr Leistung versprochen, die Entwicklung hätte jedoch deutlich länger gedauert bzw. die Komponenten hätten nicht im nötigen Umfang entwickelt werden können.

Im Folgenden werden ausgewählte Komponenten und Abläufe von WalkSim detailliert erläutert.

4.1. Distributed Virtual Environment

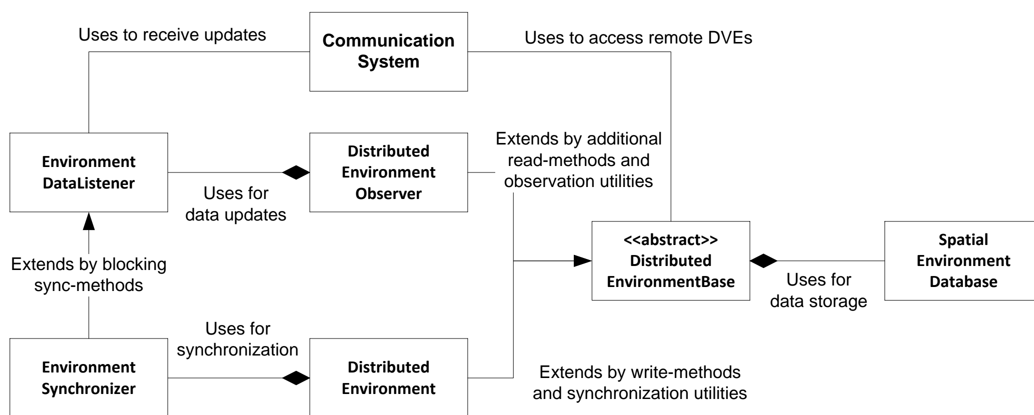


Abbildung 4.1.: Klassen des Distributed Environment

Die Basisklasse des DVE ist das *DistributedEnvironmentBase* (siehe Abb. 4.1). Diese Klasse stellt alle häufig gebrauchten Methoden zur Verfügung, um auf den verteilten Datenspeicher

einer Simulation zuzugreifen. Neben diversen Methoden, die Lesezugriff auf das DVE gewährleisten, bietet es außerdem Methoden, um Beobachter auf bestimmte Sektoren oder konkrete Objekte zu registrieren. Diese Beobachter erhalten nach dem Observer-Pattern eine Nachricht, sobald sich ein Objekt im beobachteten Sektor bzw. das beobachtete Objekt ändern, ohne dass dies wiederholt abgefragt werden muss. Diese Basisklasse instantiiert zudem die SpatialEnvironmentDatabase, welche innerhalb des DVE sämtliche lokalen Objekte verwaltet und alle Lese- und Schreibanfragen bearbeitet.

Der *DistributedEnvironmentObserver* erweitert die Basis-Klasse um Methoden, mit welchen bestimmte Sektoren des DVE manuell und individuell abonniert werden können. Dafür wird der *EnvironmentDataListener* benutzt. Dieser kann sowohl Simulationsdaten manuell von den jeweiligen DVE-Instanzen abrufen als auch automatisch auf Aktualisierungen für die abonnierten Sektoren warten und diese in die lokale Datenbank einpflegen. Der Observer stellt dabei keinerlei Methoden zur Verfügung, die das Hinzufügen, Entfernen oder Verändern von Objekten zulassen. Diese Klasse soll hauptsächlich in Visualisierungen genutzt werden, wo keine komplette Instanz des DVE sondern lediglich ein Abbild verfügbar sein soll.

Das *DistributedEnvironment* ist die eigentliche Hauptklasse des DVE. Diese Klasse wird von den einzelnen Agentenplattformen als DVE-Instanz benutzt und verwaltet sowohl Lese- als auch Schreibzugriffe auf die Datenbank. Schreibzugriffe, die fremde Objekte betreffen, werden automatisch an die jeweilige entfernte DVE-Instanz weitergeleitet. Mit Hilfe des *EnvironmentSynchronizer*, einer Erweiterung des EnvironmentDataListeners, synchronisiert die DVE-Instanz ihren Datenbestand mit den anderen Instanzen.

4.1.1. Datenmodell

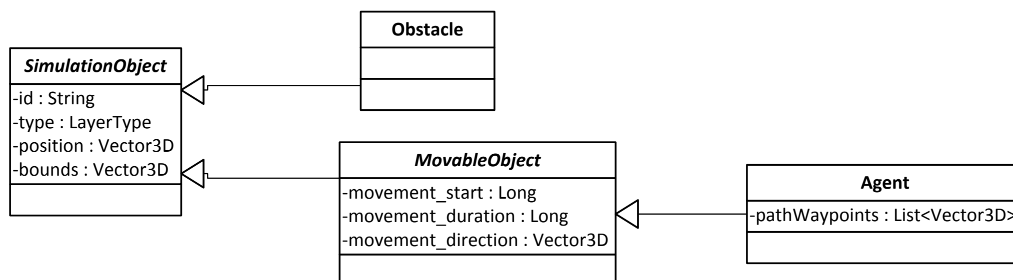


Abbildung 4.2.: Fachliches Datenmodell des DVE

WalkSim beschränkt sich für diese Arbeit auf ein recht simples Datenmodell, welches nur die nötigsten Daten modelliert, jedoch Raum für Erweiterungen lässt. Basisklasse aller Objekte

des DVE ist das abstrakte *SimulationObject*. Dieses definiert, dass jedes Objekt eine simulationsweit eindeutige ID und einen Typ besitzen muss. Mit diesem Typ kann ein Objekt einem Informationslayer zugeordnet werden (z.B. Agent, Obstacle; analog zu den Layern im GIS). Des weiteren besitzen alle Objekte eine Position im Raum und ebenso räumliche Ausmaße.

Von dieser Basisklasse werden weitere Klassen abgeleitet, die das SimulationObject um zusätzliche Eigenschaften erweitern. So besitzt das ebenfalls abstrakte *MovableObject* Eigenschaften, welche die Bewegung eines Objektes beschreiben. Dabei wird eine Startzeit t_{start} definiert, welche den Beginn der Bewegung in Simulationszeit darstellt, eine maximale Bewegungsdauer t_{max} und ein Richtungsvektor $\vec{v}$, der die Bewegungsrichtung beschreibt. Mit diesen drei Parametern und der aktuellen Position $\vec{p}$ des Objektes kann zu jeder Zeit die genaue Position $\vec{p}'$ des Objektes interpoliert werden:

$$\vec{p}' = \vec{p} + min(t_{now} - t_{start}, t_{max})\vec{v}$$

Diese Interpolation erhöht zwar den Rechenaufwand, wenn die konkrete Position gebraucht wird, jedoch ist dies deutlich genauer und erspart das pauschale zyklische Aktualisieren aller Objektpositionen in der Datenbank.

Des weiteren gibt es die Klassen *Agent* und *Obstacle*. Diese besitzen bisher keine simulationsrelevanten Eigenschaften. Das Datenmodell kann hier jedoch beliebig erweitert werden. Eine grundlegende Bedingung ist allerdings, dass die Objekte in sich abgeschlossen sein müssen, sie dürfen folglich keinerlei Referenzen nach außen besitzen.

4.1.2. Synchronisation

Während der Simulationsausführung werden alle Änderungen, die eine DVE-Instanz vornimmt, lokal in ein In-Memory-Logbuch geschrieben und dabei gleich nach den Sektoren sortiert. Wird die Synchronisation gestartet, werden alle Einträge, die sich in diesem Logbuch befinden, nach den Sektoren separiert in einer Update-Nachricht verpackt und über das Kommunikationssystem veröffentlicht. Dabei besitzt jeder Sektor eine eigene Channel-ID, über die dessen Updates versendet werden. Änderungen, die nach Beginn der Synchronisation eintreffen, werden in ein dafür neu angelegtes Logbuch geschrieben, damit keine Änderungen aus Versehen doppelt übertragen oder übergangen werden. Ist die Synchronisation abgeschlossen, wird das betreffende Logbuch verworfen, weil es nicht mehr benötigt wird.

Auch wenn die Kommunikation über die Channels des Kommunikationssystems generell asynchron abläuft, ist die Synchronisation der DVEs ein blockierender Vorgang, der erst abgeschlossen ist, wenn alle DVE-Instanzen alle Update-Nachrichten erhalten haben, auf die sie gewartet haben. Bei Netzwerkfehlern besitzt die Synchronisation einen Fallback-Modus, welcher nach einer bestimmten Wartezeit die fehlenden Updates von den entsprechenden fernen

Instanzen per RPC manuell lädt. Trifft ein automatisches Sektor-Update nach dem manuellen Laden ein, wird es ignoriert.
Totale Netzwerkausfälle kann das DVE jedoch nicht verkraften, da jedes Objekt nur eine "sichere" Instanz besitzt und alle eventuell vorhandenen Repliken auf anderen DVE-Instanzen möglicherweise veraltet sind.

4.1.2.1. Synchronisationsmenge ermitteln

Eine anspruchsvolle Aufgabe ist es, zu ermitteln, welche Daten eine Instanz des DVE benötigt, bzw. welche Sektoren abonniert werden müssen.

Dafür wurden zwei verschiedene Vorgehensweisen untersucht:
Im ersten Versuch wurde davon ausgegangen, dass die Agenten immer auf der Plattform ausgeführt werden, auf der ihr zugehöriger Sektor verwaltet wurde. Aus den geografisch begrenzten Sichtbereichen kann man schließen, dass Agenten lediglich Daten benötigen, die sich auch in ihrer räumlichen Nähe befinden. Dies sind fast ausschließlich Daten aus dem aktuellen Sektor oder maximal dem Nachbarsektor. So kann die Synchronisationsmenge ermittelt werden, indem alle Sektoren abonniert werden, die Nachbar eines eigenen Sektors sind und nicht selbst verwaltet werden.
Durch die bereits bekannte, statische Sektorstruktur kann diese Synchronisationsmenge sehr schnell ermittelt werden. Die Vorgehensweise setzt jedoch voraus, dass die Agenten einen sehr begrenzten Sichtbereich besitzen, deren Ausmaße kleiner sind, als die Größe der Sektoren. Ist der Sichtbereich größer, werden Objekte, welche zwei Sektoren weit entfernt sind, nicht synchronisiert und dementsprechend fälschlicherweise auch nicht wahrgenommen, obwohl sie im Sichtbereich des Agenten liegen. Abhängig von der Agentenmodellierung und der Größe von Sektoren und Sichtweiten kann das gravierende Auswirkungen auf den Verlauf der Simulation haben. Anders herum kann es bei großen Sektoren passieren, dass Sektoren abonniert werden, deren Daten gar nicht benötigt werden, weil kein Sichtbereich eines Agenten den Sektor schneidet.

Die zweite Methode setzt deshalb direkt bei den Abfragen des DVE an. Für jede an das DVE gestellte *get(geometry)*-Abfrage wird ermittelt, welche Sektoren diese Abfrage betrifft. Sektoren, welche weder selbst verwaltet werden noch bereits abonniert sind, werden daraufhin abonniert. Erfolgt für eine bestimmte Zeit keine Anfrage an einen abonnierten Fremdsektor, wird das Abonnement wieder entfernt. Der Vorteil dieser Methode besteht darin, dass sie unabhängig von der spezifischen Größe der Sichtbereiche der Agenten und der Größe der Sektoren ist und so auch nicht von der Modellierung der Simulation selbst abhängt. Zudem werden Sektoren nur abonniert, wenn deren Inhalte auch wirklich gebraucht werden. Es ist allerdings zu beachten, dass durch diesen weiteren Sektor-Check bei jeder *get*-Abfrage diese mehr Rechenzeit in Anspruch nimmt.

```
1  public interface SimulationObject {
2
3      String getId(); // eindeutige ID des Objektes
4      Vector3D getPosition(); // Position des Objektes
5      Vector3D getBounds(); // Ausmasse des Objektes in alle drei Dimensionen
6      LayerType getType(); // Typ des Objektes
7  }
```

Abbildung 4.3.: Basisobjekt des DVE

Für die in dieser Arbeit durchgeführten Tests wurde stets die zweite Methode gewählt, weil letztlich eine korrekte Synchronisation und damit ein ordnungsgemäßer Verlauf der Simulation wichtiger ist als ein paar gesparte Millisekunden.

4.1.3. Spatiale Hashdatenbank (In-Memory)

Innerhalb des DVE werden die Objekte in einem Key/Value-Store *SpatialEnvironmentDatabase* (siehe Abb. 4.1) gespeichert. Dieser interne Key/Value-Store ist unabhängig von jeglicher übergeordneter Verteilung und dient nur zur reinen Speicherung der lokalen Objekte und dem effizienten Zugriff auf diese.
Alle Objekte, die in der Datenbank gespeichert werden sollen, müssen das Interface des *SimulationObject* implementieren (siehe Abb. 4.3). Dieses definiert neben einer ID auch eine Position und mit der Boundingbox auch die Ausmaße des Objektes. Diese Boundingbox ist als AABB[1] definiert. Jedes Objekt besitzt zudem einen Typ, welcher es einem bestimmten Informationslayer zuordnet. Ein solcher Informationslayer trennt die verschiedenen Datentypen fachlich. Aktuell gibt es in WalkSim neben Agenten zwar lediglich *Obstacle* als Klassifizierung, jedoch können im Verlauf des Projektes noch viele weitere Typen hinzugefügt werden, um eine zusätzliche Indexdimension hinzuzufügen.

Die Datenbank besitzt zwei verschiedene Indizes.
Ein Index stellt den klassischen Key-Index aus den Key/Value-Stores dar. Mit diesem Key kann jedes Objekt eindeutig identifiziert werden. Dieser Index ist intern als Hashtabelle implementiert, sodass die Abfrage eines Objektes über seine ID, einen Aufwand von $O(1)$ besitzt.

Der zweite Index ist der spatiale Index. Dieser sorgt dafür, dass Objektmengen über geometrische Formen gesucht werden können, ohne dass über alle in der Datenbank gespeicherten Objekte iteriert werden muss. Diese Technik nennt sich spatiales Hashing. Dafür wird die Umgebung in ein zweidimensionales Gitternetz geteilt, sodass viele quadratische, gleich große Kacheln entstehen. Diese Kacheln stellen die einzelnen Einträge der Hashtabelle dar (*Hashbuckets*), in die die Objekte eingetragen werden.

[1] AABB = Axis-Aligned Bounding-Box: Quader, dessen Kanten parallel zu den Koordinatenachsen verlaufen

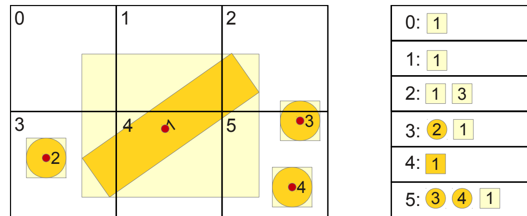


Abbildung 4.4.: Spatiales Hashing in 2D Umgebung

Daraus ergibt sich eine wie in Abb. 4.1.3 dargestellte Struktur, in der die Umgebung mit vier Objekten in sechs Zellen geteilt wurde. Die Objekte wurden zum einen anhand ihrer Position eindeutig in ein Hashbucket einsortiert. Parallel dazu wurden die Objekte noch anhand ihrer Ausmaße in mehrere Hashbuckets einsortiert.
Für die Ermittlung der korrekten spatialen Zelle und damit auch des Hashbuckets, in dem die Objekte abgelegt werden, ist eine Hashfunktion nötig. Hierfür werden zunächst mittels einer einfachen Rundungsfunktion und einer gegebenen Zellengröße $gridSize$ die Koordinaten der spatialen Zelle errechnet, in die das Objekt eingefügt werden soll.

$$x = round(x_{object}/gridSize)$$
$$y = round(y_{object}/gridSize)$$
$$z = round(z_{object}/gridSize)$$

Diese Zellenkoordinaten x, y und z werden daraufhin in eine Hashfunktion eingesetzt, welche einen möglichst chaotischen Wert zurück gibt, mit welcher das Hashbucket in der Hashtabelle adressiert werden kann.

$$hash(x, y, z) = (xp1 \oplus yp2 \oplus zp3) \bmod n$$

Die Variablen $p1$, $p2$ und $p3$ sollten möglichst hohe Primzahlen sein, in diesem Fall die Zahlen 73856093, 19349663 und 83492791 (vgl. [43]), n ist die Größe der Hashtabelle.

Ändert sich die Position eines Objektes, wird seine Position im spatialen Index neu errechnet und das Objekt entsprechend umsortiert.

Wird nun der spatiale Index mit einer geometrischen Form abgefragt, werden zuerst über die Hashfunktion alle Hashbuckets ermittelt, mit denen sich die Form schneidet. Daraufhin wird die AABB aller Objekte innerhalb der geschnittenen Hashbuckets separat mit der Form geschnitten und ggf. zur Menge der gefundenen Objekte hinzugefügt. Diese Indexsuche hat den Aufwand $O(n * m)$, wobei n die Anzahl der zu durchsuchenden Hashbuckets ist und m die durchschnittliche Anzahl der Objekte in einem Hashbucket. Beide Werten hängen von der

räumlichen Größe der Hashbuckets ab, die Anzahl der Buckets zusätzlich von der Größe der geometrischen Form. Es besteht jedoch keine direkte Abhängigkeit zur Gesamtmenge der gespeicherten Objekte.
Zusätzlich zur geometrischen Form besitzt der spatiale Index als zweite Dimension den Objekttyp als vorgelagerten Index, welcher die zu durchsuchende Menge der Objekte bereits vor der eigentlichen spatialen Suche reduziert.

Anmerkung: WalkSim wurde zu Beginn dafür entworfen, eine dreidimensionale Simulationsumgebung zu verwalten. Im Laufe der Implementierung stellte sich die Entwicklung der spatialen Datenbank für eine 3D-Umwelt als sehr aufwändig heraus. Aus diesem Grund wurde die spatiale Datenbank auf zwei Dimensionen (x und z) reduziert. Da die in dieser Arbeit durchgeführten Tests ausschließlich auf einer Ebene stattfinden und keine Treppen, schiefen Ebenen oder Stockwerke enthalten, hat dies keine weiteren Auswirkungen auf die Simulationsergebnisse.

4.1.3.1. Nebenläufigkeit

Diese spatiale Datenbank ist auf massive, nebenläufige Zugriffe ausgelegt. Jedes Hashbucket besitzt sein eigenes Read/Write-Lock, sodass es separat gesperrt werden kann. Die Sperren werden dabei immer so kurz wie möglich gehalten und nur aufrecht erhalten, solange auch tatsächlich über das Hashbucket iteriert wird und verhindert auch Änderungen an den Objekten innerhalb des Hashbuckets.
Lesezugriffe finden grundsätzlich nebenläufig statt, nur Schreibzugriffe benötigen eine exklusive Sperre. Wird ein Objekt geändert, so wird es zuvor exklusiv gesperrt.

Ein bisher ungelöstes Problem besteht in der Rückgabe gesuchter Objekte. In der aktuellen Implementierung werden stets Referenzen auf die gespeicherten Objekte zurück gegeben. Das hat den Vorteil, dass Suchvorgänge so sehr schnell sind. Der Nachteil ist jedoch, dass Änderungen, die die Datenbank vornimmt, auch auf die Instanz reflektiert werden, die die anfragende Komponente erhalten hat. Das kann durchaus zu gravierenden Problemen in der KI der Agenten führen.
Die einfachste Lösung wäre, das Objekt bei jedem Aufruf zu kopieren und nur die Kopie als Ergebnis zurückzugeben, was quasi einer Wertübergabe entspricht. Das hätte jedoch zwei entscheidende Nachteile: Zum einen würde das Kopieren jedes einzelnen Objektes in der Ergebnismenge einer Abfrage eine erhebliche Verzögerung bedeuten, weil abhängig vom Sichtbereich der Agenten pro Abfrage mehrere hundert Objekte kopiert und neu erstellt werden müssen.
Das zweite Problem besteht darin, dass Agenten mit einem internen Speicher das Ergebnis

der Abfrage lokal in ihren Speicher schreiben. So werden bei mehreren Tausend Agenten pro Server schnell etliche Gigabyte an Hauptspeicher.

4.1.3.2. Parameterjustierung

Die Performance dieser spatialen Datenbank hängt von mehreren Parametern ab:

Größe der Hashtabelle Je mehr Hashbuckets es gibt, desto seltener werden zwei weit entfernte Objekte durch Hashkollisionen im selben Hashbucket gespeichert. Da ein Schreibzugriff auf ein Objekt einen Write-Lock auf das aktuelle Bucket des Objektes erfordert, steigt auch die Skalierbarkeit an. Im Gegenzug bedeuten mehr Hashbuckets jedoch auch mehr Speicherverwaltung.

Größe der Hashbuckets Je kleiner die Hashbuckets sind, desto weniger Objekte befinden sich in einem Bucket, was lokal begrenzte Abfragen beschleunigt. Bei kleineren Buckets müssen agile Objekte jedoch häufiger umsortiert werden, wodurch mehr Sperren und höherer Verwaltungsaufwand erforderlich ist.

Welche Parametrisierung optimale Performance liefert, hängt ganz von der Simulation ab, wie viele Objekte die Simulation enthält, die Dichte der Objekte und auch deren Größe.
Aktuell verwendet die Hashdatenbank eine Tabellengröße von maximal 65.537 Buckets. Ein einzelnes leeres Hashbucket belegt dabei 754 Byte RAM, sodass eine vollständige Hashtabelle ohne Objekte einen Speicheroverhead von ca. 50MB verursacht. Die Größe der Hashbuckets beträgt dabei 10,0 m.

4.1.4. Persistenz

Für die Validierung und Analyse von Simulationen ist es nötig, dass der Verlauf einer Simulation persistent abgespeichert wird, damit dieser außerhalb von WalkSim mit weiteren Werkzeugen verarbeitet werden kann.

Ein Mechanismus um eine Simulation über die Laufzeit des Programms hinaus persistent zu speichern, ist nicht Teil dieser Arbeit. Entsprechend wird hier lediglich eine rudimentäre Implementierung vorgeschlagen. Die hier in WalkSim implementierte persistente Datenspeicherung nutzt eine SQL-Datenbank, um alle Updates, die während eines Synchronisations-Zyklus auftreten, zu speichern. Dabei werden keine vollständigen Objekte gespeichert, sondern nur die Änderungen an sich, sodass ein Simulationszustand zum Zeitpunkt t nur dadurch wiederhergestellt werden kann, indem man alle Updates anwendet, die bis zu diesem Zeitpunkt gespeichert wurden. Das erhöht zwar den Aufwand, wenn man willkürlich Zeitpunkte in der Simulation ansteuern möchte, verringert jedoch das zu speichernde Datenvolumen um ein

Vielfaches, was vor allem bei großen Simulationen mit mehreren Hunderttausend Agenten viel Speicherplatz und Performance spart.

Das erweiterbare Datenmodell von WalkSim erfordert bei jeder Änderung am Datenmodell auch eine Änderung der SQL-Tabellen, da diese das WalkSim-Datenmodell direkt widerspiegeln.

Aus Performancegründen wird als SQL-Datenbank die Implementierung *SQLite* [14] benutzt. Diese Datenbank ist sehr leichtgewichtig und speichert alle Daten einer Datenbank in einer einzigen Datei auf dem Datenträger und benötigt keine extra Datenbankanwendung, sondern kann direkt aus der JVM von WalkSim benutzt werden. Die Schnittstellen sind dafür entworfen, dass auch jede andere SQL-Datenbank dafür benutzt werden könnte. Es ist jedoch damit zu rechnen, dass jede externe Datenbankanwendung wie PostgreSQL oder MySQL die Performance erheblich reduziert.

Da SQLite die Daten in einer lokalen Datei speichert, enthalten die lokalen Datenbank-Dateien der einzelnen Plattformen lediglich die Updates der jeweiligen Plattform. Wird eine Simulation verteilt ausgeführt, so müssen diese einzelnen Datenbanken nach der Simulation manuell zusammen geführt werden.

Da die Persistenz-Schnittstelle selbst nicht Teil dieser Arbeit ist, wird im Weiteren nicht weiter auf die Funktionen und Leistung dieser eingegangen. Es ist jedoch anzumerken, dass die aktuelle Implementierung lediglich für einen Prototypen geeignet ist und für den täglichen Bedarf erweitert oder ausgetauscht werden sollte.

4.2. Partitionierung

Für eine optimale Partitionierung der Agenten bzw. der Sektoren einer Simulation werden mehrere der in Kapitel 2.3 beschriebenen Methoden implementiert und in verschiedenen Szenarien miteinander verglichen. Konkret werden die Algorithmen *KMeans*, *QHull*, *Bisection* und der Algorithmus von Peschlow implementiert.

Der Algorithmus von Takahashi ist für die Implementierung für WalkSim ungeeignet, da dessen Kernkonzept — der reine Austausch von Agenten um so einen Lastausgleich beizubehalten — nicht mit der Sektorpartitionierung vereinbar ist. Jeder Sektor kann theoretisch beliebig viele Agenten enthalten und damit auch beliebig viel Last erzeugen. Ein Sektoraustausch kann so zu einem Lastungleichgewicht führen. Dem kann zwar durch zusätzliche Bedingungen und Schranken entgegen gewirkt werden, jedoch würde das den Algorithmus zu sehr verändern, als dass dies hier in Betracht gezogen werden kann.

Bei Fußgängersimulationen entsprechender Größe verändert sich das Gesamtbild der Simulation nur sehr langsam, da sich die Agenten mit verhältnismäßig geringer Geschwindigkeit

bewegen. Deshalb ist es nicht nötig, die Partitionierung in jedem Schritt durchzuführen. WalkSim führt deshalb eine Partitionierung nur alle zehn Simulationstakte aus.

4.2.1. Partitionierungsdaten

Die Partitionierung einer Simulation wird zentral als einzelner Simulationsschritt im SimulationExecutor durchgeführt. Dafür benötigt der SimulationExecutor bzw. der konkrete Algorithmus diverse Daten aus der laufenden Simulation, um so Kommunikations- und Lastkosten zu bewerten.

Nun würde es dem Sinn der partiellen Synchronisation widersprechen, wenn der SimulationExecutor ein vollständiges Abbild des DVE hätte, um die Partitionierung möglichst genau optimieren zu können. Darum wird das für den Algorithmus benutzte Simulationsabbild auf ein Minimum reduziert. Der Algorithmus erhält für seine Bearbeitung Zugriff auf folgende Daten:

- Anzahl der einzelnen DVE-Updates, die in einem Sektor erzeugt wurden
- Anzahl der Agenten in einem Sektor
- für jeden Sektor die Menge der Nachbarsektoren, die Agenten innerhalb des Sektors sehen können
- Zeit, die benötigt wurde, um die Agenten-Ereignisse, die für Agenten innerhalb eines Sektors auftraten, zu verarbeiten

Sowohl bei der Anzahl der Updates als auch bei der Rechenzeit wird der Durchschnitt über die letzten zehn Simulationsschritte verwendet, da die Agenten oft nicht in jedem Simulationsschritt Last erzeugen und die Einzeldaten ansonsten sehr stark fluktuieren würden.

4.2.1.1. Zeitmessung

Für eine gute Lastverteilung muss zuerst ermittelt werden, wie viel Last ein Agent bzw. ein Sektor konkret erzeugt. Die Messung dieser Last stellt sich jedoch als durchaus kompliziertes Vorhaben heraus. Hierfür gibt es mehrere verschiedene Herangehensweisen:

- Es könnte für jedes Ereignis, welches ein Agent bearbeitet, die CPU-Zeit gemessen werden, die der ausführende Thread benötigt, um das Ereignis zu bearbeiten. Die Java-API stellt zwar die Möglichkeit zur Verfügung, Thread-CPU-Zeiten zu messen, jedoch nur auf Basis der Genauigkeit von Millisekunden. Die Bearbeitung eines Ereignisses benötigt jedoch meist deutlich weniger als eine Millisekunde, sodass diese Messung ungenaue Ergebnisse liefern würde.

- Anstatt die CPU-Zeit eines Threads zu messen, könnte mit Hilfe von *System.nanotime()* die genauere Java-Zeitmessung genutzt werden, um die Bearbeitungszeit zu messen. Bei dieser Methode wird jedoch lediglich die verstrichene reale Zeit gemessen, nicht die, die der Thread auch tatsächlich auf der CPU gerechnet hat. WalkSim verwendet für die Abarbeitung der Agentenereignisse viele Threads und kann nicht beeinflussen, zu welchem Zeitpunkt welcher Thread tatsächlich ausgeführt wird und welche in der Warteschlange bleiben müssen. So kann es sein, dass ein Thread ein Ereignis in einem Scheduler-Zyklus fertig verarbeitet hat, es kann aber auch sein, dass ein Thread in der Bearbeitung mehrfach durch Scheduling-Ereignisse des Betriebssystems unterbrochen wird und so die gemessene Zeit um ein Vielfaches höher liegt als die eigentlich gebrauchte Rechenzeit.
- Eine weitere Möglichkeit wäre, einfach die verbrauchte Gesamtzeit für die Verarbeitung der Agentenereignisse auf der Plattform zu messen, durch die Gesamtanzahl der Ereignisse zu teilen und entsprechend der Ereignisse pro Sektor die vermutete Last zu interpolieren. Das setzt jedoch die Vereinfachung voraus, dass alle Agentenereignisse zumindest näherungsweise den gleichen Aufwand benötigen. Da jedoch die Implementierung der Agenten nicht als bekannt angenommen werden kann, ist auch nicht bekannt, wie lange welcher Agent zur Bearbeitung bestimmter Ereignisse benötigt. Aus diesem Grund birgt diese Art der Messung eine zu große Gefahr der Ungenauigkeit.

In WalkSim stellten sich alle drei Methoden als sehr unzuverlässig heraus und lieferten keine Werte, die sinnvoll im Partitionierungsalgorithmus weiterverarbeitet werden konnten. Aus diesem Grund wird für die Lastmessung in dieser Arbeit vorausgesetzt, dass alle Agenten die gleiche Modellierung besitzen und dementsprechend eine ähnliche Last erzeugen. Das macht die eigentliche Messung von Last unnötig.
Sollte WalkSim mit Agenten ausgeführt werden, deren Last gegenüber anderen Agenten der Simulation stark variiert, ist diese Methode ebenso nicht mehr praktikabel und es müssen andere Wege gesucht werden.

4.2.1.2. Datenbeschaffung

Die Daten zu den einzelnen Sektoren werden auf den einzelnen Agentenplattformen gesammelt und aufbereitet. Um für die Übermittlung der Daten Zeit zu sparen, wird ein Trick angewandt. Wenn der SimulationExecutor den Plattformen den Befehl gibt, sich zu synchronisieren, wartet dieser auf eine Erfolgsmeldung von jeder einzelnen Plattform. Die für die Partitionierung von den Plattformen benötigten Daten werden von der Plattform in diese Antwortnachricht verpackt und vom SimulationExecutor an den Partitionierungsalgorithmus weitergeleitet.

4.2.2. Anpassung der Referenzimplementierungen für WalkSim

Bis auf den Bisection-Algorithmus verteilen alle vorgestellten Algorithmen keine räumlichen Bereiche sondern einzelne Agenten. Da WalkSim jedoch die Partitionierung aus Performancegründen auf den Sektoren aufbaut, müssen diese Algorithmen entsprechend auf die Bedürfnisse von WalkSim angepasst werden. Dabei wird versucht, die Algorithmen so wenig wie möglich zu verändern, um so die signifikanten Eigenschaften des Algorithmus beizubehalten.

KMeans

Der KMeans-Algorithmus ist ursprünglich kein Algorithmus für die Partitionierung von Fußgängersimulationen, und muss dementsprechend ebenso angepasst werden.
Der Vorteil an KMeans besteht darin, dass er mit jeglichen Objekten arbeiten kann, die eine Position besitzen, sodass die Sektorstruktur keine besonderen Probleme macht, da auch Sektoren eine Position besitzen. Zu beachten ist hierbei, dass nicht die Ausmaße des Sektors beachtet werden sondern lediglich sein Mittelpunkt. Bei Sektoren, die deutlich länger als breit sind, können so Ungenauigkeiten entstehen.

In der Folge wird pro Partitionierungsschritt eine feste Anzahl an Iterationen durchgeführt. In diesen Folgeiterationen werden die Sektoren mit Hilfe einer Kostenfunktion ggf. einer neuen Plattform zugeordnet. Ein Sektor wird nur von seiner aktuellen Partition zu einer neuen Partition verschoben, wenn sich die Kosten der aktuellen Partition stärker verringern als sich die der neuen Partition erhöhen, und sich die Gesamtkosten so senken. Dabei wird der Sektor zu der Partition migriert, bei der die Kostenersparnis am höchsten ist. Die Kostenfunktion errechnet sich aus der Summe der euklidschen Distanzen aller Sektoren zum Partitionszentrum multipliziert mit der jeweiligen Anzahl an Agenten. Durch diese Kostenfunktion werden Sektoren bevorzugt, die nahe am Partitionszentrum liegen.
Bei diesem Schritt werden zunächst ausschließlich Sektoren behandelt, die mindestens einen Agenten enthalten. In einem folgenden Schritt werden alle leeren Sektoren jeweils der Partition zugewiesen, zu deren Zentrum sie den geringsten Abstand haben. Ein Nachteil dieser Kostenfunktion ist, dass eine Partition mit vielen dicht zusammenstehenden Agenten genauso viele Kosten verursacht, wie eine Partition mit wenig Agenten, die aber weitläufig verteilt sind.

QHull

Der Partitionierungsalgorithmus QHull wird in WalkSim in einer vereinfachten Form angewendet und stellt eine Erweiterung des KMeans dar. Die Hauptfunktion von QHull, die konvexe

Hülle um die Agenten, welche eine geschlossene Region bilden soll, über welche die Partitionierung geregelt wird, ist in WalkSim nicht nötig, da die Sektoren bereits Regionen repräsentieren.

Für die initiale Partitionierung wird die zuvor beschriebene Implementierung des KMeans-Algorithmus genutzt.
Der ursprüngliche Algorithmus iteriert bei der folgenden dynamischen Partitionierung solange über alle Server, solange auch nur einer von der Durchschnittslast abweicht (Abweichung != null). Dadurch, dass die Sektoren unterschiedlich viele Agenten enthalten können, kann es für WalkSim unmöglich werden, eine Abweichung von null überhaupt zu erreichen, sondern dass die Abweichung während des Algorithmus stets um den Nullpunkt pendelt, ohne ihn zu erreichen. Deshalb wurde die Abbruchbedingung entsprechend verändert, sodass nur noch ein Grenzwert unterschritten werden muss.
Als Lastmetrik wird wie im Algorithmus beschrieben die Anzahl der Agenten benutzt.

Partitionierung nach Peschlow

Die Partitionierung nach Peschlow kann weitgehend so implementiert werden, wie sie zuvor in Kapitel 2.3.3.4 beschrieben wurde. Die Lastdaten werden auf Sektoren aggregiert und nur Sektoren werden migriert, nicht einzelne Agenten. Die Bestimmung der CPU-Last eines einzelnen Agenten erfolgt allerdings nicht auf die im ursprünglichen Algorithmus beschriebene Weise, da sich diese als zu ungenau herausstellte (vgl. Kapitel 4.2.1.1). Für den Algorithmus wird angenommen, dass alle Agenten die gleiche Last erzeugen.

Bisection

Der Bisection-Partitioning-Algorithmus ist als Graph-Partitionierer der einzige Algorithmus, der tatsächlich Nutzen aus den vordefinierten Nachbarschaftsbeziehungen zieht. Das Sektormodell von WalkSim ist dabei eins zu eins auf den Bisection-Algorithmus anwendbar, da auch dort die Umwelt in Kacheln mit mehreren Agenten pro Kachel geteilt wurde. Knotengewicht bleibt die Anzahl der Agenten, Kantengewicht die Anzahl der Update-Nachrichten, die zwischen den Sektoren ausgetauscht werden. Dabei kann jede Kante in beide Richtungen verschiedene Gewichtungen haben, abhängig davon, ob die Agenten des einen Sektors den anderen sehen.

Die Implementierung des Algorithmus gestaltete sich jedoch schwierig, da entscheidende Punkte des Algorithmus nicht detailliert genug beschrieben waren. So mussten diese Teile in Eigenregie implementiert werden.
Zum einen war nicht beschrieben, wie in einer Partition die beiden Sektoren mit der größten Entfernung gefunden werden. Dies wurde mit Hilfe des Distanz-Vektor-Verfahrens errechnet und die Nachbarschaftsbeziehungen der Sektoren als Kanten benutzt. Dieses hat jedoch einen

Aufwand von $O(n^2 m)$ (n = Anzahl der Sektoren, m Anzahl Schritte auf dem längsten kürzesten Pfad). Zum anderen ist unbeschrieben, wie daraufhin die übrigen Sektoren den zwei Partitionen zugeordnet werden. Um dies zu lösen, wurden zwei sortierte Listen erstellt. Jede Liste enthält alle noch nicht zugewiesenen Sektoren. Sortiert werden beide Listen jeweils nach der Entfernung zu jeweils einem der beiden Startsektoren der Partitionen, wofür die bereits berechnete Distanz-Vektor-Tabelle benutzt wird. Danach wird der Reihe nach der bei jedem Schritt jeweils größten Partition der dichteste Sektor aus der jeweiligen Liste zugewiesen. Dabei wird zuvor geprüft, ob die Partition nach der Zuweisung des Sektors weiterhin eine zusammenhängende Partition ist, also einer der direkten Nachbarn des neuen Sektors bereits Mitglied der Partition ist.

In den fortlaufenden Iterationen während der Simulation werden die Sektoren, wie schon im Algorithmus beschrieben, anhand der Kostenfunktion zwischen den Partitionen verschoben. Bei der Entfernung eines Sektors aus einer Partition muss die Partition jedoch zwingend weiterhin eine zusammenhängende Partition bleiben. Dafür muss von einem zufällig aus der Partition ausgewählten Sektor jeder andere Sektor der Partition über die Nachbarschaftsbeziehungen erreichbar sein.

Peschlow + Bisection

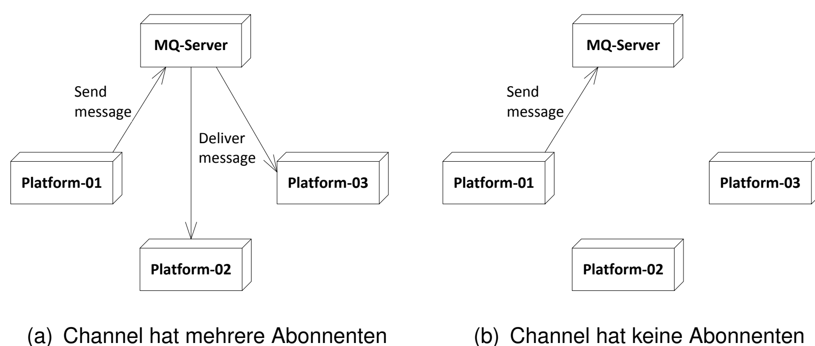
Während der Testläufe und Ergebnisanalyse wurde ein weiterer Algorithmus implementiert, welcher die dort aufgefallenen Vorteile des Peschlow- und des Bisection-Algorithmus kombinieren soll.

So wurde die Optimierungsfunktion des Bisection-Algorithmus durch eine kombinierte Variante der beiden Teilschritte des Peschlow-Algorithmus ersetzt. Beide Teilalgorithmen werden hier direkt nacheinander ausgeführt. Zudem wurde ein Schwellwert eingeführt, welcher verhindert, dass ein Sektor zu einer anderen Partition migriert wird, wenn dieser nur marginal mehr mit der anderen Partition kommuniziert.

4.3. Kommunikationssystem

Bei der Entwicklung von verteilten Systemen ist es üblich, die Netzwerkkommunikation vom Rest des Systems zu abstrahieren. So ist zum einen die eigentliche Anwendung von jeglichen Netzwerkaufgaben entkoppelt, zum anderen ist so eine Ortstransparenz für die Komponenten von WalkSim hergestellt, bei der die einzelnen Komponenten nicht wissen müssen, auf welchem Host sich eine Komponente konkret befindet. Komponenten kommunizieren miteinander, als wären sie auf dem gleichen Host.

Das Kommunikationssystem *WalkCS* muss sowohl blockierende Methodenaufrufe auf fremde Komponenten (RPC) unterstützen als auch die asynchrone Veröffentlichung von Nachrichten unter einer bestimmten Channel-ID.
In der Java-Welt gibt es für solche Kommunikationssysteme viele Implementierungen. Die gängigsten Message-Bus-Systeme sind *ActiveMQ* [1] und *RabbitMQ* [4]. Die klassischen Message-Bus-Systeme haben jedoch einen gravierenden architektonischen Nachteil für den Einsatz in WalkSim.
Praktisch alle gängigen Message-Bus-Systeme arbeiten mit einem zentralen MQ-Server[2].



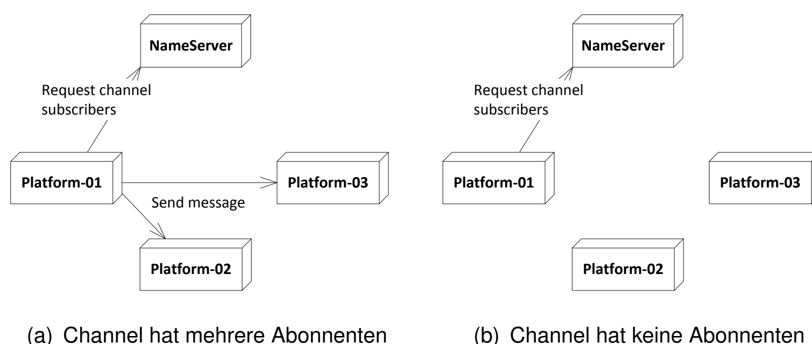
(a) Channel hat mehrere Abonnenten (b) Channel hat keine Abonnenten

Abbildung 4.5.: Weg einer Nachricht in einem zentralisierten Messagebus

Über diesen Server wird sämtliche Kommunikation geleitet, die über den Messagebus gesendet werden soll. Alle Komponenten, die Nachrichten empfangen wollen, müssen sich zuvor bei diesem MQ-Server anmelden. Das hat den Vorteil, dass zentral geregelt wird, wer welche Nachrichten bekommt und die Zustellung immer konsistent ist.
Der Nachteil dieser Struktur wird in Abb. 4.5 verdeutlicht. Die Abbildung 4.5(a) zeigt den typischen Aufbau einer verteilten Struktur mit einem Messagebus. Wird eine Nachricht veröffentlicht, wird sie zuerst an den MQ-Server gesendet. Dieser leitet die Nachricht daraufhin an alle Hosts weiter, die den Channel dieser Nachricht abonniert haben. Bei zwei Empfängern muss die Nachricht also dreimal gesendet werden, damit sie alle Empfänger erreicht. Abb. 4.5(b) stellt den Extremfall dar, in welchem die Nachricht gar keinen Empfänger besitzt, weil kein Host den Channel abonniert hat. Wird nun eine Nachricht veröffentlicht, muss sie aber trotzdem zum MQ-Server geschickt werden, weil erst dort geprüft wird, wer überhaupt Empfänger der Nachricht ist.

Die Abbildung 4.6 schlägt eine alternative Struktur vor. Hier existiert kein zentraler MQ-Server, sondern lediglich ein NameServer. Dieser verwaltet die gleichen Listen über Abonnenten wie der MQ-Server, nur dass hier diese Listen den jeweiligen Teilnehmern des Messagebusses zur Verfügung gestellt werden. Der Sender einer Nachricht stellt anschließend selbstständig eine

[2]MQ=Message-Queue



(a) Channel hat mehrere Abonnenten (b) Channel hat keine Abonnenten

Abbildung 4.6.: Weg einer Nachricht in einem verteilten Messagebus

Verbindung mit allen Empfängern her und sendet die Nachricht direkt dort hin. Dies spart in Abb. 4.6(a) den Hop zum MQ-Server, sodass eine Nachricht an zwei Empfänger tatsächlich auch nur zweimal übertragen wird. Im Fall 4.6(b) wird die Nachricht überhaupt nicht übertragen, weil es keinen Empfänger gibt.

Die verteilte Architektur mit dem NameService hat vor allem bei der partiellen Synchronisierung immense Vorteile. Die partielle Synchronisierung sorgt dafür, dass viele Update-Nachrichten des DVE oft gar keinen Empfänger haben, weil der betreffende Sektor außerhalb der Sichtbereiche anderer DVE-Instanzen liegt. Der Vorteil der partiellen Synchronisierung würde dementsprechend vernichtet werden, wenn die Nachricht trotzdem erst serialisiert, versendet und deserialisiert werden muss, um in einem zentralen MQ-Server festzustellen, dass die Nachricht keinen Empfänger hat.

4.3.1. Implementierung

Die Implementierung von WalkCS besteht aus zwei Teilen:
Die Klassen *RPCManager* und *ChannelManager* verwalten die lokal registrierten Komponenten. Komponenten, die das *WalkMainComponent*-Interface implementieren, können sich als RPC-fähiges Objekt registrieren lassen und so RPCs empfangen. Die Nachrichten-Channels können von allen Komponenten benutzt werden, sowohl zum Veröffentlichen als auch zum Abonnieren.

Der*NetworkLayer* verwaltet die eigentlichen Netzwerkverbindungen und versendet und empfängt alle Nachrichten. Er unterhält die zwei Klassen *MessageSender* und *CSTcpServer*. Möchte ein Host eine Nachricht an einen anderen Host senden, baut der MessageSender des einen Hosts eine TCP-Verbindung zum Listening-Socket des CSTcpServers des anderen Hosts auf. Der *ConnectionProcessor* des empfangenden Hosts verarbeitet die empfangene

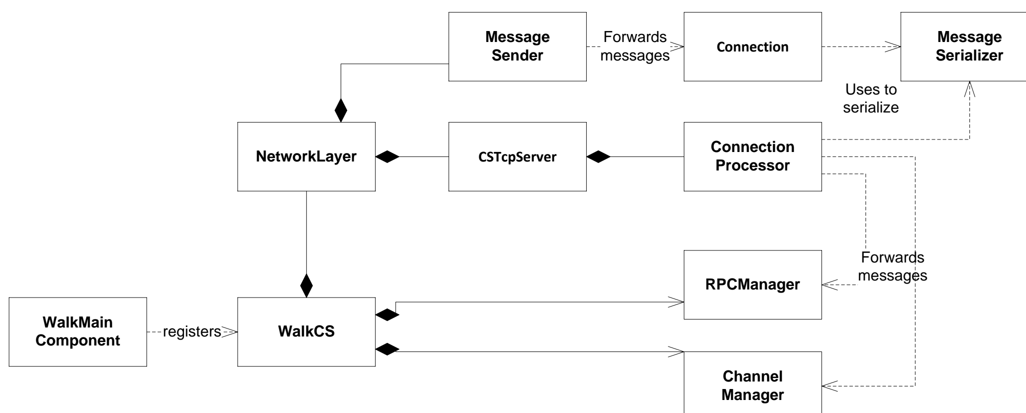


Abbildung 4.7.: Hauptklassen des WalkCS

Nachricht und leitet sie lokal an die entsprechende Komponente weiter. Diese Komponente verarbeitet die Nachricht und gibt eine Antwortnachricht an den ConnectionProcessor zurück, welche zurück zum sendenden Host gesendet wird.

WalkCS unterstützt Connection-Multiplexing. Dabei wird eine einzige TCP-Verbindung zwischen zwei Hosts für mehrere Nachrichten parallel genutzt. Nacheinander können mehrere Nachrichten über eine Verbindung gesendet werden. Die jeweiligen Antworten auf die Nachrichten können darauf in beliebiger Reihenfolge empfangen werden. Die Nachricht-Antwort-Paare werden durch eine systemweit eindeutige UUID[3] identifiziert. Als Serialisierer wird der native Java-Serialisierer genutzt. Dieser hat von allen untersuchten Alternativen die beste Performance und benötigt vor allem keine Anpassungen an das Datenmodell von WalkSim. Damit können alle Objekte transferiert werden, die java-serialisierbar sind. Für die zusätzliche Verringerung des Datenvolumens unterstützt WalkCS GZIP-Komprimierung.

4.3.1.1. NameService

Der *NameService* ist in jedem WalkCS-System nur einmal vorhanden und verwaltet sowohl alle angemeldeten RPC-Objekte als auch alle Channel-Abonnements.

Auf jedem Host gibt es einen *NameServiceProxy*, welcher den Zugriff auf den eigentlichen Name-Service vornimmt. Der Proxy besitzt einen Cache, in welchem Ergebnisse früherer Anfragen gespeichert werden, sodass nicht für jeden RPC eine erneute Abfrage beim Name-

[3]**U**niversally **U**nique **ID**entifier

Service nötig ist. Wechselt das Zielobjekt eines RPC zwischenzeitlich den Host, wird dies erst nach einem ersten Verbindungsversuch zum alten Host festgestellt, welcher daraufhin mit einer Fehlermeldung antwortet.
Der Proxy speichert auch die Abonnenten von Channel-Nachrichten. Diese Abonnenten können sich sehr häufig ändern. Um trotzdem allen Abonnenten des Channels die Nachricht zustellen zu können, bietet der NameService die Möglichkeit, sich nicht nur für die eigentlichen Nachrichten eines Channels beim NameService zu registrieren, sondern auch für Veränderungen in der Abonnentenliste. So kann der Proxy seinen Cache auf einem aktuellen Stand halten, ohne ein wiederholten Polling beim NameService durchzuführen..

4.3.2. Nachteile

Dass die gängigen Messagebus-Implementierungen einen zentralen Server benutzen und keine verteilte Variante, hat natürlich Gründe. Der Hauptgrund dafür – und gleichzeitig die größte Schwäche des WalkCS – ist, dass jeder Host eine Verbindung zu jedem anderen Host aufbauen können muss, um Nachrichten zu verschicken. Jegliche Art von Firewall verhindert dies, sodass potentielle externe Hosts, die sich hinter einem Router befinden (z.B. Heimnetzwerk, Handys oder Tablets) keinen Zugriff auf das System haben werden. Es kann jedoch davon ausgegangen werden, dass Simulationen vorwiegend in geschlossenen Clustern ausgeführt werden, bei denen sich alle Server im gleichen Subnetz befinden und diese Einschränkung somit kein Problem darstellt. Für externe Hosts ist es zudem theoretisch möglich, mit Hilfe eines VPN-Zugangs auch Firewalls zu umgehen.

Ein zweiter Nachteil ist die verteilte Haltung der Channel-Abonnements. Es kann nicht garantiert werden, dass ein Host sich für einen Channel registriert und ab diesem Zeitpunkt sofort alle Nachrichten des Channels erhält, da das Abonnement des Channels zuerst vom NameService im Netzwerk publiziert werden muss und dies ebenfalls über Channels geschieht. In kritischen Situationen muss der Abonnent dafür selbst eine Lösung finden. Bei der blockierenden Synchronisierung des DVE wird als Fallback deshalb ein RPC-Aufruf benutzt.

Vor allem durch das Firewall-Problem ist der hier entwickelte Messagebus sicherlich für fast alle anderen Anwendungsfälle nicht anwendbar. Die Nachteile der Implementierung sind für WalkSim jedoch bei weitem nicht so schwerwiegend wie die Vorteile, die die Einsparungen in der Nachrichtenübertragung bieten.

4.4. Agenten und KI

Wie in Multiagentensysteme üblich arbeiten auch in WalkSim die Agenten ereignisorientiert. Das Scheduling der Ereignisse übernimmt der AgentExecutor (vgl. Abb. 4.8), den jeder Agent

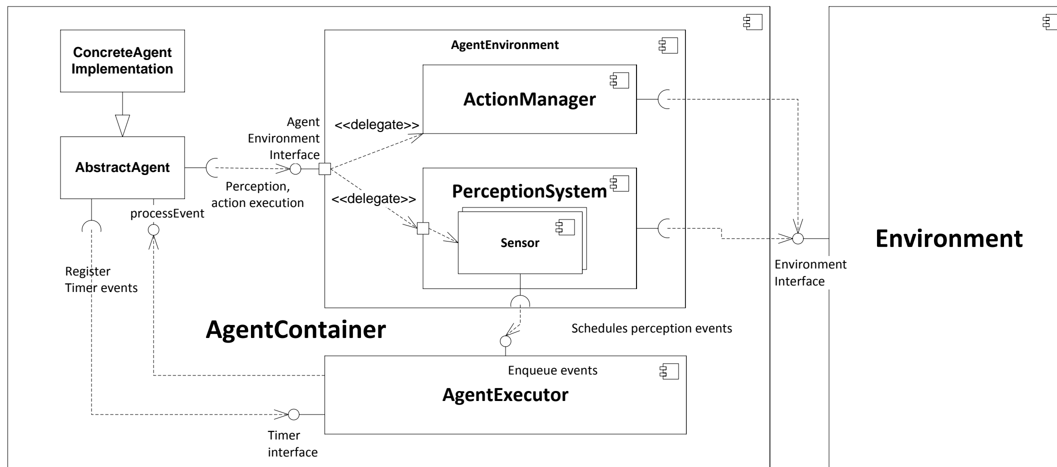
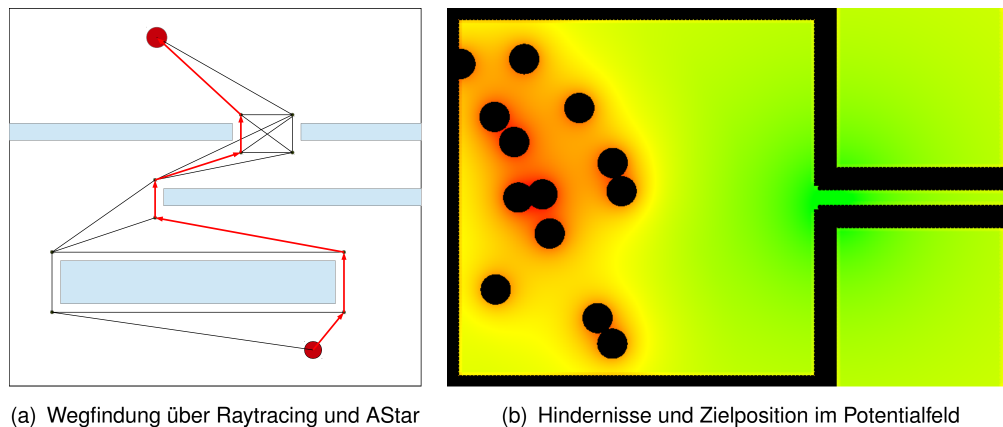


Abbildung 4.8.: Komponenten des Agenten

individuell besitzt. Dieser verwaltet alle Ereignisse, die den Agenten erreichen sollen, und reicht diese nacheinander an ihn weiter. Die Aktivität des AgentExecutors wird dabei zentral von der Agentenplattform gesteuert, da WalkSim Simulationen getaktet ausführt.

Die Agenten selbst sind in ihrer Implementierung weitgehend frei programmierbar und verhalten sich wie eine Blackbox. Über eine Schnittstelle werden dem Agenten die Ereignisse zugespielt, die intern bearbeitet werden und entsprechende Handlungen zur Folge haben. Über das *AgentEnvironment* haben die Agenten dabei Zugriff auf ihre Umwelt und können Aktionen durchführen (vgl. Abb. 4.8).
Der Zugriff auf die Umwelt geschieht mit Hilfe von verschiedenen Sensoren, die die einzelnen menschlichen Sinne darstellen (Sehen, Hören, Fühlen etc.). Die Sensoren fragen dafür die lokale DVE-Instanz ab und stellen den Agenten die Daten aufbereitet zur Verfügung. Dabei steht dem Agenten neben der klassischen kontinuierlichen Abfrage der Sensoren auch eine Push-Methode zur Verfügung, bei der die Sensoren dem Agenten per Ereignis die Änderung der Umwelt mitteilen, ohne dass der Agent sie konkret abfragen muss.
Dem Agenten stehen eine begrenzte Menge an Elementaraktionen zur Verfügung, mit denen er sich oder seine Umwelt verändern kann. Zur Zeit ist der Agent lediglich auf eine *MoveAction* limitiert, welche die Position des Agenten verändert. Die Menge der Aktionen ist jedoch beliebig erweiterbar.
Weiterhin ist es den Agenten möglich, zeitgesteuerte Ereignisse zu beantragen. Durch diese können Agenten in individuellen Intervallen aufgeweckt werden, selbst wenn keine sonstigen Wahrnehmungsereignisse auftreten.

4.4.1. Verwendete Agenten-KI



(a) Wegfindung über Raytracing und AStar (b) Hindernisse und Zielposition im Potentialfeld

Abbildung 4.9.: Wegfindung des Agenten

Die in dieser Arbeit verwendete KI des Agenten ist möglichst simpel gehalten. Es ist dem Agenten möglich, sicher eine individuell definierte Zielposition zu erreichen und dabei auch beweglichen Hindernissen auszuweichen. Erweitere Zielsuchfunktionen und Interaktionen mit der Umwelt oder anderen Agenten sind nicht implementiert.
Die Wegfindung passiert in zwei Schritten.

Der ersten Schritt führt eine grobe Wegfindung mit Hilfe des AStar-Algorithmus durch, mit welcher der Agent einen generellen Weg zwischen statischen Hindernissen hindurch zum Ziel errechnet. Dieser AStar benutzt jedoch kein homogenes, engmaschiges Gitternetz für die Navigation, sondern einen Raytracing-Graphen, welcher lediglich die Hindernisse selbst enthält. Dafür werden zuerst die Eckpunkte aller Hindernisse aus der Umgebung ausgelesen und als Ecken in einem Graph gespeichert. Im Anschluss werden mittels Raytracing für jede dieser Ecken alle anderen Ecken ermittelt, die unter Berücksichtigung der statischen Hindernisse sichtbar sind und entsprechend Kanten im Graphen erstellt (siehe Abb. 4.9(a)). Dabei entsteht ein Graph, der alle Hindernis-Ecken miteinander verbindet, die gegenseitig sichtbar sind. Die Erstellung dieses Graphen hat mit $O(\frac{n^3}{4})$ (n = Anzahl der Hindernisse) einen recht hohen Aufwand, jedoch muss der Graph lediglich einmal zu Beginn der Simulation aufgebaut werden und wird intern in einem Cache vorgehalten.
Mit diesem Graphen kann nun ein AStar-Algorithmus ausgeführt werden, welcher den kürzesten Pfand von einem Start zu einem Zielpunkt berechnet (rote Pfeile in der Abbildung). Der Vorteil des Raytracing-Graphen ist dabei, dass er viel weniger Knoten hat als ein üblicher, aus Kacheln bestehender Graph und zur Laufzeit so um ein Vielfaches schneller ist. Zusätzlich ist die Entfernung zwischen den einzelnen Wegpunkten recht groß, sodass der Agent auf dem

Weg dorthin einen gewissen Spielraum hat, um dynamischen Hindernissen auszuweichen, die nicht im Graphen vorhanden sind (z.B. anderen Agenten).

Im zweiten Schritt findet mit Hilfe von Potentialfeldern eine lokale Wegfindung statt (siehe Abb. 4.9(b)). Dabei besitzt jedes Hindernis ein negatives Potentialfeld (rot oder schwarz) und der aktuelle Zielwegpunkt ein positives (grün). Das Potentialfeld wird bei jeder neuen Bewegungsberechnung neu aufgebaut und stellt somit immer den aktuellen Zustand der Nachbarschaft dar. Der Agent bewegt sich nun immer in die Richtung, in der er lokal das höchste Potential erreichen kann. So nähert er sich seinem Zielpunkt und kann gleichzeitig anderen Agenten ausweichen.

Weitere Wegfindungsfunktionen wie die Reduzierung der Geschwindigkeit bei Stau, Flocking oder intelligentes Ausweichen sind nicht implementiert. Letzteres kann hin und wieder dazu führen, dass zwei Agenten vor einer Tür feststecken, weil sie kontinuierlich beide gleichzeitig durch die Tür gehen wollen, was durch die Kollisionserkennung verhindert wird. Ebenso wenig wird die Wegfindung in mehrstöckigen Umgebungen unterstützt.

4.5. Szenarios

```
1  <scenario>
2    <identifier>example01</identifier>
3    <name>Beispiel-Szenario 01</name>
4    <group>Test</group>
5
6    <environment>Rimea01</environment>
7
8    <agentDefinition>
9      <name>agent001</name>
10     <agenttype>EGOAPAgent</agenttype>
11     <agentdeclaration>plainrunner</agentdeclaration>
12
13     <position><x>0.1</x><y>1.3</y><z>0.0</z></position>
14
15     <startParameter>
16       <name>age</name>
17       <integer>40</integer>
18     </startParameter>
19     <startParameter>
20       <name>targetPosition</name>
21       <vector3d><x>1.0</x><y>2.0</y><z>0.0</z></vector3d>
22     </startParameter>
23   </agentDefinition>
24 </scenario>
```

Abbildung 4.10.: Beispiel einer Szenariodefinition in XML

Die Szenarien in WalkSim werden vom *ScenarioContainer* verwaltet. Diese werden als XML-Datei auf dem Datenträger gespeichert und werden beim Programmstart geladen. Neben der Auswahl einer Simulationsumgebung, die den Gebäudeplan enthält, kann in diesen Szenario-Dateien der Ablauf einer Simulation mit folgenden Elementen detailliert beschrieben werden:

- **AgentDefinition**: Definiert einen Agenten mit seinem konkreten Implementierungs-Typ, seiner Startposition und diversen Startparametern, die vom konkreten Typ des Agenten abhängen.
- **AgentCollection**: Bezeichnet die Definition einer ganzen Agentengruppe. Die Agenten werden in einem spezifizierten Areal gleichverteilt.
- **AgentSpawnEvent**: Ermöglicht die dynamische Erstellung von Agenten und Agentengruppen in einer laufenden Simulation. So können Agenten definiert werden, die erst nach einer bestimmten Zeitspanne auftauchen.

Sowohl die Startparameter der Agenten als auch die Anzahl der Agenten und die Zeiten der Ereignisse können mit Zufallszahlengeneratoren bestimmt werden.

4.6. Simulationsaufführung

Wie bereits zuvor beschrieben, findet die Simulationsausführung in WalkSim schrittbasiert statt, sodass es in regelmäßigen Abständen Synchronisationspunkte gibt, an denen die Plattformen aufeinander warten müssen. Durch eine virtuelle Simulationszeit kann die Simulation abgekoppelt von der realen Zeit ausgeführt werden und läuft nicht auseinander, falls eine Komponente für einen Teilschritt mehr Zeit benötigt, als eingeplant ist bzw. wenn die Simulation aufgrund der Größe gar nicht in Realzeit ausgeführt werden kann.

Die Simulationsschleife gliedert sich in folgende Teilschritte:

1. **Szenarioereignisse**: Alle Szenarioereignisse, die im nächsten Schritt auftreten, werden behandelt und ausgeführt.
2. **Agentenschritt**: Die Agenten auf den einzelnen Servern verarbeiten alle Ereignisse, die innerhalb des nächsten Schrittes anfallen und handeln entsprechend ihrer Implementierung danach.
3. **Synchronisation**: Zu Beginn der Synchronisation wird die Kollisionserkennung ausgeführt und entsprechend auftretende Kollisionen beseitigt. Anschließend synchronisieren sich die einzelnen Server untereinander.
4. **Partitionierung**: Die Verteilung der Agenten auf die teilnehmenden Agentenserver wird neu berechnet.

5. **Migration**: Die neu berechnete Verteilung wird den Agentenservern mitgeteilt und diese migrieren Agenten ggf. auf ihre neu zugewiesenen Server.

In den Schritten zwei, drei und fünf wird ein Kommando auf dem Command-Channel der Simulation über das Kommunikationssystem veröffentlicht und so von allen Plattformen parallel empfangen. Der SimulationExecutor blockiert solange, bis jede Plattform mit einer weiteren Nachricht die vollständige Ausführung des Kommandos bestätigt hat.

Die Länge eines solchen Schrittes ist frei konfigurierbar und liegt standardmäßig bei 40 Millisekunden Simulationszeit. Je kleiner die Schrittlänge ist, desto genauer arbeitet die Kollisionserkennung und desto “gleicher” sind die Datenbestände auf den einzelnen Servern. Je länger der Schritt ist, desto performanter ist das System, da die Agenten seltener in ihrer Arbeit unterbrochen werden.

5. Benchmarks und Experimente

Um die in den vorigen Kapiteln aufgestellten Thesen zu beweisen, werden diverse Testfälle und Experimente durchgeführt. Zum einen geht es darum, zu zeigen, ob die partielle Synchronisation tatsächlich besser ist als die vollständige Synchronisation. Zum anderen soll untersucht werden, welche der implementierten Partitionierungsalgorithmen am besten mit der Sektoraufteilung arbeiten kann. Dafür werden verschiedene Extremszenarien erstellt, die die Algorithmen auf unterschiedliche Weise herausfordern.
Weiterhin soll mit Hilfe von Benchmarks die pure Leistung von WalkSim untersucht werden. Dafür ist zu ermitteln, wie performant die einzelnen Komponenten arbeiten, wie skalierbar das Gesamtsystem ist und wie viele Agenten auf einer festgelegten Experimentierplattform sinnvoll ausgeführt werden können.

Explizit nicht Teil der Tests sind sowohl die Physik als auch die KI-Modellierung der Agenten. Die Physik ist lediglich durch eine rudimentäre Kollisionserkennung implementiert und auch die KI bietet lediglich Eigenschaften, die für die flüssige Fortbewegung der Agenten zwingend nötig sind. Die Szenarien werden so entworfen, dass es keine Engstellen gibt, an denen sich potentiell Staus entwickeln können, um eine flüssige Bewegung zu garantieren.

5.1. Testszenarien

Um die verschiedenen Eigenschaften von WalkSim und der Algorithmen zu untersuchen, werden verschiedene Szenarien und Terrainumgebungen entworfen.

Die Testumgebung ist, soweit nicht anders angegeben, eine Menge virtueller Maschinen, die auf einer Workstation ausgeführt werden. Die Workstation kann auf zwei Prozessoren des Typs Intel Xeon 5600 (6 Kerne@2.8GHz plus Hyper-Threading) und insgesamt 48 GB DDR3-RAM zugreifen. Die verwendeten virtuellen Maschinen haben jeweils Zugriff auf 2 Kerne und 4 GB RAM. Als Betriebssystem wird die Linux Distribution Ubuntu-Server in der Version 12.04 und 64-Bit verwendet, die Java-VM ist die aktuelle Version OpenJDK 7. Der Haupt-Server, auf welchem der WalkManager und auch der NameService ausgeführt wird, kann auf 4 Kerne und 4 GB zurückgreifen.

Die Umwelt ist in Sektoren aufgeteilt, die in ihrer Form und Größe an die Umgebung angepasst sind. Zwei Sektoren besitzen eine Nachbarschaftsbeziehung, wenn es einem Agenten möglich

ist, von einem Sektor direkt in den anderen zu gelangen, ohne einen anderen Sektor oder ein Hindernis zu durchqueren. Wenn es nicht anders angegeben ist, bewegen sich alle Agenten mit einer individuell zufälligen, aber über die Simulation konstanten Geschwindigkeit von 1.0 bis 1.5 m/s. Die Ausmaße eines Agenten werden durch eine zu den Koordinatenachsen parallele Box mit einer Kantenlänge von 0.4 m beschrieben.

5.1.1. Messgrößen

Für die Beurteilung der Tests werden diverse Laufzeitmessungen vorgenommen. Die Messungen, die die Simulation selbst betreffen, werden für jeden Simulationsschritt durchgeführt und soweit möglich auch für jede Plattform einzeln. Messungen, die das Hostsystem betreffen, werden sekündlich von einem Timer gesteuert aufgenommen und ebenfalls separat für jede Plattform gespeichert.

WalkCS

- Anzahl der insgesamt geschriebenen und gelesenen Bytes (nicht nach Verbindung aufgeschlüsselt)
- Anzahl der insgesamt über das Netzwerk gesendeten Channel-Nachrichten. Dabei wird jede Nachricht bei mehreren Empfängern entsprechend mehrfach gezählt.
- Gesamtzahl der RPC-Aufrufe (nicht nach Komponente oder Methode aufgeschlüsselt)

Hostplattform:

- Durchschnittliche CPU-Auslastung der letzten Sekunde. Dieser Wert wird ermittelt durch die Messung der verbrauchten CPU-Zeit durch die gesamte JavaVM seit der letzten Messung geteilt durch die Anzahl der verfügbaren Kerne. Der gemessene Wert liegt damit im Bereich zwischen 0 und 1.
- Durchschnittliche CPU-Auslastung, die durch den Garbage-Collector der JavaVM in der letzten Sekunde verbraucht wurde. Die Messung erfolgt analog zur Gesamt-CPU-Auslastung der JavaVM.

Simulation:

- Dauer der einzelnen Simulationsschritte (Szenarioereignisse, Agenten, Synchronisation, Partitionierung, Migrationscheck) zentral im SimulationExecutor.
- Dauer der Kollisionserkennung, der eigentlichen Synchronisation und der Ermittlung der für die Partitionierung erforderlichen Daten separat auf jeder Agentenplattform, da der SimulationExecutor diese drei Schritte durch das Kommando *Synchronisation* zusammenfasst.

- Anzahl der empfangenen Objekt-Aktualisierungen
- Anzahl der abonnierten Sektoren
- Anzahl der ausgeführten Agenten
- Anzahl der Agenten, die auf andere Plattformen migriert wurden

5.1.2. Synchronisation

Um die Performance der Synchronisation zu testen, wird eine möglichst große, hindernisfreie Umgebung benötigt. Dafür wurde ein einfacher Raum mit den Außenmaßen 3000 x 3000 m erstellt, welcher außer den Außenwänden keinerlei Hindernisse enthält. Die Umwelt wird schachbrettartig in 900 quadratische Sektoren der Größe 100 x 100 m geteilt, wobei aneinander grenzende Sektoren auch im Sektorbaum eine Nachbarschaftsbeziehung besitzen.

Auf diesem Gelände werden verschiedene Anzahlen von Agenten mit Hilfe einer Gleichverteilung zufällig verteilt. Trotz der zufälligen Verteilung kann bei der großen Menge von Agenten davon ausgegangen werden, dass die Verteilung weitgehend homogen ist. Jeder Agent bekommt beim Start eine zufällige Zielposition im Gesamtgebiet zugeteilt. Das sorgt dafür, dass die Agenten möglichst wild durcheinander laufen und keine erkennbaren Gruppen entstehen.

Es werden Szenarien mit 50.000, 100.000, 200.000, 300.000 und 500.000 Agenten durchgeführt. Jede Anzahl von Agenten wird zudem mit unterschiedlicher Serveranzahl getestet, soweit dies machbar ist. Getestet wurden alle Anzahlen auf jeweils vier und acht Servern, die Mengen 50.000 bis 300.000 zudem auch mit zwei Servern. 500.000 Agenten ließen sich aus Speicherproblemen nicht auf zwei Server verteilen, da 4 GB Hauptspeicher nicht für 250.000 Agenten ausreichten. Eine Erhöhung des Speichers für die Simulation auf zwei Servern hätte dazu geführt, dass die Ergebnisse nicht mehr vergleichbar gewesen wären.

Für sämtliche Tests wird der Partitionierungsalgorithmus KMeans verwendet, da dieser die geringste Last erzeugt.

Ziel dieses Tests ist es, zum einen die Synchronisierungsmodelle bei möglichst großer Last zu vergleichen. Das simple Szenario ohne Hindernisse wurde gewählt, damit die Synchronisationsmenge je Sektor möglichst gleich ist und keine Verfälschungen durch Engstellen oder andere szenario- oder umgebungsbedingte Faktoren entstehen. Die maximal einfache Umwelt sorgt auch dafür, dass die Agenten nur eine minimale Last erzeugen, da ohne Hindernisse keine Raytracing-Wegfindung nötig ist.

5.1.3. Skalierbarkeit

Mit einem weiteren Performance-Test soll untersucht werden, wie gut WalkSim auf viele Server skaliert. Dafür wurden mehrere Szenarien entworfen. Im ersten Test wird geprüft, wie sich die Performance des Systems verändert, wenn eine fixe Szenariokonfiguration auf unterschiedlich vielen Servern ausgeführt wird. Der Test besteht aus einem einzelnen Raum der Größe 8.000 x 750 m. In diesem Raum sind 80.000 Agenten zufällig mit einer Gleichverteilung angeordnet. Wie bereits in den Skalierungstestfällen haben die Agenten einen zufälligen Zielpunkt irgendwo im gegebenen Raum, zu dem sie sich möglichst linear hinbewegen. Die Form des Simulationsraumes ist bewusst lang und schmal gehalten.
In 15 Testläufen wird dieser Testfall nun auf 2 bis 16 virtuellen Servern ausgeführt. Die Server mussten im Vergleich zum vorigen Synchronisationstest jedoch auf einen Kern und 2 GB RAM reduziert werden.

Der zweite Skalierungstest untersucht, wie sich der Serververbund verhält, wenn sowohl Serveranzahl als auch Simulationsgröße ansteigen. Für diesen Test wurde für jede Serveranzahl eine eigene Simulationsumgebung erstellt. Die Simulation für zwei Server besteht aus einem Raum der Größe 1.000 x 750 m, in welchem 10.000 Agenten gleichverteilt sind. Für jeden weiteren Server wurde der Raum um 500 m verlängert und die Agentenzahl um 5.000 Agenten erhöht. Dadurch ist die nominelle Last pro Server, die durch die Agenten erzeugt wird, über alle Testläufe hinaus konstant.
Genauso wie der erste Skalierungstest wird dieser Testfall ebenso auf 2 bis 16 Servern ausgeführt.

5.1.4. Partitionierung

Um die einzelnen Partitionierungsalgorithmen zu testen, wurden mehrere einzelne Szenarien entworfen, die für sich Extremsituationen darstellen. Auch wenn diese Situationen wahrscheinlich nur vereinzelt in realen Simulationen auftreten werden, sind sie gut geeignet, um die Grenzen der einzelnen Algorithmen aufzuzeigen.

5.1.4.1. Test 1: Einzelgruppen

Der Einzelgruppentest soll prüfen, wie die Algorithmen mit Agenten umgehen können, die sich in größeren Gruppen separat zueinander bewegen. Dafür wurde eine simple quadratische Umgebung mit einer Kantenlänge von 500 m erstellt (vgl. Abb 5.1) und in 2.500 Sektoren der Kantenlänge 10x10m aufgeteilt. In dieser Umgebung wurden acht Agentengruppen von je 100 Agenten verteilt. Alle Agenten bewegen sich mit einer konstanten Geschwindigkeit von 1 m/s auf einem quadratischen Pfad entlang. Die Startpositionen und Pfade sind so angelegt, dass

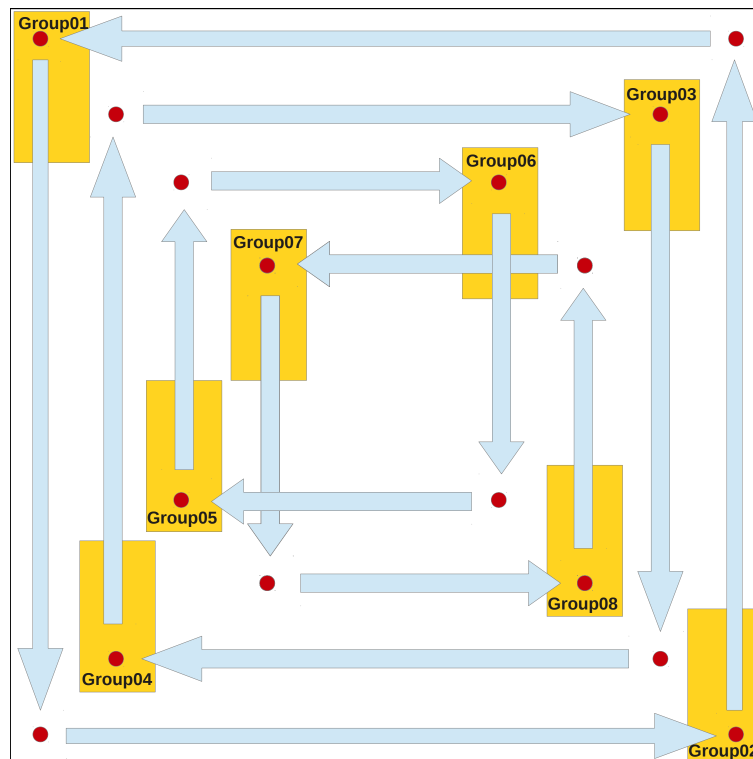


Abbildung 5.1.: Aufbau von Test 1: Einzelgruppen

sich zwar die Pfade überschneiden, es jedoch nie dazu kommt, dass ein Agent andere Agenten von anderen Gruppen sehen kann.

Der Test wird auf jeweils zwei, vier und acht Servern ausgeführt.
Der perfekte Algorithmus würde in diesem Szenario die acht Gruppen erkennen und im Fall von acht verfügbaren Servern jeweils jedem Server eine Gruppe zuweisen. Dadurch, dass die Gruppen stets genügend Abstand zueinander haben, ist es nicht erforderlich, Agenten in andere Sektoren zu migrieren. Es genügt, die Verantwortlichkeit der Sektoren zu ändern, sobald ein Agent diese betritt. Ebenso ist es durch den Sicherheitsabstand nicht nötig, eine Synchronisation durchzuführen, da die Agenten keine Fremdsektoren in ihrem Sichtbereich haben.

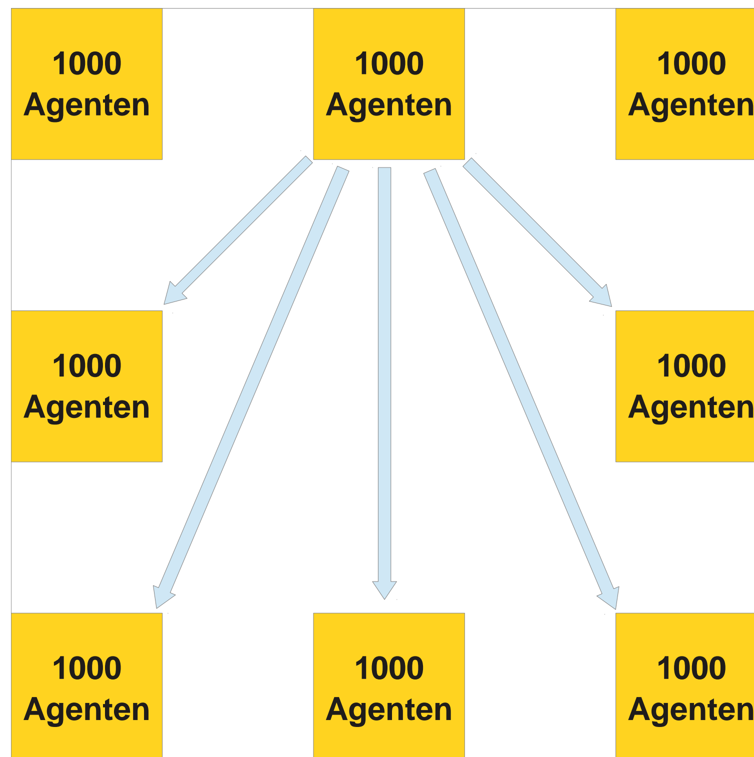


Abbildung 5.2.: Aufbau von Test 2: Chaotische Menge (Bewegungsrichtung exemplarisch)

5.1.4.2. Test 2: Chaotische Menge

Der zweite Test soll das Verhalten der Algorithmen untersuchen, für den Fall, dass sich die Agenten ohne erkennbare Gruppen oder zentrale Wege wild durcheinander bewegen. Dafür wurde eine 500 x 500 m große einfache Umgebung ohne Hindernisse erstellt und in 20 x 20 Sektoren aufgeteilt. Am Rand dieses Raumes wurden sowohl an den vier Ecken als auch in der Mitte der Kanten jeweils 1000 Agenten platziert (vgl. Abb. 5.2). Jede dieser Agentengruppen besteht wiederum aus fünf Gruppen zu 200 Agenten. Jede dieser Gruppen hat einen der fünf am weitesten entfernten Startbereiche als Zielpunkt. Hat ein Agent seinen Zielbereich erreicht, wird er auf seine Startposition zurückgesetzt und bewegt sich erneut zu seinem Ziel.

Der Test wird auf jeweils zwei, vier und acht Servern ausgeführt.
Das erwartete Ergebnis für den Fall von acht Servern ist, dass sich die acht Startbereiche auf die acht Server verteilen und sich die Partitionen zu Beginn keilförmig in Richtung Mitte zuspitzen. Diese Verteilung sollte sich im Verlauf der Simulation auch nicht maßgeblich ändern, da die Agenten sich annähernd symmetrisch und gleichmäßig in der Umgebung verteilen.

5.1.4.3. Test 3: Evakuierung mit einem Notausgang

Der dritte Test untersucht das Verhalten der Algorithmen, wenn Teilbereiche der Simulation besonders dicht besiedelt sind während die anderen Bereiche nahezu leer sind. Dafür wurde in einer Umgebung 500 x 1.000 m großer Raum mit 20 x 40 quadratischen Sektoren erstellt. An der unteren breiten Seite besitzt der Raum mittig einen 30 m breiten Durchgang, welcher zu einem weiteren, aber kleineren Raum führt. Im großen Raum werden 10.000 Agenten gleichverteilt angeordnet und haben alle das Ziel durch den Durchgang hindurch in den kleineren Raum zu gelangen. Haben sie den Durchgang ein paar Meter durchschritten, verschwinden die Agenten automatisch aus der Simulation.

Die Simulation wird auf zwei, vier und acht Servern durchgeführt. Das ideale Ergebnis wäre eine gleichmäßige, keilförmige Partitionsaufteilung, bei der die Partitionen wie Tortenscheiben mit dem Durchgang als Spitze verteilt sind. So bleiben die Agenten auf dem Weg bis zum Durchgang stets in ihrer Partition und die Partition bleibt weitgehend unverändert.

5.1.4.4. Test 4: Homogenes Bewegen einer breiten Masse

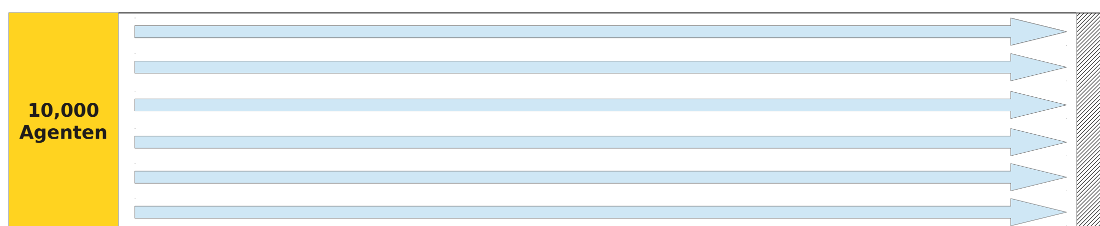


Abbildung 5.3.: Aufbau von Test 4: Homogene Bewegung

Test 4 untersucht die gleichförmige Bewegung einer großen Masse von Agenten. In diesem Test bewegen sich 25.000 Agenten einen 400 m breiten und 2.000 m langen Gang entlang. Der Gang ist unterteilt in 20 x 20 m große Sektoren. Die Geschwindigkeit der Agenten beträgt konstante 2 m/s. Die Agenten bewegen sich nahezu parallel zueinander durch den Gang und kreuzen nicht die Wege anderer Agenten. Aus technischen Gründen ist eine echte parallele Bewegung für die KI der Agenten nicht möglich und wird durch mehrere nebeneinander gelegene Zielpunkte simuliert, von denen der Agent immer den dichtesten anstrebt. Das führt zu einer leichten Aufspaltung der Agentenmenge in mehrere Keile. Dies ist für diesen konkreten Test jedoch nicht relevant.

Wie die anderen auch, wird der Test ebenfalls auf zwei, vier und acht Servern ausgeführt.
Die optimale Partitionierung besteht aus mehreren schmalen, langen Partitionen, die sich ne-

beneinander liegend über die gesamte Länge des Ganges ziehen. Dadurch bewegen sich die Agenten immer in der gleichen Partition und es ist keine Agentenmigration nötig.

5.1.4.5. Test 5: U-Bahn-Station/Bahnhof

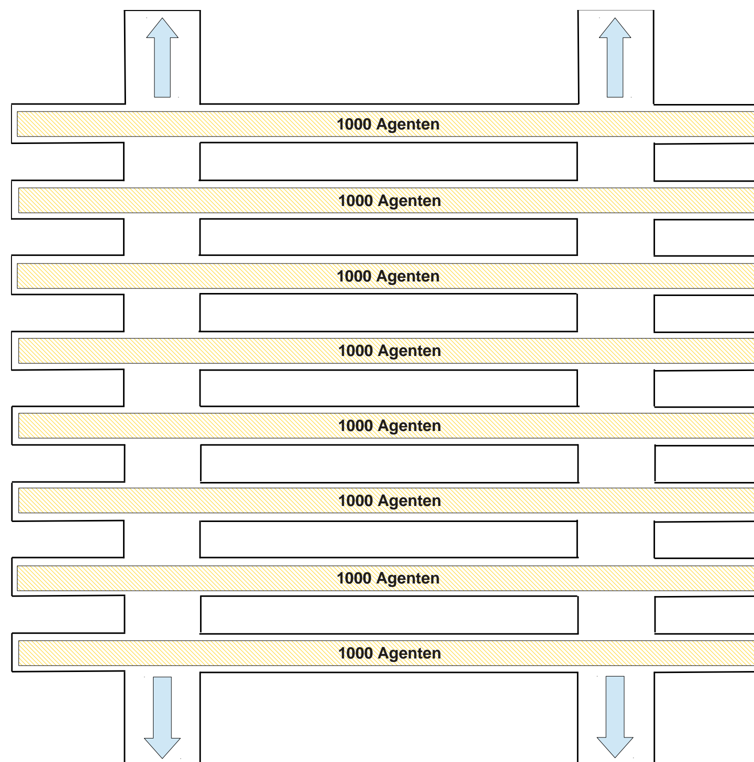


Abbildung 5.4.: Aufbau von Test 5: Bahnhof

Test 5 simuliert einen Bahnhof mit mehreren Bahnsteigen, die durch zwei Kreuzgänge miteinander verbunden sind. Die insgesamt 1.000 x 1.000 m große Umgebung besteht aus acht Bahnsteigen, die jeweils eine Länge von 1.000 m und eine Breite von 50 m haben (siehe Abb. 5.4). Die beiden Unterführungen verlaufen senkrecht zu den Bahnsteigen und haben eine Größe von 1.000 x 100 m. Die Kreuzgänge schneiden die Bahnsteige so, dass die Bahnsteige auf den jeweiligen Außenseiten um 125 m über die Kreuzung hinaus ragen. Die Bahnsteige und die Unterführungen sind jeweils in 25 x 25 m große Sektoren unterteilt, was in Summe 832 Sektoren ergibt.

Über die komplette Länge aller acht Bahnsteige sind pro Bahnsteig jeweils 1.000 Agenten

homogen verteilt. An den vier Enden der Unterführungen befindet sich jeweils ein Ausgang. Die Agenten versuchen während der Simulation zu dem ihnen am nächsten gelegenen Ausgang zu gelangen und verschwinden, sobald sie diesen erreicht haben.

Ziel dieses Tests ist es, eine Umgebung zu untersuchen, in der die Sektoren nicht wie ein Gitternetz angeordnet sind, sondern Faktoren aus der Umwelt mit einbeziehen (z.B. nicht begehbare Schienen o.ä.). Der Partitionierungsalgorithmus sollte dies an den gespeicherten Nachbarschaftsbeziehungen erkennen und die Partitionen entlang dieser erstellen. Auf keinen Fall sollte sich eine Partition so über zwei Bahnsteige erstrecken, dass zwei separate Partitionen entstehen.

5.2. Nicht Teil der Tests

Einige Algorithmen und Parameter werden hier nicht weiter untersucht, da deren Analyse den Rahmen dieser Arbeit sprengen würde. Nichtsdestotrotz ist es zu empfehlen, in anderen Arbeiten Untersuchungen dazu anzustellen, da diese Parameter die Simulation sowohl im Ergebnis als auch in ihrer Performance beeinflussen können.

5.2.1. Spatiale Hashdatenbank

Die Leistung einer spatialen Hashdatenbank ist hauptsächlich abhängig von zwei Parametern. Die Größe der Hashbuckets beeinflusst, wie viele Objekte in einem einzelnen Hashbucket gespeichert werden und entsprechend bei einer Abfrage alle untersucht werden müssen. Wie dieser Parameter eingestellt werden muss, kann nicht pauschal beantwortet werden, sondern hängt sehr von den letztlich in der Praxis verwendeten Szenarien ab und wie sich dort die jeweilige Dichteverteilung der Agenten verhält. Da im WALK-Projekt noch keine praxistauglichen Szenarien entwickelt wurden, die von der Größe her für einen Benchmark geeignet wären, kann dieser Parameter hier noch nicht untersucht werden.
Der zweite Parameter ist die Größe der gesamten Hashtabelle. Die Justierung dieses Parameters hängt direkt von der Größe der einzelnen Hashbuckets ab, da bei größeren Buckets bei einer gleichgroßen Umgebung weniger Hasheinträge benötigt werden. Idealerweise sollte die Hashtabelle so groß sein, dass gar keine Kollisionen auftreten. Inwiefern das letztlich sinnvoll ist, hängt jedoch auch von der Größe des verfügbaren Hauptspeichers ab, da mit größerem Java-Heap auch die CPU-Zeit des Garbage-Collectors zunimmt.

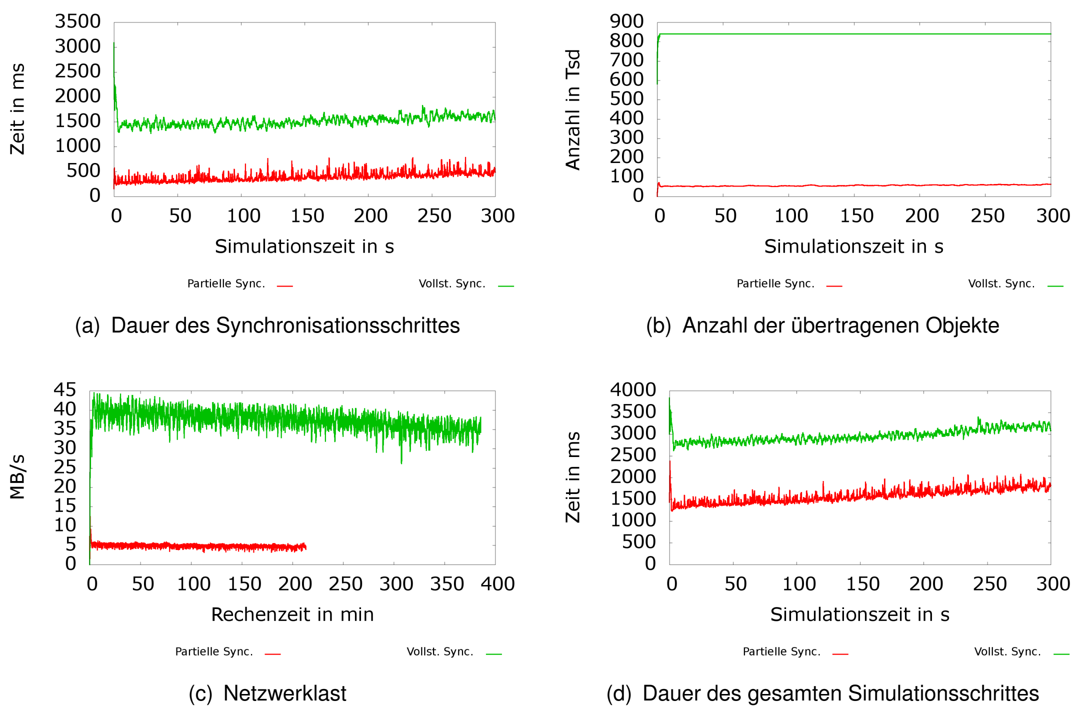
5.2.2. Simulationsschrittweite

Die Simulationsschrittweite ist der Takt jeder Walk-Simulation. Je geringer die Schrittweite ist, desto genauer funktionieren Synchronisation, Partitionierung und Kollisionserkennung. Die gesamte Simulation wird dafür jedoch deutlich langsamer, da all diese Teilschritte viel häufiger ausgeführt werden und im Verhältnis zur Agentenrechenzeit viel mehr Gewicht bekommen.
Je höher auf der anderen Seite die Schrittweite ist, desto ungenauer wird die Simulation, weil eben diese für die Konsistenz wichtigen Schritte, nicht mehr häufig genug ausgeführt werden. Letztlich beeinflusst die Schrittweite damit das Ergebnis der Simulation, da so Kollisionen eventuell zu spät oder gar nicht aufgelöst werden und Agenten aufgrund seltenerer Synchronisierung mit alten Datenbeständen arbeiten.

Welcher Wert als Schrittweite der beste Spagat aus Konsistenz und Performance ist, kann nur mit einer fachlichen Validierung des Simulationsergebnisses ermittelt werden, jedoch ist der fachliche Teil der Agenten nicht Teil dieser Arbeit und wird dementsprechend nicht weiter verfolgt.

6. Ergebnisse

6.1. Vergleich der Synchronisation



(a) Dauer des Synchronisationsschrittes

(b) Anzahl der übertragenen Objekte

(c) Netzwerklast

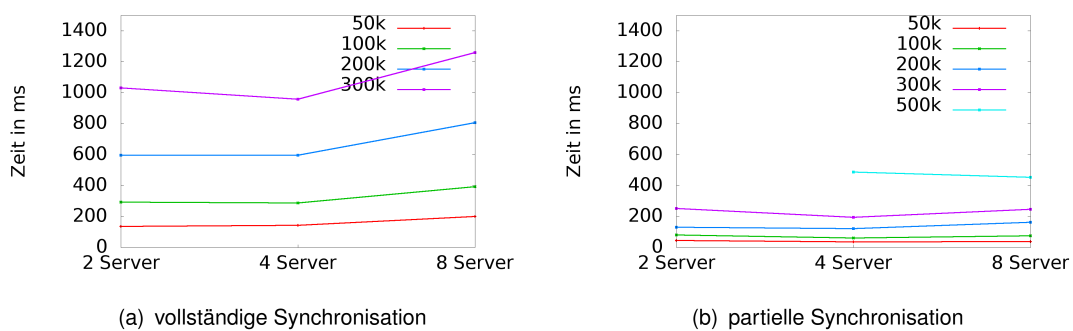
(d) Dauer des gesamten Simulationsschrittes

Abbildung 6.1.: Vergleich vollständiger und partieller Synchronisation bei der Ausführung von 300.000 Agenten auf 8 Servern

Die Ergebnisse beim Vergleich der partiellen Synchronisation mit der vollständigen Synchronisation fallen sehr eindeutig aus. In Abb. 6.1(a) sieht man sehr deutlich, wie viel schneller die partielle Synchronisation ist. Während die partielle Synchronisation pro Simulationsschritt nur ca. 400 ms benötigt, arbeitet die vollständige Synchronisation 1.5 Sekunden, bis alle Plattformen den gleichen Zustand besitzen. Hauptsächlich ist dieser Unterschied damit zu erklären, dass die Anzahl der zu übertragenden Daten sehr unterschiedlich ist. Bei der vollständigen

Synchronisation müssen pro Takt in Summe fast 850.000 Objekte zwischen den acht Servern übertragen werden (siehe Abb. 6.1(b)). Die Übertragung dieser Objekte verursacht eine durchschnittliche Netzwerklast von 35 bis 40 MB/s (vgl. Abb. 6.1(c)), was deutlich höher ist als die durchschnittlichen 5 MB/s bei der partiellen Synchronisation. In einem Simulationsschritt muss bei der vollständigen Synchronisation so eine Datenmenge von ca. 110 MB übertragen werden, während die partielle Synchronisation nur ca. 8 bis 9 MB pro Takt übertragen muss.
In Abb. 6.1(d) wird ersichtlich, dass dieser Unterschied bei der Synchronisation einen signifikanten Einfluss auf die Dauer der Simulation hat.

6.1.1. Synchronisationsdauer



(a) vollständige Synchronisation

(b) partielle Synchronisation

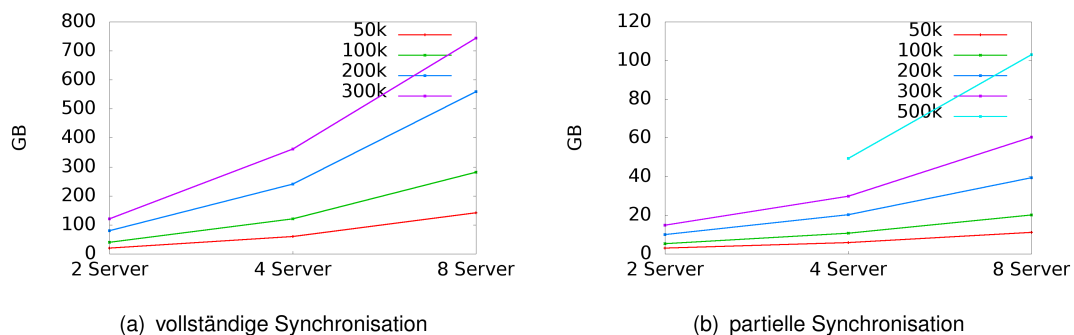
Abbildung 6.2.: Durchschnittliche Dauer der Synchronisationsschritte für verschiedene Agenten- und Serveranzahlen

Abb. 6.2 zeigt, wie sich die Dauer des Synchronisationsschrittes mit Anzahl der Agenten und Server verändert. Man sieht, dass eine höhere Anzahl der Agenten bei beiden Synchronisationsmethoden auch eine längere Synchronisationsdauer zur Folge hat, wobei der Anstieg ungefähr linear ist. Eine Verdopplung der Serveranzahl lässt die Synchronisationsdauer bei der vollständigen Synchronisation vor allem beim Sprung von vier auf acht Server leicht ansteigen (vgl. 6.2(a)). Bei der partiellen Synchronisation hingegen verändert sich die Dauer nur unwesentlich, wenn mehr Server zum Cluster hinzugefügt werden (vgl. 6.2(a)).
Sehr deutlich zu sehen ist, dass die vollständige Synchronisation in jeder Konfiguration deutlich mehr Zeit benötigt als die partielle Synchronisation.

Eine Simulation mit 500.000 Agenten konnte bei der vollständigen Synchronisation nicht sinnvoll ausgeführt und ausgewertet werden, da die 3.5 GB RAM, die der VM zur Verfügung standen, nicht ausreichend waren. Die Simulation konnte zwar ausgeführt werden, die VM war dabei jedoch zu über 80% damit beschäftigt den Garbage-Collector auszuführen.
Dadurch, dass bei der partiellen Synchronisation die zu verwaltenden Datenmengen lokal auf

den Servern geringer war, trat dieses Problem hier nur auf, wenn lediglich zwei Server verwendet wurden. Bei vier und acht Servern war der Hauptspeicher von 3.5 GB ausreichend.

6.1.2. Datenmengen



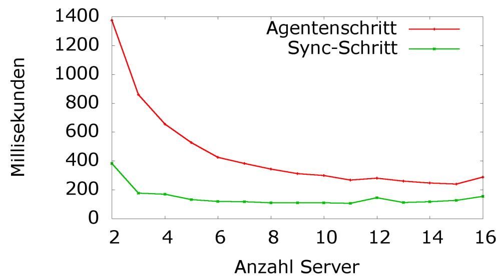
(a) vollständige Synchronisation

(b) partielle Synchronisation

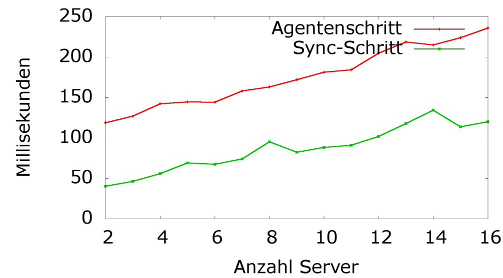
Abbildung 6.3.: Gesamtmenge der übertragenen Daten während der Simulation für verschiedene Agenten- und Serveranzahlen

Die Abb. 6.3 stellt die gesamte Datenmenge dar, die während einer Simulation zwischen allen Servern übertragen wurde. Es ist sehr deutlich zu sehen, wie die Menge der Daten sowohl im Verhältnis zur Anzahl der Agenten als auch zur Anzahl der Server stetig ansteigt. Der Anstieg verhält sich dabei weitgehend linear.
Sehr deutlich ist dabei erneut der Unterschied zwischen vollständiger und partieller Synchronisation. Abhängig von der Anzahl der Server ist die Menge der übertragenen Daten bei der vollständigen Synchronisation teilweise über zehnmal so hoch wie bei der partiellen Synchronisation.

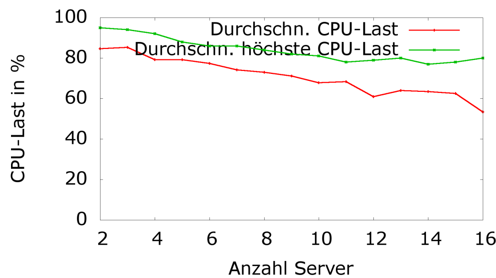
6.2. Skalierbarkeit



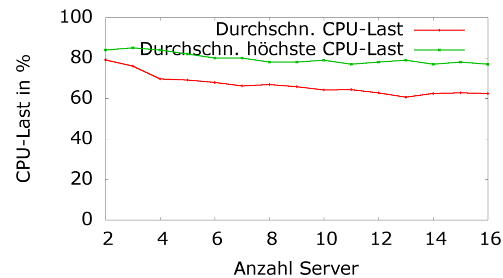
(a) Dauer der Teilschritte bei der Ausführung von 80.000 Agenten mit verschiedenen Serveranzahlen



(b) Dauer der Teilschritte bei linearer Erhöhung von Serveranzahl und Simulationsgröße um 5.000 Agenten je Server



(c) CPU-Auslastung bei der Ausführung von 80.000 Agenten mit verschiedenen Serveranzahlen



(d) CPU-Auslastung bei linearer Erhöhung von Serveranzahl und Simulationsgröße um 5.000 Agenten je Server

Abbildung 6.4.: Ergebnisse des Skalierungstests

In der Abbildung 6.4 sind die Ergebnisse der zwei Skalierungstests dargestellt.

Im Test mit der konstanten Anzahl an Agenten nahm die Dauer des Agentenschrittes kontinuierlich ab, je mehr Server man dem Verbund hinzufügt (vgl. Abb. 6.4(a)). Mit höherer Serveranzahl geht der Performancegewinn jedoch gegen null und die Dauer des Agentenschrittes pendelt sich bei ca. 250 bis 300 ms ein.
Die für die Synchronisation benötigte Zeit sinkt überraschenderweise bei steigender Serveranzahl kurz ab und bleibt anschließend konstant bei 100 bis 130 ms. Gleichzeitig steigt die insgesamt während der Simulation übertragene Datenmenge linear zur Anzahl der Server an.

In der Abb. 6.4(b) sind die Ergebnisse des zweiten Tests zu sehen. Hier steigt sowohl die Zeit für den Agentenschritt als auch die für den Synchronisationsschritt konstant an. Die Ausführung von 10.000 Agenten auf zwei Servern benötigt dabei nur halb so lange wie die Ausführung

von 80.000 Agenten auf 16 Servern. Der Anstieg ist bei der Synchronisation deutlicher zu sehen. Hier benötigen 16 Server ca. dreimal so lange, sich zu synchronisieren wie zwei Server. Auch bei diesem Test steigt die Anzahl der übertragenen Daten mit Anzahl der Server und Größe der Simulation linear an.

Bei beiden Skalierungstests fällt die CPU-Auslastung, je mehr Server dem Verbund hinzugefügt werden (siehe Abb. 6.4(c) und 6.4(d)). Dabei weist der Test mit konstanten 80.000 Agenten bei geringer Serveranzahl eine höhere CPU-Last auf als der, wo auf den wenigen Servern auch insgesamt weniger Agenten ausgeführt werden.
Die durchschnittlich höchste individuelle Last eines Servers im Verbund lag dabei deutlich über der Gesamtdurchschnittslast, was auf eine nicht optimale Lastverteilung schließen lässt.

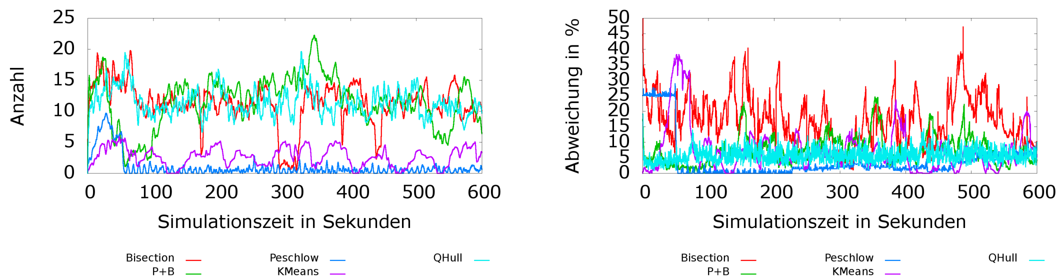
6.3. Partitionierung

Im Folgenden werden die einzelnen Testfälle detailliert betrachtet und für alle Tests die wichtigen Messdaten in Tabellen und Grafiken veranschaulicht.

Für die aufgeführten Zeitwerte ist zu beachten, dass die ausgeführten Simulationen allesamt nichtdeterministisch und zudem von unkontrollierbaren äußeren Umständen abhängig sind (z.B. Betriebssystem-Scheduler, Java-Garbage-Collector, Netzwerk des Hostsystems), dass die marginalen Abweichungen in den Rechenzeiten auch bei mehreren Durchläufen und langer Laufzeit nicht als ausschlaggebend hingenommen werden können, um einen *besten* Algorithmus zu finden. Vor allem die vielen bei jedem Teilschritt nötigen Netzwerkzugriffe sind dafür ein zu großer Unsicherheitsfakor. Deshalb werden im Folgenden primär die Kennzahlen betrachtet, die hauptsächlich für die unterschiedlichen Rechenzeiten der Algorithmen verantwortlich sind.

6.3.1. Test 1 - Einzelgruppen

Test 1 stellte mit mehreren sehr kleinen separaten Gruppen an die Algorithmen direkt eine besondere Herausforderung. In Abb. 6.5(a) sieht man, dass die beiden positionsbasierten Algorithmen KMeans und QHull die Gruppen im wesentlichen nicht erkannt haben und dementsprechend einzelne Gruppen auf zwei Server verteilt haben. Das führt zu deutlich höheren Kommunikationskosten als bei den anderen Algorithmen, die diese Kosten direkt mit in die Rechnung einbeziehen. Der einzige Algorithmus, der die Gruppen weitgehend gut erkennt, ist der Peschlow-Algorithmus. Nach einer kurzen Einschwingphase konnte er die Kommunikationskosten stets auf einem Minimum halten.



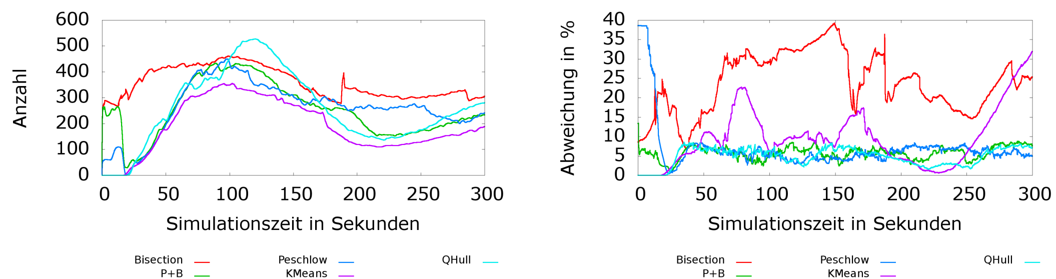
(a) Anzahl der von anderen Plattformen erhaltenen Update-Objekte

(b) Mittlere Abweichung der Agentenzahl pro Server vom Gesamtdurchschnitt

	KMeans	QHull	Peschlow	Bisection	P+B
Anzahl Migrationen	3743	68214	2646	36617	10540
Update-Objekte Ø	17	61	8	99	24
abonnierte Sektoren Ø	1	4	5	8	5
Migrationsschrittdauer (ms) Ø	12	12	11	16	12
Sync-Schrittdauer (ms) Ø	28	45	31	74	85
Gesamt-Schrittdauer (ms)[1] Ø	40	57	42	90	97

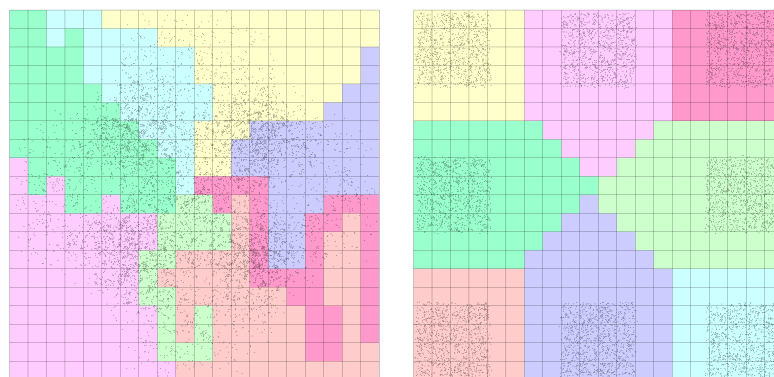
Abbildung 6.5.: Testergebnisse für Test 1

Die gleichmäßige Verteilung der Agenten auf die acht Server (siehe Abb. 6.5(b)) konnte deshalb ebenfalls vom Peschlow-Algorithmus am besten gelöst werden, da die Gruppen alle erkannt wurden. Die Algorithmen KMeans und QHull schnitten auch hier sehr schlecht ab. Jedes Mal, wenn sich zwei Gruppen während der Simulation sehr nahe kamen, wurden sie auf den gleichen Server migriert und dementsprechend führte dieser Server deutlich mehr Agenten aus als die anderen Server. Die mittlere Abweichung von der durchschnittlichen Agentenzahl hält sich mit 30 Prozent noch in Grenzen, jedoch verwaltet ein Server mit zwei Gruppen für diesen Moment mehr als doppelt so viele Agenten wie die anderen Server, sodass der Agentenschritt entsprechend länger dauert.
In der Tabelle sieht man zudem, dass die Anzahl der Migrationen beim QHull- und beim Bisection-Algorithmus um ein vielfaches höher liegen als bei den anderen Algorithmen. Innerhalb der zehn Minuten der Simulation migrierte der Bisection-Algorithmus jeden einzelnen der 400 Agenten im Schnitt 99 mal, QHull sogar 170 mal.



(a) Anzahl der von anderen Plattformen erhaltenen Update-Objekte

(b) Mittlere Abweichung der Agentenzahl pro Server vom Gesamtdurchschnitt



(c) Screenshot einer Simulation mit Bisection-Partitionierung

(d) Screenshot einer Simulation mit Peschlow-Partitionierung

Abbildung 6.6.: Testergebnisse und Beispielpartitionierungen für Test 2

6.3.2. Test 2 - Chaotische Menge

	KMeans	QHull	Peschlow	Bisection	P+B
Anzahl Migrationen	57815	58720	89634	79673	96569
Update-Objekte Ø	1487	1833	2065	2693	2234
abonnierte Sektoren Ø	16	19	24	30	23
Agentenschrittdauer (ms) Ø	27	29	37	43	37
Migrationsschrittdauer (ms) Ø	24	12	27	16	33
Sync-Schrittdauer (ms) Ø	77	79	83	83	84
Gesamt-Schrittdauer (ms) Ø	128	140	147	142	154

Der Test 2 zeigt sehr gut die Vor- und Nachteile des KMeans-Algorithmus. In Abb 6.6(a) ist zu sehen, dass QHull und Peschlow, welche beide KMeans für die initiale Partitionierung benutzen, zu Beginn keinerlei Kommunikationskosten haben, weil KMeans die acht Startgruppen er-

kannt hat und die Gruppen sich gegenseitig nicht sehen. Der eigentliche KMeans-Algorithmus hat diese jedoch nicht erkannt und die Gruppen teilweise gespalten, weshalb dort geringe Kommunikationskosten auftreten. Die beiden auf Bisection basierenden Algorithmen erkennen keinerlei Gruppen und haben dementsprechend höhere Kommunikationskosten.

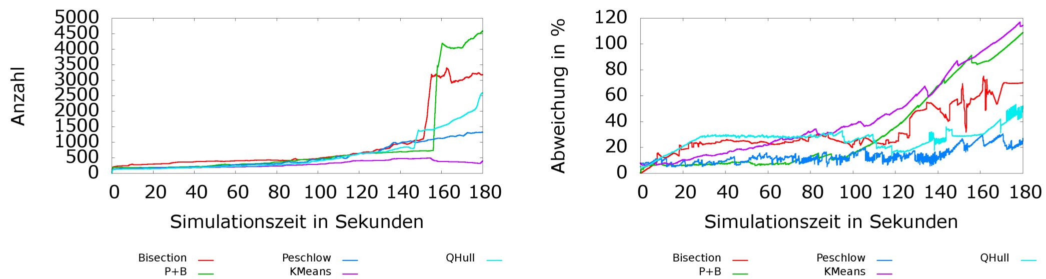
Die falsch erkannten Gruppen des KMeans Algorithmus verursachen zu Beginn ein starkes Ungleichgewicht bei der Agentenverteilung auf die einzelnen Server, welches sich im Laufe der Simulation reduziert und zum Ende hin wieder stark ansteigt. Der Peschlow-Algorithmus profitiert nur kurzfristig von der idealen Anfangsverteilung und pendelt sich letztlich bei einer Abweichung von 15 Prozent ein. Der Bisection-Algorithmus erzeugt eine im Vergleich extrem schlechte Verteilung, bei welcher sowohl die Agentenverteilung als auch die Kommunikationskosten teils deutlich schlechter sind als bei allen anderen Algorithmen.

Die beiden Screenshots stellen zwei markante Partitionierungen dar. Screenshot 6.6(c) zeigt die Partitionierung via Bisection zu einem fortgeschrittenen Zeitpunkt. Es ist sehr gut zu erkennen, dass mehrere Partitionen eine sehr verschlungene Form angenommen haben. Im Verhältnis zu den selbst kontrollierten Sektoren ist die Anzahl der abonnierten Nachbarsektoren damit extrem hoch.
Screenshot 6.6(d) stellt die initiale Partitionierung des Peschlow-Algorithmus dar. Hier sind alle acht Gruppen auf einem separaten Server und durch die Entfernung der Agenten zueinander ist keinerlei Kommunikation nötig.

6.3.3. Test 3 - Evakuierung mit einem Notausgang

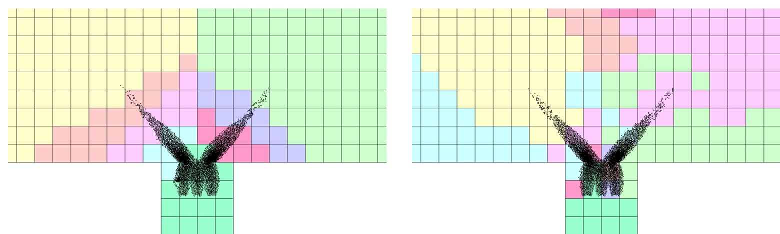
	KMeans	QHull	Peschlow	Bisection	P+B
Anzahl Migrationen	50677	75307	136480	149848	264607
Update-Objekte Ø	2281	4840	4309	5066	4151
abonnierte Sektoren Ø	14	16	18	24	21
Agentenschrittdauer (ms) Ø	63	57	75	78	85
Migrationsschrittdauer (ms) Ø	26	33	40	40	47
Sync-Schrittdauer (ms) Ø	132	132	142	180	271
Gesamt-Schrittdauer (ms) Ø	221	222	257	298	403

In Test 3 wurde untersucht, wie sich die Algorithmen in einer sich langsam verdichtenden Agentenmenge verhalten.
Zu Beginn der Simulation können alle Algorithmen den Kommunikationsaufwand (siehe Abb. 6.7(a)) recht gering halten. Dieser steigt mit steigender Agentendichte stetig an. Dabei kann KMeans die besten Ergebnisse erzielen, während bei sich QHull der Aufwand zum Ende der Simulation stetig erhöht. Bei den beiden auf Bisection basierenden Algorithmen steigen die Kommunikationskosten gegen Ende der Simulation sprunghaft auf ein Vielfaches an.



(a) Anzahl der von anderen Plattformen erhaltenen Update-Objekte

(b) Mittlere Abweichung der Agentenzahl pro Server vom Gesamtdurchschnitt



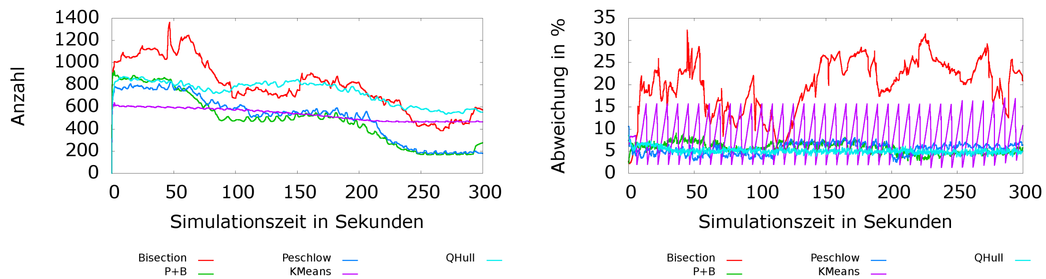
(c) Screenshot einer Simulation mit Partitionierung

(d) Screenshot einer Simulation mit Partitionierung

Abbildung 6.7.: Testergebnisse und Beispielpartitionierungen für Test 3

In Abb. 6.7(b) ist zu sehen, dass sich die Agentenverteilungen sehr unterschiedlich verhalten. Obwohl alle Algorithmen initial eine sehr ausgewogene Verteilung erzeugen, verschlechtert sich diese vor allem bei Bisection und QHull sehr schnell und bleibt anschließend auf einem konstant schlechten Level. Beim KMeans-Algorithmus verschlechtert sich die Verteilung im Laufe der Simulation immer weiter. Der Server, der die Sektoren um den Ausgang verwaltet, führt am Ende beinahe alle Agenten alleine aus, während die anderen Server teilweise fast gar keine Agenten mehr ausführen. Diese Situation lässt sich sehr gut im Screenshot 6.7(c) erkennen.

Je verdichteter die restlichen Agenten am Ende der Simulation sind, desto häufiger werden die Agenten bei den beiden Bisection-Algorithmen migriert. Teilweise werden bei jeder Partitionierung 30 Prozent der Agenten auf andere Server migriert, während z.B. bei KMeans kaum Migrationen stattfinden.
Zeitweise entstanden dabei bizarre Partitionierungen. Der Screenshot 6.7(d) stellt die Partitionierung zum Ende der Simulation mit dem Peschlow-Algorithmus dar. Es ist zu sehen, dass einige Partitionen nur noch wenige Sektoren umfassen, in denen sich auch Agenten befinden. Sehr auffällig ist jedoch, dass die Partitionen nicht mehr zusammenhängend sind, sondern die



(a) Anzahl der von anderen Plattformen erhaltenen Update-Objekte

(b) Mittlere Abweichung der Agentenzahl pro Server vom Gesamtdurchschnitt

Abbildung 6.8.: Testergebnisse für Test 4

Verteilung im Zentrum sehr durcheinander gewürfelt wirkt. Dies hat natürlich einen immens hohen Kommunikationsaufwand zur Folge.

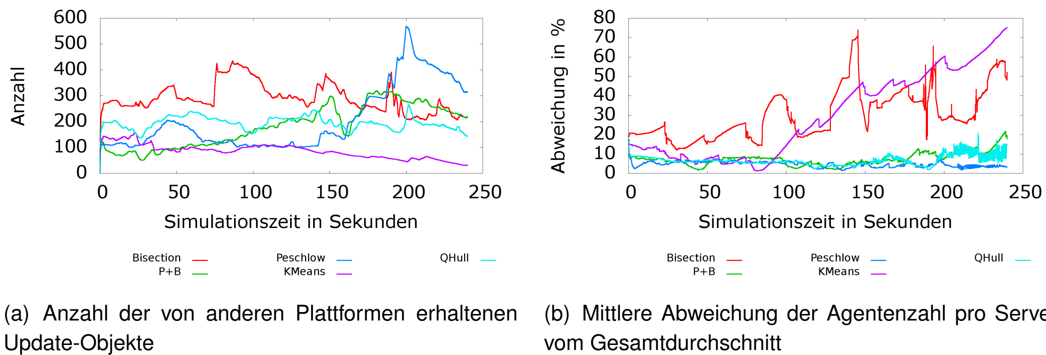
6.3.4. Test 4 - Homogene Menge

	KMeans	QHull	Peschlow	Bisection	P+B
Anzahl Migrationen	146466	235456	324944	395104	243206
Update-Objekte Ø	4250	5322	4288	6080	4365
abonnierte Sektoren Ø	17	18	14	21	14
Agentenschrittdauer (ms) Ø	88	95	120	133	119
Migrationsschrittdauer (ms) Ø	31	40	42	50	38
Sync-Schrittdauer (ms) Ø	180	175	160	177	162
Gesamt-Schrittdauer (ms) Ø	299	310	322	360	319

In Test 4 wurde untersucht, wie die Algorithmen eine große Gruppe von Agenten partitionieren, in der sich die Agenten mit gleichförmiger Geschwindigkeit parallel zueinander einen Gang entlang bewegen.

Bei den Kommunikationskosten befinden sich alle Algorithmen lange Zeit auf einem annähernd gleichen Niveau (siehe Abb. 6.8(a)), wobei auch hier der Bisection-Algorithmus höhere Kosten verursacht als die anderen Algorithmen. Zum Ende der Simulation hin können die drei Algorithmen, die Kommunikationskosten optimieren, diese signifikant reduzieren, während KMeans und QHull auf einem konstanten Level bleiben.

Bei der Verteilung der Agenten schaffen es sowohl der Peschlow, QHull als auch der B+P-Algorithmus die mittlere Abweichung von der idealen Agentenanzahl bei durchschnittlich unter



(a) Anzahl der von anderen Plattformen erhaltenen Update-Objekte

(b) Mittlere Abweichung der Agentenzahl pro Server vom Gesamtdurchschnitt

Abbildung 6.9.: Testergebnisse für Test 5

sieben Prozent zu halten. Beim Bisection-Algorithmus steigt diese Abweichung kurz nach Simulationsbeginn drastisch an und bleibt auf einem hohen Level.
Sehr auffällig ist, dass KMeans im Schnitt ebenso bei zehn Prozent liegt, um diesen Wert jedoch periodisch stark schwankt.

Die perfekte Partitionierung, eine Teilung der Umwelt in acht parallele Streifen, welche Agentenmigrationen komplett unnötig machen, hat keiner der Algorithmen erreicht.

6.3.5. Test 5 - Bahnhof

	KMeans	QHull	Peschlow	Bisection	P+B
Anzahl Migrationen	28643	130579	61085	41823	23270
Update-Objekte Ø	1380	3442	1956	1557	1534
abonnierte Sektoren Ø	2	5	2	3	3
Agentenschrittdauer (ms) Ø	174	166	287	248	218
Migrationsschrittdauer (ms) Ø	19	40	29	21	17
Sync-Schrittdauer (ms) Ø	38	50	43	51	83
Gesamt-Schrittdauer (ms) Ø	231	250	359	320	318

Der Bahnhof stellt die einzige Umgebung dar, welche große Hindernisse aufweist und nur in schmale Gänge geteilt ist.

Durch die geringen Sektorverbindungen sind die Kommunikationskosten insgesamt relativ niedrig. Bis auf QHull können alle Algorithmen die Kosten auf einem geringen Niveau halten. Zum Ende hin, wo sich die Agenten an den vier Ausgängen sammeln, steigen die Kommunikationskosten beim Peschlow-Algorithmus jedoch ebenfalls stark an.

Bei der Agentenverteilung können nur QHull und Peschlow gute Ergebnisse erzeugen. Die restlichen drei Algorithmen weisen eine extrem schlechte Verteilung auf. In der Detailbetrachtung ist aufgefallen, dass bei diesen Algorithmen jeweils ein Server für jeden Ausgang zuständig ist und bei der hohen Agentendichte viel zu tun hat, während die anderen vier Server die weniger frequentierten bzw. bereits leeren Gänge verwalten und so annähernd keine Agenten ausführen.
Die beiden Algorithmen QHull und Peschlow können bei der Agentenverteilung zwar punkten, jedoch steigt vor allem zum Ende hin die Zahl der durchgeführten Migrationen sehr stark an und liegt vor allem bei QHull ein Vielfaches über dem Aufwand, den die drei Algorithmen mit der schlechteren Verteilung verursachen.

6.3.6. Zusammenfassung

An den Tests sind mehrere zentrale Punkte zu erkennen.
Die Anzahl der Migrationen variierte teils stark von Algorithmus zu Algorithmus. Der positionsbasierte Algorithmus KMeans wies fast immer die niedrigste Anzahl an Agentenmigrationen auf, während vor allem der QHull- und der Bisection-Algorithmus dazu tendierte, einen Sektor immer wieder zwischen zwei Servern hin und her zu schieben.

Je konzentrierter sich die Agenten in einem kleinen Bereich befanden, desto schwieriger wurde es für die Algorithmen, eine gute und vor allem stabile Partitionierung zu erreichen. KMeans wies eklatante Schwächen auf, wenn sich die Agentenmengen drastisch verdichteten (wie in den Tests 3 und 5). Vor allem QHull und Bisection taten sich schwer damit, eine stabile Verteilung zu erzeugen und verrannten sich häufig in alternierenden Partitionierungen, bei der ein Sektor immer wieder zwischen zwei Servern hin und her geschoben wurde und so die Migrationen einen erheblichen Teil des Simulationsschrittes ausmachten.

7. Diskussion

7.1. Synchronisationsperformance

Vergleicht man die im vorigen Kapitel dargestellten Ergebnisse der vollständigen und der partiellen Synchronisation miteinander, wird schnell deutlich, dass die partielle Synchronisation signifikante Vorteile gegenüber der vollständigen Synchronisation hat. Ausschlaggebend ist dabei hauptsächlich die zu übertragende Datenmenge. Vor allem bei großen Simulationen, welche letztlich mit Hilfe von WalkSim ausgeführt werden sollen, wiegt dieser Nachteil sehr schwer. Bei einer Simulation mit 300.000 Agenten auf acht Servern muss jeder Server ein Paket von ca. 2-3 MB an jeden einzelnen anderen Server senden. Das resultierende Gesamtpaket von über 110 MB, die in jedem Schritt übertragen werden müssen, benötigt selbst bei moderner Netzwerkinfrastruktur viel Zeit, bis alle Daten vollständig übertragen werden. Zusätzlich muss der Sender die Daten vor der Übertragung serialisieren und der Empfänger deserialisieren und in dessen Datenbank einpflegen.
Einen großen Anteil an diesem Kommunikationskostenunterschied hat das speziell darauf zugeschnittene Kommunikationssystem. Ein klassischer, zentralisierter Messagebus wie ActiveMQ hätte die zu übertragene Datenmenge bei der partiellen Synchronisation nicht derart stark senken können. Dort hätte für jeden Server das Datenpaket von 2-3 MB Größe zuerst immer an den zentralen Message-Broker gesendet werden müssen, obwohl für viele Pakete gar kein Empfänger existiert.

Ein weiteres Problem ist, dass die Datenmenge, die ein Server bei der vollständigen Synchronisation übertragen muss, bei steigender Serverzahl linear ansteigt, da die eigenen Updates an jeden anderen Server übertragen werden müssen. Bei der partiellen Synchronisation verändert sich für einige Server im Idealfall gar nichts, wenn ein weiterer Server hinzugefügt wird, der nicht in dessen Nachbarschaftsbereich liegt. Während die Synchronisationsdauer bei der partiellen Synchronisation mit steigender Serverzahl annähernd konstant bleibt, steigt diese bei der vollständigen Synchronisation merklich an, wenn auch nicht linear.

Auf den Graphen (c) und (d) der Abbildung 6.1 ist zu sehen, dass die Netzwerklast mit fortschreitender Simulation abfällt und die Dauer des gesamten Simulationsschrittes ansteigt. Dieser Anstieg ist darauf zurückzuführen, dass sich die Agentenmenge während der Simulation in Richtung Raummitte hin bewegt und entsprechend verdichtet. Eine dichtere Agentenmenge

führt dazu, dass ein Agent mehr andere Agenten in seinem Sichtbereich hat und dementsprechend auch mehr rechnen muss. Dadurch, dass so die Gesamtrechenzeit für einen Takt höher wird, die zu übertragende Datenmenge jedoch gleich bleibt, verringert sich letztlich die Datenrate ein wenig.

Es ist zu betonen, dass WalkSim von Grund auf für die partielle Synchronisation optimiert und die vollständige Synchronisation nur nachträglich mit wenig Aufwand implementiert wurde. An einigen Stellen im System finden sich folglich Programmteile, die für die vollständige Synchronisation nicht benötigt werden und so unnötige Last erzeugen. In den Ergebnissen ist jedoch zu sehen, dass der größte Faktor die zu übertragende Datenmenge ist und hier die Serialisierung und Übertragung die meiste Zeit in Anspruch nimmt.
Optimiert man WalkSim für die vollständige Synchronisation, würde sich die Performance sicherlich noch steigern lassen. Die Unterschiede der beiden Methoden sind jedoch so extrem, dass es fraglich ist, ob die Ergebnisse dadurch signifikant beeinflusst werden.

7.2. Skalierbarkeit

7.2.1. Skalierung bei konstanter Agentenmenge

Die Skalierbarkeit von WalkSim liegt weitgehend im erwarteten Bereich. Bei gleichbleibender Agentenzahl führt eine Vergrößerung des Serververbundes dazu, dass die Agentenlast besser aufgeteilt werden kann. Diese Verteilung hat bei großen Serveranzahlen jedoch ihre Grenzen und ist sehr an den Partitionierungsalgorithmus gebunden. Beim ersten Skalierungstest ist aufgefallen, dass der verwendete Peschlow-Algorithmus die Agenten nicht gleichmäßig genug verteilen konnte. Bei 16 Servern und 80.000 Agenten führte so einer der Server längere Zeit über 8.000 Agenten aus, obwohl es idealerweise nur 5.000 sein sollten. Dadurch dass dieser eine Server nun mehr Zeit benötigt, dauert auch der gesamte Agentenschritt länger. Eine höhere Anzahl von Servern führt daher nicht immer zu besserer Leistung. Je mehr Partitionen es gibt, desto eher könnte eine dieser Partitionen ein Ungleichgewicht aufweisen, welches die Gesamtperformance beeinträchtigt. Dies spiegelt sich auch in der CPU-Auslastung wider. Je mehr Server an einer Simulation beteiligt sind, desto mehr sinkt die durchschnittliche CPU-Auslastung des Gesamtsystems. Die durchschnittlich höchste CPU-Last eines einzelnen Servers liegt dafür über die gesamte Simulation hinweg mit über 80 Prozent deutlich höher.
Ein weiterer Faktor bei der Auslastung des Gesamtsystems sind externe Faktoren wie der Scheduler des Betriebssystems, die Netzwerkinfrastruktur oder der Garbage-Collector der JVM. Vor allem der Garbage-Collector (GC) hat einen nicht zu unterschätzenden Einfluss auf das verteilte System. Da die Ausführung des GC in Java nur bedingt kontrolliert werden kann, führt jeder Host unabhängig von den anderen Hosts seinen GC aus und hält für diese Zeit die gesamte Simulation auf. Je mehr GCs insgesamt beteiligt sind, desto größer ist der Effekt. Vor

allem bei zu schwachen Hostsystemen mit zu wenig RAM muss der GC zu häufig ausgeführt werden, sodass dieser systemweit teilweise 10 Prozent der gesamten CPU-Last ausmachte.

Die Synchronisationsleistung blieb für jede Serveranzahl auf einem ungefähr gleichen Niveau. Das liegt vor allem daran, dass die zu übertragene Datenmenge pro Server annähernd konstant bleibt.
Hier ist jedoch zu betonen, dass das verwendete Szenario genau für diesen Fall gedacht ist. Ein langer schmaler Gang wird bei starker Partitionierung so geteilt, dass die einzelnen Partitionen wie aufgereiht aussehen und jede Partition maximal zwei andere Partitionen als Nachbarn hat und sich die Anzahl der zu synchronisierenden Rand-Sektoren pro Server auch bei steigender Serveranzahl nicht ändern. In einem quadratischen Raum können Partitionen auch mitten im Raum liegen und so rundum diverse Nachbarpartitionen aufweisen, was den Synchronisationsaufwand entsprechend erhöhen könnte.

Ein wichtiger externer Faktor für die Synchronisationsleistung ist zudem die Netzwerkinfrastruktur. So beeinflusst nicht nur die verfügbare Bandbreite und die Ping-Zeiten die Dauer der Synchronisation. In großen Netzwerkstrukturen spielt die Topologie und die Leistung der verwendeten Switche eine nicht zu unterschätzende Rolle. Schwierig ist dies vor allem dadurch, dass man den Serververbund nicht in Gruppen unterteilen kann, sondern potenziell jeder Server zu jedem anderen Server eine gute Verbindung benötigt.

7.2.2. Skalierung bei linear steigender Simulationsgröße

Beim zweiten Skalierungstest stiegen sowohl die Agentenschrittdauer als auch die Synchronisationsdauer mit steigender Serverzahl und Simulationsgröße leicht an.
Der Anstieg der Agentenschrittdauer ist auf den gleichen Effekt zurückzuführen, wie beim vorigen Test. Je mehr Partitionen es gibt, desto eher weicht eine in ihrer Größe von der mittleren Anzahl ab und treibt die Rechenzeit in die Höhe.

Der Anstieg der Synchronisationszeit ist mit zwei Punkten zu erklären:
Zum einen steigt mit steigender Serveranzahl auch die absolut zu synchronisierende Datenmenge an, da es mehr Randsektoren gibt. Zum anderen steigt jedoch auch die Anzahl der zu verwaltenden Sektoren linear zur Anzahl der Server an und auch wenn das optimierte Kommunikationssystem viele Sektor-Nachrichten gar nicht erst versendet, so müssen diese trotzdem von der jeweiligen DVE-Instanz aufbereitet und dem WalkCS übergeben werden. Sowohl die Aufbereitung der Daten als auch die Prüfung, ob diese versendet werden müssen, kostet ebenfalls Zeit.

7.2.3. Zusammenfassung

Die Skalierung der WalkSim-Plattform befindet sich bereits auf einem ordentlichen Niveau. Die Performance skaliert annähernd linear mit der Anzahl der Server, wobei aufgrund der Notwendigkeit von Netzwerkkommunikation eine wirklich lineare Skalierung für diese Art von Problem praktisch nicht erreichbar ist.

Eine Schwachstelle für die Skalierung ist die getaktete Ausführung einer Simulation. Dadurch, dass es für die einzelnen Server immer wieder Synchronisierungspunkte gibt, an denen die schnelleren Server auf die langsamen Server warten müssen und eine perfekte Verteilung langfristig nur schwer möglich ist, sind höhere Auslastungsquoten nur schwer zu erreichen. Im Gegenteil wird die rechentechnische Auslastung der einzelnen Server potentiell mit steigender Anzahl der Server weiter abnehmen. Durch die Taktung gibt immer der langsamste Server die Geschwindigkeit vor und bei steigender Anzahl an Servern steigt auch die Wahrscheinlichkeit von temporären Ausreißern durch äußere Umstände wie CPU-Scheduling, Netzwerkfehler oder Garbage-Collection. Vor allem letztere hat bei großen Netzwerken aus zu schwachen Hosts einen erheblichen Einfluss auf die Skalierbarkeit.

Das Verhältnis der Zeit, die für die Synchronisation aufgewendet wird gegenüber der Zeit, die für die Agenten aufgewendet wird, ist für die hier durchgeführten Tests sehr ungünstig. Wie in Kapitel 4.4.1 bereits beschrieben, ist die hier verwendete KI der Agenten eine extrem simple KI. Für reale Szenarien wird eine deutlich komplexere KI benötigt, die dementsprechend auch deutlich mehr Leistung braucht. Aus diesem Grund ist sowohl das Verhältnis der einzelnen Teilschrittzeiten als auch die Gesamtsumme der ausführbaren Agenten pro Server mit Vorsicht zu genießen. Für realistische Szenarien muss davon ausgegangen werden, dass die Anzahl der ausführbaren Agenten deutlich geringer sein wird.

7.3. Partitionierung

Für eine Analyse der Partitionierungsalgorithmen ist es wichtig, zu Beginn festzuhalten, wann ein Algorithmus gut oder schlecht ist und was ein Algorithmus auf jeden Fall vermeiden sollte.

Die eigentliche Aufgabe eines Verteilungs- bzw. Partitionierungsalgorithmus in einem verteilten System ist die schnellstmögliche Abarbeitung der anfallenden Arbeitspakete unter Berücksichtigung der verfügbaren Ressourcen. Für WalkSim kann diese Aufgabe in drei Teilaufgaben geteilt werden:

- Die Dauer des Agentenschrittes t_A muss minimal sein. Das wird erreicht, indem alle Server möglichst gleich ausgelastet sind. Da in den hier betrachteten Testfällen alle Agenten annähernd gleiche Last erzeugen, kann *Last* mit der Anzahl der Agenten gleichgesetzt

werden. Zu beachten ist, dass der Agentenrechenschritt so lange dauert, bis auch der letzte Server fertig ist. Eine mittlere gute Verteilung ist wertlos, wenn es einen Ausreißer gibt, der doppelt so lange rechnen muss.

- Die Dauer der Synchronisierung t_S muss minimal sein. Das ist zu erreichen, indem möglichst wenig Nachrichten und wenig Objekte ausgetauscht werden müssen. Da in WalkSim pro Sektor eine Nachricht verschickt wird, muss hier vor allem die Anzahl der abonnierten Sektoren minimiert werden. Bei sehr vollen Sektoren spielt jedoch auch die Anzahl der Objekte eine Rolle.
- Die Zeit für Migrationen t_M muss minimal sein. Wird ein Sektor von einer DVE-Instanz auf eine andere verschoben, werden nicht nur alle Agenten des Sektors migriert, sondern auch alle Daten des Sektors auf die neue Plattform geladen. Das wird in den meisten Fällen damit erreicht, indem die Anzahl der Sektormigrationen minimal gehalten wird.

Aus dieser Liste kann nun eine allgemeine Formel erstellt werden, welche minimiert werden muss:

$$t_G = t_A + t_S + t_M$$

In den meisten Fällen, die in der folgenden Diskussion betrachtet werden, wird es jedoch nicht passend sein, stupide diese Formel anzuwenden um den besten Algorithmus zu sehen. Zum einen sind die gemessenen Zeiten aufgrund äußerer Einflüsse und oft nur marginaler Differenzen nicht signifikant genug, um realistische Schlüsse zu ziehen. Zum anderen konnten aus logistischen Gründen nur verhältnismäßig kleine Simulationen durchgeführt werden.
Im Folgenden wird daher hauptsächlich Bezug auf die oben beschriebenen Kennzahlen genommen, welche die Zeitwerte hauptsächlich beeinflussen.

Für die Einordnung müssen diese einzelnen Werte gewichtet werden, da ein versendetes Update-Objekt mehr oder weniger nicht gleich viel Unterschied ausmacht, wie ein Agent mehr oder weniger auf einer Plattform. In realistischen Szenarien wird WalkSim Agenten ausführen, die nicht nur deutlich mehr Last erzeugen, als die hier verwendeten, sondern auch einen weitaus größeren internen Zustand besitzen, welcher bei einer Migration ebenfalls übertragen werden muss. Im Gegenzug wird sich die Menge der ausgeführten Aktionen pro Agent höchstwahrscheinlich kaum erhöhen. Aus diesem Grund ist es für einen Partitionierungsalgorithmus wichtiger, einen guten und vor allem stabilen Lastausgleich zu schaffen, selbst wenn dies einen höheren Synchronisationsaufwand bedeutet.

7.3.1. KMeans

Der KMeans-Algorithmus ist ein sehr simpler Algorithmus, welcher in seiner Funktionsweise aus dem Bereich des Datamining oft schon geläufig ist. Er betrachtet für die Partitionierung

lediglich die Position des Agenten und nicht, wer mit wem kommuniziert. Das macht ihn zu einem schnellen Algorithmus, da zum einen bei der Partitionierung wenig Rechenaufwand nötig ist und zum anderen im Vorfeld keine Daten über Kommunikationsstrukturen gesammelt werden müssen.

Während der Berechnung der Partitionierung wird versucht, die summierte Entfernung der Agenten bzw. hier der Sektoren zum Zentrum der Partition möglichst gering zu halten. Das resultiert darin, dass die Partitionen generell eher kreisförmig sind und entsprechend durch den daraus resultierenden geringen Umfang eine minimale Anzahl Nachbarsektoren haben.
Durch diese Kreisform werden bei KMeans in fast allen Tests nur wenige Fremdsektoren abonniert und entsprechend wenig Daten ausgetauscht, was zu einem sehr niedrigen Kommunikationsaufwand führt.

Die Tatsache, dass der hier verwendete KMeans-Algorithmus die Summe aller Entfernungen der Agenten zum Partitionszentrum als Gesamtkosten der Partition betrachtet ist dabei zugleich der größte Nachteil. Dadurch kann eine Partition mit vielen dicht beieinander stehenden Agenten die gleichen Gesamtkosten besitzen wie eine Partition, bei der wenige Agenten aber weiter voneinander entfernt sind. Dieser Effekt ist vor allem bei Test 3 und 5 zu beobachten, wo einige wenige Server am Ende fast alle Agenten verwalten und der Rest nichts zu tun hat.
Ein weiterer Faktor für das Ungleichgewicht ist die initiale zufällige Verteilung. KMeans ist extrem davon abhängig, wo die ersten Partitionszentren platziert werden, was vor allem bei den acht Gruppen in Test 2 deutlich wurde. Eine zu schlechte Auswahl an dieser Stelle kann der Algorithmus nachträglich nicht mehr ausgleichen.

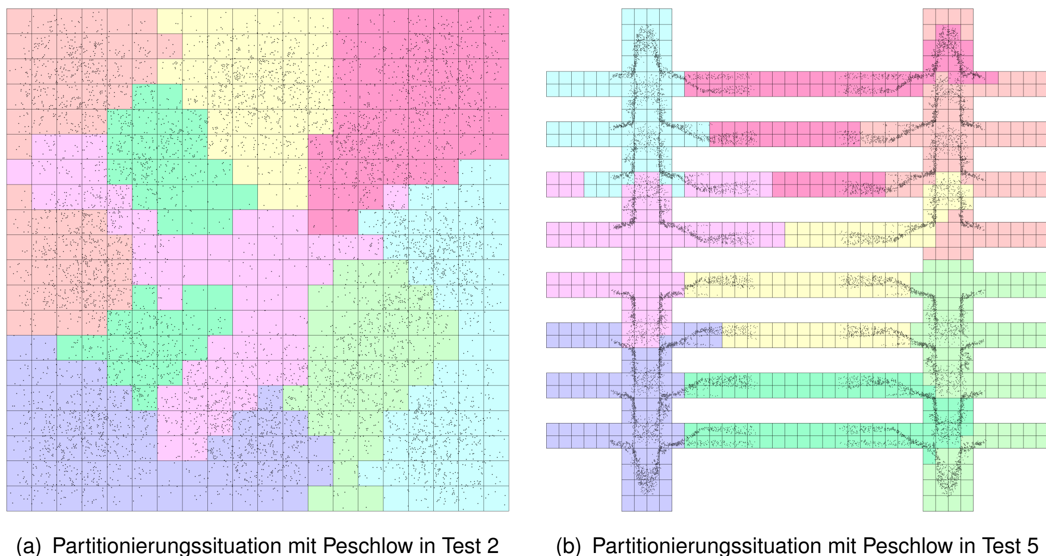
7.3.2. QHull

QHull soll KMeans um den fehlenden Lastausgleich erweitern und ist damit auch recht erfolgreich, sodass die Agentenverteilung bei allen fünf Testfällen besser gelang als bei KMeans. Dafür stiegen die Kommunikationskosten zwischen den Plattformen deutlich gegenüber KMeans an.
Dies ist vor allem darauf zurückzuführen, dass QHull anscheinend nicht gut mit den Sektoren umgehen kann. Zum einen war es QHull nicht möglich, die konvexe Hülle beizubehalten, die im Algorithmus eigentlich gefordert war. Das lag zum Teil daran, dass bei einer Unterlastung Nachbarpartitionen bevorzugt wurden, die mehr Agenten ausführten als die anderen Nachbarn. So entstanden teilweise leicht verschlungene Partitionsformen, welche entsprechend mehr Nachbarsektoren besaßen als die runden Partitionen von KMeans.
Zum anderen konnte QHull keine stabile Verteilung erreichen, sobald die Anzahl der Sektoren, auf die sich die Agenten verteilten, zu gering wurde. Die Anzahl der durchgeführten Migrationen war bei QHull nicht nur deutlich höher als gegenüber KMeans, sondern teilweise auch höher als bei den restlichen Algorithmen. Vor allem in Test 3 migrierte der Algorithmus gegen

Ende kontinuierlich Sektoren zwischen den Partitionen hin und her, um eine ausgeglichene Agentenverteilung zu schaffen, was aufgrund der stark verdichteten Agentenmenge am Ausgang kaum möglich war.

Das Fehlen einer Kommunikationskostenoptimierung ist vor allem in Test 1 gut zu beobachten. Der initiale KMeans erkannte die acht Gruppen, was zu Beginn zu niedrigen Kommunikationskosten führte. Im Laufe der Simulation wanderten die Agenten jedoch durch ihre Fortbewegung in andere Partitionen und stellten ein Lastungleichgewicht her. Dieses Ungleichgewicht wurde mit unzähligen Migrationen ausgeglichen. Weil dabei auf keinerlei Kommunikationsstruktur geachtet wurde, stiegen die Kommunikationskosten sehr stark an.

7.3.3. Peschlow



(a) Partitionierungssituation mit Peschlow in Test 2 (b) Partitionierungssituation mit Peschlow in Test 5

Abbildung 7.1.: Beispielpartitionierungen mit Peschlow

Der Peschlow-Algorithmus ist der erste Algorithmus, welcher neben der CPU-Last auch das Kommunikationsaufkommen aktiv in seine Rechnung mit einbezieht. Dadurch bleiben sowohl die Lastverteilung als auch das Kommunikationsaufkommen meist auf einem mittleren Niveau.

Die originale Beschreibung des Algorithmus interpretierte *Agentenlast* als die Zeit, die ein Server benötigt, ein einzelnes Agentenereignis zu bearbeiten. Diese Zeit zu messen ist jedoch sehr schwierig, bzw. das Messergebnis ist sehr unsicher. Da die in dieser Arbeit entwickelten Agenten alle eine ähnliche Last erzeugen, wurde die Agentenanzahl mit der Agentenlast

gleichgesetzt. Werden in Zukunft andere Agenten implementiert, muss diese Festlegung evtl. überdacht und eine neue Lösung gefunden werden.

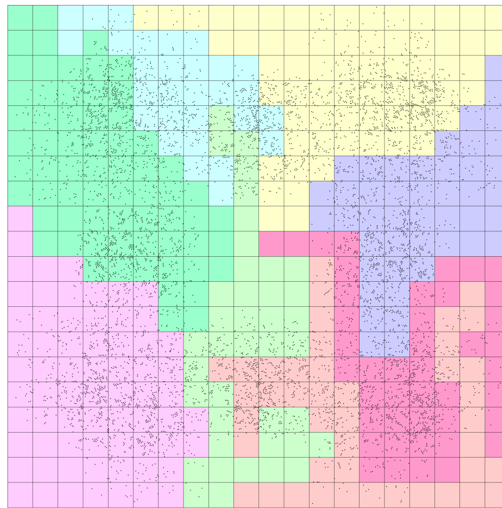
Die Teilung des Algorithmus in zwei separat ausgeführte Teilschritte verursachte in mehreren Fällen eine hohe Migrationsrate, da die Sektoren, die zuvor durch die Kommunikationsoptimierung migriert wurden, in der nächsten Lastoptimierung wieder zurück migriert wurden. Optimaler wäre es, wenn beide Schritte intern nacheinander berechnet werden, ohne dass zwischendurch eine Migration durchgeführt wird.
Ebenfalls eine hohe Migration wurde durch Sektoren verursacht, die zu zwei Partitionen ein sehr ähnliches Kommunikationsaufkommen hatten. Hier konnte oft beobachtet werden, dass diese Sektoren zwischen den Servern pendelten. Dies könnte verhindert werden, indem ein Grenzwert eingeführt wird, ab der ein Sektor zum anderen Server migriert wird.

In den Tests ist es mehrfach vorgekommen, dass eine Partition nach einer gewissen Zeit nicht mehr zusammenhängend war, sondern es sich zwei Teilpartitionen gebildet haben, die getrennt voneinander waren (siehe Abb. 7.1(a)). In den beiden vorigen Algorithmen wurde dies dadurch effektiv verhindert, indem die Distanz zur Partition in eine Migrationseintscheidung einbezogen wurde. Dies entfällt beim Peschlow-Algorithmus. Das führt dazu, dass in solchen Fällen deutlich mehr Sektoren abonniert werden müssen, als bei anderen Algorithmen.
In den meisten Fällen konnte sich die kleinere Teilpartition jedoch nicht lange halten und wurde während der Kommunikationskostenoptimierung zu anderen Nachbarpartitionen migriert.
Beim Test 5 war es allerdings unvermeidbar, dass sich geteilte Partitionen gebildet haben, da der KMeans-Algorithmus, welcher für die initiale Partitionierung verwendet wird, nicht darauf achtet, ob Sektoren innerhalb einer Partition einen vollständigen Spanning-Tree bilden können oder nicht (siehe Abb. 7.1(b)).

Der Peschlow-Algorithmus betrachtet in der Kommunikationsoptimierung stets nur die lokale Situation eines Sektors. Ein Sektor wird migriert, wenn er zu einem anderen Server mehr Kosten verursacht, als er nach der Migration zu seinem aktuellen Server verursacht. Das führte regelmäßig dazu, dass sich zwei Partitionen leicht ineinander verzahnen oder sich offensichtlich ungünstige Partitionsformen ergaben, die hätten gelöst werden können, indem zwei Partitionen ihre “Auswüchse” gegenseitig ausgetauscht hätten.
Eine derart großräumige Betrachtung würde den Aufwand des Algorithmus jedoch höchstwahrscheinlich derart drastisch erhöhen, da die Anzahl der zu überprüfenden Konstellationen von Migrationen exponentiell ansteigen würde.

7.3.4. Bisection

Der Bisectionalgorithmus ist ein graphbasierter Algorithmus und bringt dementsprechende Vor- und Nachteile mit sich.



(a) Partitionierungssituation mit Bisection in Test 2

Abbildung 7.2.: Beispielpartitionierungen mit Bisection

Der Vorteil einer graphbasierten Partitionierung ist, dass eine Partition niemals gespalten werden kann und immer als zusammenhängender Subgraph existiert. Außerdem ist die Sektorstruktur von WalkSim selbst schon ein Graph und dementsprechend perfekt für einen solchen Algorithmus geeignet.

Leider erweist sich der Bisection-Algorithmus nicht als so optimal, wie erhofft. Dies liegt hauptsächlich an der verwendeten Optimierungsfunktion $C_P = w_1 C_L + w_2 C_C$. Im Algorithmus wird versucht die Summe aus Lastkosten und Kommunikationskosten minimal zu halten. Im Ergebnis führt dies dazu, dass eine Partition mit extrem hohen Kommunikationskosten und wenig Agenten genauso viel Gesamtkosten hat wie eine Partition ohne Kommunikation aber dafür mit vielen Agenten. Da in WalkSim Agentenausführung und Synchronisierung in zwei getrennten Schritten ausgeführt wird, kommt es hier in den einzelnen Teilschritten zu extremen Lastungleichgewichten.
In Abb. 7.2(a) ist dies besonders deutlich zu sehen. Einige der Partitionen umfassen viele Sektoren mit sehr vielen Agenten, kommunizieren aber wenig mit Nachbarn, andere Partitionen beinhalten sehr wenige Agenten, sind allerdings extrem verschlungen und haben sehr viele Nachbarsektoren, sodass der Kommunikationsaufwand sehr hoch ist.
Zudem ist es sehr abhängig von der Simulation und der konkreten Implementierung der Agenten, wie die Gewichtung der beiden Faktoren w_1 und w_2 erfolgen sollte, da das Verhältnis von Rechenzeit zu Kommunikation bei jeder Agentenimplementierung anders sein kann. Dieses Verhältnis sollte idealerweise dynamisch gewichtet werden.

Ein weiteres Problem ist genauso wie im Peschlow-Algorithmus die lediglich lokale Optimierung, jedoch tritt sie bei diesem Algorithmus bedingt durch die Graph-Bildung deutlich mehr in Erscheinung. Kommt es dazu, dass sich eine sehr schmale, lang gezogene Partition entwickelt, ist es oft sinnvoll, einen kompletten Arm dieser Partition einer anderen hinzuzufügen. Dies wird jedoch dadurch verhindert, dass eine Partition nicht gesplittet werden kann. Befindet sich am Ende dieses Arms eine Situation, die lokal nicht weiter optimiert werden kann, so kann die Unförmigkeit nicht beseitigt werden und es ergibt sich in den meisten Fällen eine Partition in Form einer Hantel (siehe die hellgrüne Partition im unteren Teil der Abb. 7.2(a)).

Ein extremer Nachteil des Bisection-Algorithmus ist seine Laufzeit. Vor allem die initiale Partitionierung benötigt extrem viel Zeit, da die Dauer beinahe kubisch zur Anzahl der Sektoren ansteigt. In wirklich großen Simulationen mit mehreren tausend Sektoren und vielen Partitionen benötigt die initiale Partitionierung u.U. mehrere Minuten.

7.3.5. Peschlow + Bisection

Ein Versuch, einige Nachteile der beiden Algorithmen Peschlow und Bisection zu beseitigen, war eine Kombination beider Algorithmen. Die recht gute Kostenfunktion aus dem Peschlow-Algorithmus wurde mit der Graph-Struktur des Bisection-Algorithmus kombiniert. Beide Teilalgorithmen des Peschlow-Algorithmus wurden direkt nacheinander in einem Schritt durchgeführt. Weiterhin wurde ein Schwellwert eingeführt, unter welchem Sektoren trotz höherer Kommunikation zu anderen Plattformen nicht migriert wurden.

Die Kombination der beiden Algorithmen führte zu minimalen Verbesserungen bei Lastausgleich und Migrationen gegenüber den beiden Einzelalgorithmen. Der Kommunikationsaufwand sank zwar gegenüber dem puren Bisection-Algorithmus deutlich, lag in Summe jedoch weiterhin über den Ergebnissen des Peschlow-Algorithmus.
Die Nachteile aus der langen Laufzeit des Bisection-Algorithmus treffen auf den kombinierten Algorithmus ebenfalls zu. Durch die graphbasierte Partitionierung, welche keine Partitionssplittung zulässt, verformen sich die Partitionen zudem extrem und limitieren den Algorithmus vor allem in sehr zerklüfteten Simulationsumgebungen bei den möglichen Sektormigrationen.

Ebenso wie der pure Bisection-Algorithmus besitzt auch die Kombination mit dem Peschlow-Algorithmus den extremen Nachteil, dass die initiale Verteilung bei größeren Simulationen so lange braucht, dass der Einsatz des Algorithmus nicht mehr sinnvoll ist.

7.3.6. Zusammenfassung

Die fünf untersuchten Algorithmen sind in ihrer Herangehensweise und den daraus resultierenden Ergebnissen sehr unterschiedlich und haben entsprechend in verschiedenen Bereichen sowohl ihre Vor- als auch ihre Nachteile.

7.3.6.1. Vergleich

Die rein positionsoptimierten Algorithmen KMeans und QHull arbeiten am besten in einer großen freien Umgebung ohne Wände und andere Hindernisse und mit einer sehr homogen verteilten Agentenmenge und bringen hier Ergebnisse, die teilweise deutlich besser sind als die der anderen Algorithmen.

Algorithmen, die versuchen, neben der Lastverteilung auch die Kommunikationskosten zu optimieren, sind deutlich flexibler, was die Art der Simulationsumgebung angeht und sind deshalb potentiell besser für den praktischen Einsatz gedacht, da in der Realität große und hindernisfreie Szenarien eher eine Ausnahme darstellen.

Sehr schlecht umgehen konnten alle Algorithmen damit, wenn sich alle Agenten auf kleinem Raum befanden. Entweder wurden dabei zu viele Agenten auf nur einen Server verschoben oder es wurde kontinuierlich versucht vergeblich zu optimieren, was extrem viele Migrationen zur Folge hatte.
Es muss allerdings auch gesagt werden, dass eine solche Situation generell sehr schwer zu partitionieren ist, weil die Agenten dort teilweise auf weniger Sektoren verteilt sind, als Server zur Verfügung stehen. Eine Lösungsmöglichkeit wäre es hier, von vornherein nicht alle Server mit einzubeziehen, sondern die Agenten auf weniger Server zu verteilen und so wenigstens die Kommunikations- und Migrationskosten gering zu halten.

Vergleicht man die erhaltenen Werte aller Tests und Algorithmen miteinander, liefern die beiden Algorithmen, dessen Partitionierung auf dem Algorithmus von Peschlow basiert, die besten Ergebnisse im Bezug auf Lastausgleich und Migrationsanzahl. Der reine Peschlow-Algorithmus erzeugt dazu gegenüber dem P+B-Algorithmus einen geringeren Kommunikationsaufwand. Der P+B-Algorithmus disqualifiziert sich aufgrund seiner inakzeptablen Laufzeit für die initiale Partitionierung großer Simulationen für eine praktische Verwendung.
Alles in allem konnte folglich der Algorithmus von Peschlow in Summe die besten Ergebnisse erzeugen.

7.3.6.2. Allgemeine Schwachstellen

In allen untersuchten Algorithmen stellte die Kommunikationskostenoptimierung lediglich eine lokale Optimierung dar. Obwohl es häufig offensichtlich war, dass es sinnvoll wäre, eine ganze Gruppe von Sektoren eines Servers auf einmal auf einen anderen Server zu migrieren, konnte das nicht erkannt werden, weil die Sektoren stets einzeln betrachtet wurden und die Optimierungsfunktion für die Sektoren einzeln keine Verbesserung feststellen konnte.

Für den Lastausgleich nahmen bis auf den Peschlow-Algorithmus alle Algorithmen an, dass alle Agenten die gleiche Last erzeugen und auch der Peschlow-Algorithmus konnte keine zufriedenstellende Methode bieten, die Last eines Agenten zu messen. Dementsprechend werden alle Algorithmen beim Lastausgleich erheblich schlechter abschneiden, sobald Agenten verwendet werden, welche individuell unterschiedlich viel Last erzeugen.

Die initiale Partitionierung der Simulation ist ein Problem, welches keiner der untersuchten Algorithmen zufriedenstellend lösen konnte. Die drei Algorithmen, welche dafür KMeans nutzten, waren sehr auf die Zufallsverteilung der initialen Partitionen angewiesen. Wurden vorhandene Gruppen zufällig gut getroffen, entstand ein völlig anderer Performance-Verlauf, als wenn die Partitionen einzelne Gruppen in der Mitte teilen.
Die beiden auf Bisection basierenden Algorithmen erstellen zwar eine nahezu perfekte Verteilung, benötigen bei mehreren Tausend Sektoren und vielen Servern jedoch derart viel Zeit, dass deren Anwendung nicht sinnvoll ist.

Eine grundlegende Idee bei der Partitionierung in WalkSim war es, die Simulation in einzelne Sektoren einzuteilen und diese mit Hilfe von Nachbarschaftsinformationen bereits im Vorfeld mit Daten auszustatten, welche eine intelligente Partitionierung erleichtern.
Wie an den Ergebnissen zu sehen ist, sind die nicht graphbasierten Algorithmen in Summe keineswegs schlechter als die graphbasierten. Im Gegenteil stehen sich Letztere mit der Bedingung, stets eine geschlossene Partition zu erzeugen, vor allem bei zerklüfteten Sektorstrukturen regelmäßig selbst im Weg.

8. Ausblick

Sowohl in den Ergebnissen als auch in der darauf folgenden Diskussion der Ergebnisse konnte gezeigt werden, dass WalkSim eine taugliche Simulationsplattform für große Fußgängersimulationen darstellt.

Vor allem die Verwendung der partiellen Synchronisation in Kombination mit dem dafür entwickelten verteilten Messaging-System konnte drastische Vorteile gegenüber der in anderen Arbeiten verwendeten vollständigen Synchronisation bieten.
Die Skalierbarkeit der Plattform auf viele Server ist bereits gut, bietet jedoch weiterhin Verbesserungspotential. Vor allem die Kalibrierung der Parameter wie die Sektorgröße, Sichtbereichsgröße der Agenten und Simulationsschrittweite bieten Raum für Optimierungen. Diese Parameter sollten vor allem im Hinblick auf ihren Einfluss auf das Ergebnis einer Simulation untersucht werden.

Ein spannendes Feld für die Zukunft wäre der Einbau weiterer Simulationsteile, die nicht auf der Agentenplattform selbst basieren, sondern physikalische Informationen liefern. Ebenso wurde die Implementierung von Interaktionen der Agenten mit der Umwelt oder anderen Agenten in dieser Arbeit noch gar nicht betrachtet.

Betrachtet man die untersuchten Partitionierungsalgorithmen, so stellt man fest, dass hier vielversprechende Ansätze existieren, die allerdings alle ihre Schwächen haben. Insgesamt sind diese Algorithmen allesamt sehr *Low-Level* gehalten. Eine Idee für die weitere Forschung wäre, das Szenario und die KI der Agenten mit in die Partitionierung mit einzubeziehen und eine vorausschauende Partitionierung zu entwickeln, welche auch die Bewegungen der Agenten mit einbezieht und so die Anzahl von Migrationen weiter minimiert.

A. Weitere Bestandteile der WalkSim-Plattform

Im Folgenden werden weitere wichtige Bestandteile der WalkSim-Plattform beschrieben, welche im Laufe der Arbeit entwickelt wurden, jedoch kein relevanter Teil dieser Arbeit sind.

A.1. Visualisierung

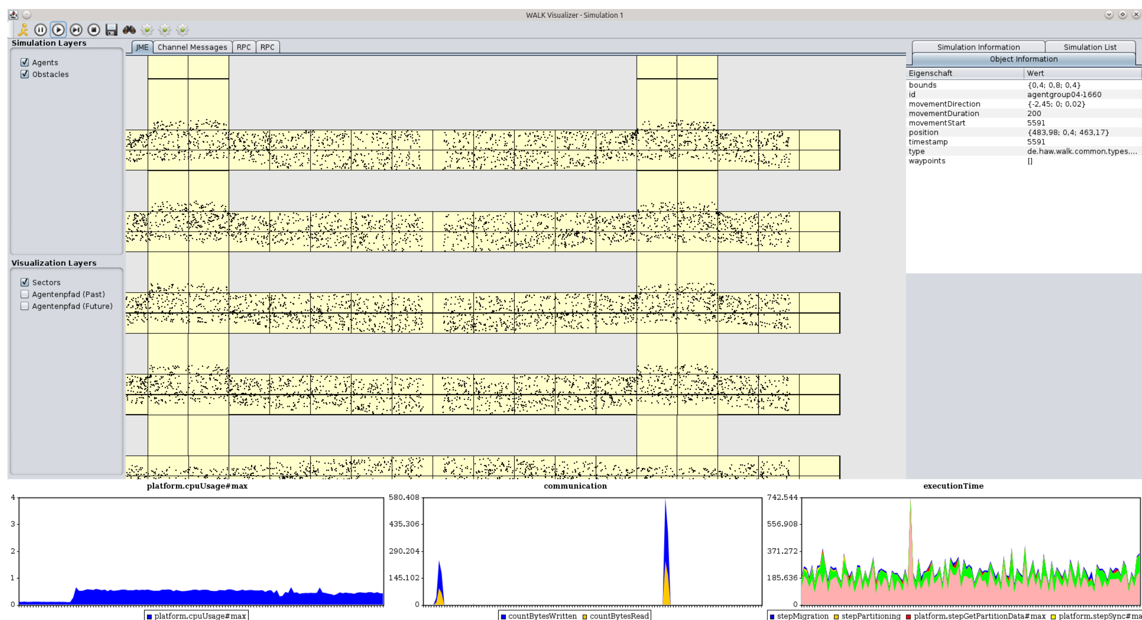


Abbildung A.1.: Screenshot der WalkSim-Bedienoberfläche

Für die Bedienung von WalkSim und die Ausführung der Experimente wurde eine primitive Benutzeroberfläche entwickelt.

Mit dieser Oberfläche können Simulationen im WalkManager gestartet, pausiert und gestoppt werden. Ebenso können alle laufenden Simulationen aufgelistet und beobachtet werden.

A.1.1. Benutzeroberfläche

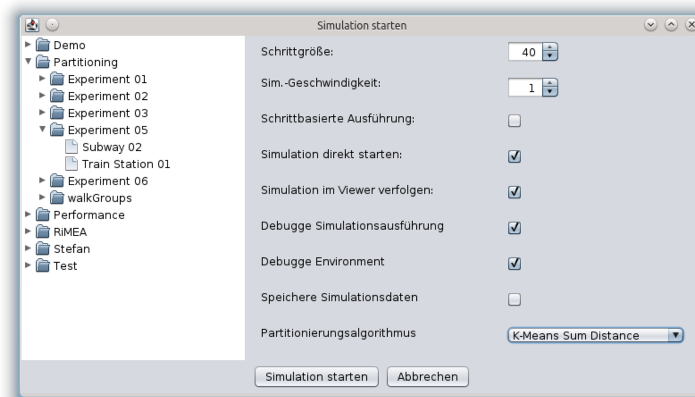


Abbildung A.2.: Dialog zum Starten einer neuen Simulation

Die Benutzeroberfläche gliedert sich in fünf Hauptbereiche (siehe Abb. A.1)

- Der große Bereich in der Mitte des Fensters visualisiert den aktuellen Zustand einer Simulation. Die Grafik dieser Visualisierung ist bewusst simpel und funktionell gehalten, um mit wenig Aufwand alle relevanten Informationen darzustellen. Es kann die Bewegung aller Agenten beobachtet und anhand der farbigen Kästchen die Verteilung der Sektoren im Serververbund verfolgt werden.
- Im oberen Teil des Fensters befindet sich die Simulationssteuerung. Mit dieser können neue Simulationen gestartet und bereits laufende pausiert oder gestoppt werden.
- Der linke Bereich der Benutzeroberfläche steuert die jeweils im Simulationsbereich sichtbaren Informationslayer. Diese können separat ein- und ausgeschaltet werden.
- Rechts vom Simulationsfenster können weitere Informationen über die Simulation und einzelne Objekte der Simulation beobachtet werden (z.B. die Eigenschaften einzelner Agenten).

- Der untere Bereich der WalkSim-GUI beinhaltet diverse konfigurierbare Datenreihen, die live zur Simulation visualisiert werden und nützliche Debugging-Informationen sowohl zur Plattform (z.B. CPU-Last, Netzwerkdatenverkehr) als auch zur Simulation (z.B. Schrittdauer) liefern.

Für den Start einer neuen Simulation werden alle verfügbaren Szenarien des WalkManagers geladen (Siehe Abb. A.2). Dabei lässt sich die auszuführende Simulation mit diversen Einstellungsmöglichkeiten parametrisieren.

A.1.2. Architektur

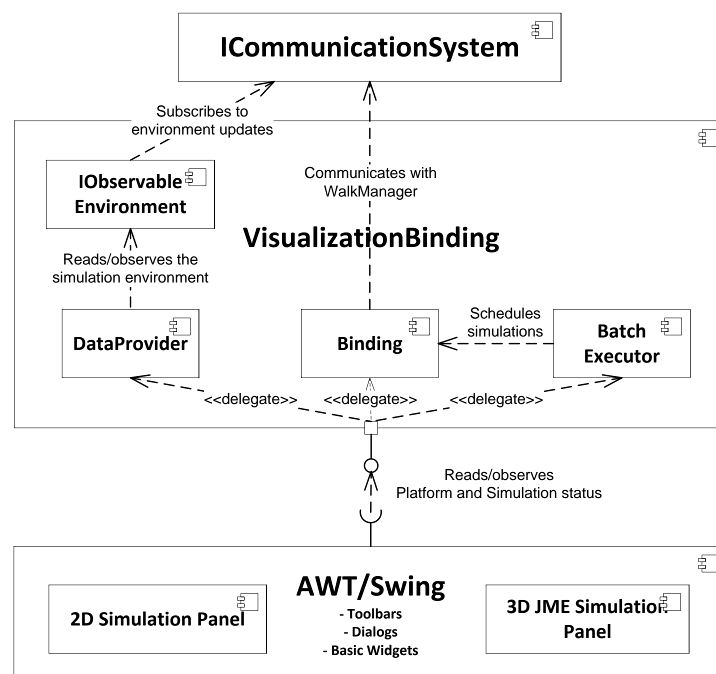


Abbildung A.3.: Komponenten der Visualisierung

Die Visualisierungskomponenten wurden von Beginn an modular entwickelt. So ist der Unterbau der Visualisierung, das *VisualizationBinding*, welcher diverse Klassen für die Kommunikation mit dem WalkManager und die Beobachtung des DVE enthält, von der eigentlichen grafische Visualisierung entkoppelt (siehe Abb. A.3).

Diese Modularisierung erlaubt es, auf einfache Weise andere grafische Oberflächen zu entwickeln, welche den gleichen Unterbau verwenden. Es ist bereits heute möglich zwischen einer simplen 2D-Visualisierung und einer einfachen 3D-Visualisierung auf Basis der jMonkey-Engine zu wechseln.

A.2. Runtime-Plattform

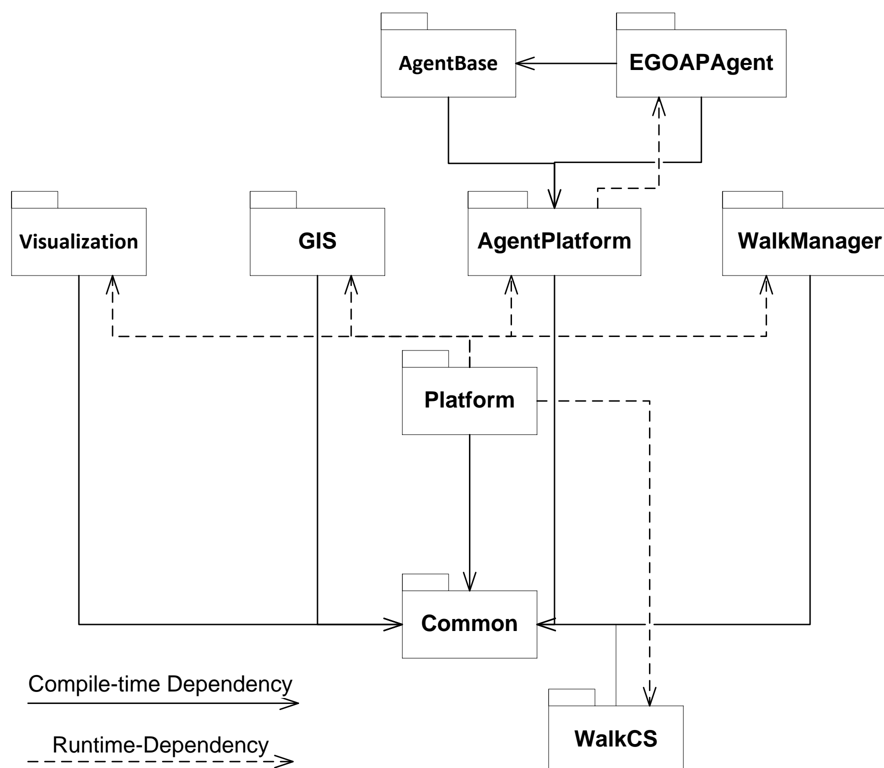


Abbildung A.4.: Paket-Abhängigkeiten in WalkSim

Die WalkSim-Plattform ist modular aufgebaut. Jede der einzelnen Komponenten kann jederzeit gegen eine andere Implementierung ausgetauscht werden, solange das definierte Interface der Komponente eingehalten wird.

Durch die Unabhängigkeit der Komponenten voneinander können diese auch beim Deploy-

ment separat ausgeliefert und in verschiedenen Kombinationen auf den einzelnen Servern gestartet werden.

Abbildung A.4 stellt die Abhängigkeiten zwischen den einzelnen Modulen dar. Bis auf die Agentenmodule sind alle Pakete lediglich abhängig vom Common-Modul, welches das fachliche Datenmodell, die Schnittstellendefinitionen aller Module und andere gemeinsam genutzte Klassen enthält. Das Modul *Platform* enthält eine Startroutine, welche per Dependency-Injection zum Plattformstart alle benötigten Module lädt und startet. So kann über eine Konfigurationsdatei oder über Kommandozeilenparameter für jeden Host separat definiert werden, welche Module gestartet werden sollen.
WalkSim kann verschiedene Agenten-Implementierungen verwalten. Diese werden ebenso getrennt von der AgentPlatform entwickelt und erst zur Laufzeit in das System eingebunden.

A.3. Kollisionserkennung

Um zu vermeiden, dass Agenten durcheinander oder durch statische Hindernisse hindurch wandern können, wurde eine einfache Kollisionserkennung implementiert. Diese ist bewusst sehr simpel gehalten, da der Einbau einer komplexen Physik-Engine unnötig viel Aufwand verursacht hätte und eine Kollisionserkennung für diese Arbeit ohnehin nur bedingt relevant ist.
Die Kollisionserkennung wird mit Hifle des Separating-Axis-Theorems [47] durchgeführt. Dieses findet für zwei sich schneidende Axis-Aligned-Bounding-Boxes die *Minimum Translation Vectors*, den kleinsten Verschiebungsvektor, bei welchem sich die AABBs nicht mehr schneiden würden.
Bei einer festgestellten Kollision werden beide Objekte (bzw. nur eines, wenn das andere ein statisches Hindernis ist) anhand dieses Minimum Translation Vectors verschoben und deren Bewegung gestoppt.

B. DVD-Anlage

Der Arbeit liegt ein DVD-Datenträger mit folgendem Inhalt bei:

- **WalkSim/bin/** Enthält die kompilierte Version von WalkSim. WalkSim kann mit Hilfe der **.sh* Scripte auf Unix-Systemen und der **.bat* Scripte auf Windows-Systemen gestartet werden. Das Script *walk-standalone.(sh|bat)* startet die WalkSim-Plattform als Standalone-System ohne Netzwerkkommunikation und mit allen benötigten Komponenten. Für eine Auswahl einzelner Module ist das Script *walk.(sh|bat)* zu verwenden. Die unterstützten Parameter sind der beigelegten README.txt zu entnehmen.
- **WalkSim/src/** Enthält den gesamten Quellcode von WalkSim inklusive aller benötigten Dritt-Bibliotheken.
  - **./WalkCommon/** Enthält alle Klassen der Pakete *Common* und *WalkCS*
  - **./WalkCore/** Beinhaltet den Quellcode der Komponenten *AgentPlatform*, *GIS*, *WalkManager*, *Platform* und der Agentenimplementierungen
  - **./Visualization/** Enthält den Quellcode und die verwendeten Ressourcen der Visualisierung
  - **./build.xml** Ant-Script zum Kompilieren und Ausführen von WalkSim auf Basis des Quellcodes
- **experiment-daten/graphen/** Visuelle Darstellung der Ergebnisse in diversen Graphen
- **experiment-daten/video/** Videoaufnahmen der einzelnen Partitionierungs-Experimente
- **MA_christian_thiel.pdf** Die digitale Version der Masterarbeit als PDF
- **README.txt**

Literaturverzeichnis

[1] Apache ActiveMQ.

[2] EODiSP. http://www.pnp-software.com/eodisp/home.html.

[3] Massive Software. CTAN http://www.massivesoftware.com/.

[4] RabbitMQ. http://www.rabbitmq.com/.

[5] H2 Database. CTAN http://www.h2database.com/html/main.html, dez 2012.

[6] JADE. CTAN http://http://jade.tilab.com/, nov 2012.

[7] MySQL 5.1 Referenzhandbuch. CTAN http://dev.mysql.com/doc/refman/5.1/de/index.html, dez 2012.

[8] PostgreSQL 9.2 Dokumentation. CTAN http://www.postgresql.org/docs/9.2/static/, dez 2012.

[9] HSQLDB. CTAN http://hsqldb.org/, feb 2013.

[10] Microsoft SQL Server. CTAN http://www.microsoft.com/en-us/sqlserver/default.aspx, feb 2013.

[11] MySQL :: The world's most popular open source database. CTAN http://www.mysql.com/, feb 2013.

[12] Oracle Database 11g. CTAN http://www.oracle.com/de/products/database/overview/index.html, feb 2013.

[13] PostgreSQL: The world's most advanced open source database. CTAN http://www.postgresql.org/, feb 2013.

[14] SQLite Homepage. CTAN http://www.sqlite.org/, jan 2013.

[15] Marten van Steen Andrew Tanenbaum. *Introduction to Multiagent Systems*. Pearson Studium, München/Germany, 2003.

[16] Apache Software Foundation. HBase. CTAN http://hbase.apache.org/, may 2012.

[17] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[19] Stephen Chenney. Flow tiles. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '04, pages 233–242, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[20] Judith Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The dod high level architecture: an update. In *Proceedings of the 30th conference on Winter simulation*, WSC '98, pages 797–804, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[21] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The department of defense high level architecture. In *Proceedings of the 29th conference on Winter simulation*, WSC '97, pages 142–149, Washington, DC, USA, 1997. IEEE Computer Society.

[22] Richard M. Fujimoto and Richard M. Weatherly. Time management in the dod high level architecture. In *Proceedings of the tenth workshop on Parallel and distributed simulation*, PADS '96, pages 60–67, Washington, DC, USA, 1996. IEEE Computer Society.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[24] GigaSpaces. GigaSpaces homepage. Avaliable at
textithttp://www.gigaspaces.com/, 2006.

[25] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News*, page 2002, 2002.

[26] Dirk Helbing, Illes Farkas, and Tamas Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, September 2000.

[27] Jeff Orkin. Goal Oriented Action Planning. CTAN `http://web.media.mit.edu/~jorkin/goap.html`.

[28] Franziska Klugl, Georg Klubertanz, and Guido Rindsfuser. *Agent-Based Pedestrian Simulation of Train Evacuation Integration Environmental Data*, pages 631–638. Springer verlag Berlin Heidel-Berg, 2009.

[29] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[30] Malcolm Yoke Hean Low, Wentong Cai, and Suiping Zhou. *A federated agent-based crowd simulation architecture*, pages 188–194. 2007.

[31] Miguel Lozano, Pedro Morillo, Daniel Lewis, Dirk Reiners, and Carolina Cruz-Neira. A distributed framework for scalable large-scale crowd simulation. In *Proceedings of the 2nd international conference on Virtual reality*, ICVR'07, pages 111–121, Berlin, Heidelberg, 2007. Springer-Verlag.

[32] John C. S. Lui and M. F. Chan. An efficient partitioning algorithm for distributed virtual environment systems. *IEEE Trans. Parallel Distrib. Syst.*, 13:193–211, March 2002.

[33] R. Minson and G. K. Theodoropoulos. Distributing repast agent-based simulations with hla. *Concurr. Comput. : Pract. Exper.*, 20(10):1225–1256, July 2008.

[34] Patrick Peschlow, Tobias Honecker, and Peter Martini. A flexible dynamic partitioning algorithm for optimistic distributed simulation. In *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, PADS '07, pages 219–228, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Presgias Canada Inc. AI.Implant. CTAN http://www.presagis.com/products_services/products/modeling-simulation/simulation/aiimplant/.

[36] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[37] Arnon Rotem-Gal-Oz. Fallacies of Distributed Computing Explained. CTAN http://www.rgoarchitects.com/Files/fallacies.pdf, dez 2012.

[38] Wei Shao and Demetri Terzopoulos. Autonomous pedestrians. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '05, pages 19–28, New York, NY, USA, 2005. ACM.

[39] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[40] Anthony Steed and Roula Abou-Haidar. Partitioning crowded virtual environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, VRST '03, pages 7–14, New York, NY, USA, 2003. ACM.

[41] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, February 2003.

[42] Toshihiro Takahashi and Hideyuki Mizuta. Efficient agent-based simulation framework for multi-node supercomputers. In *Proceedings of the 38th conference on Winter simulation*, WSC '06, pages 919–925. Winter Simulation Conference, 2006.

[43] Matthias Teschner, Bruno Heidelberger, Matthias Mueller, Danat Pomeranets, and Markus Gross. Optimized spatial hashing for collision detection of deformable objects. pages 47–54, 2003.

[44] Gerta Köster Thomas Thiel-Clemen, Stefan Sarstedt. Walk - emotion-based pedestrian movement situation in evacuation scenarios. In *Simulation in Umwelt- und Geowissenschaften - Workshop Berlin 2011*, pages 103–112, Aachen, DE, 2011. Shaker Verlag.

[45] G. Vigueras, M. Lozano, and J. M. Orduña. Enhancing workload balancing in distributed crowd simulations through the partitioning method. In *proceedings of the 2009 International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2009).*, pages 1117–1128, 2009. ISBN: 978-84-612-9727-6.

[46] William. NoSQL Databases. CTAN http://nosql-database.org/, dez 2012.

[47] William. Separating Axis Theorem. CTAN http://www.codezealot.org/archives/55, may 2012.

[48] Michael Woolridge and Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[49] Suiping Zhou, Dan Chen, Wentong Cai, Linbo Luo, Malcolm Yoke Hean Low, Feng Tian, Victor Su-Han Tay, Darren Wee Sze Ong, and Benjamin D. Hamilton. Crowd modeling and simulation technologies. *ACM Trans. Model. Comput. Simul.*, 20:20:1–20:35, November 2010.

Abbildungsverzeichnis

2.1. Verteilte Architektur beim System der University of Valencia [45] 8
2.2. Architektur in der HLA [2] 15
2.3. Bigtable's Verteilungshierarchie [18, S. 4] 19
2.4. Verschiedene geometrische Formen von Sichtbereichen 21
2.5. Zwei mögliche räumliche Partitionierungen einer gleichmäßig ausgelasteten Simulation . 23
2.6. Pseudocode des KMeans-Algorithmus 26
2.7. Pseudocode des QHull-Algorithmus 28
2.8. Pseudocode des Peschlow-Algorithmus (Teilalgorithmus für Kommunikationsoptimierung 29
2.9. Pseudocode des Peschlow-Algorithmus (Teilalgorithmus für Lastausgleich) 30
2.10. Pseudocode des Takahashi-Algorithmus 31
2.11. Pseudocode des Bisection-Algorithmus von Lui 32

3.1. Grundlegende Architektur von WalkSim 35
3.2. Fachliche Integration des DVE in WalkSim 37
3.3. Sektorstruktur des DVE 38

4.1. Klassen des Distributed Environment 45
4.2. Fachliches Datenmodell des DVE 46
4.3. Basisobjekt des DVE 49
4.4. Spatiales Hashing in 2D Umgebung 50
4.5. Weg einer Nachricht in einem zentralisierten Messagebus 59
4.6. Weg einer Nachricht in einem verteilten Messagebus 60
4.7. Hauptklassen des WalkCS 61
4.8. Komponenten des Agenten 63
4.9. Wegfindung des Agenten 64
4.10. Beispiel einer Szenariodefinition in XML 65

5.1. Aufbau von Test 1: Einzelgruppen 72
5.2. Aufbau von Test 2: Chaotische Menge 73
5.3. Aufbau von Test 4: Homogene Bewegung 74
5.4. Aufbau von Test 5: Bahnhof 75

6.1. Benchmark: Partielle gegen vollständige Synchronisation 78
6.2. Benchmark: Synchronisationsdauer 79
6.3. Benchmark: Übertragene Datenmenge 80
6.4. Ergebnisse des Skalierungstests 81
6.5. Testergebnisse für Test 1 83

6.6. Testergebnisse und Beispielpartitionierungen für Test 2 84
6.7. Testergebnisse und Beispielpartitionierungen für Test 3 86
6.8. Testergebnisse für Test 4 87
6.9. Testergebnisse für Test 5 88

7.1. Beispielpartitionierungen mit Peschlow 96
7.2. Beispielpartitionierungen mit Bisection 98

A.1. Screenshot der WalkSim-Bedienoberfläche 103
A.2. Dialog zum Starten einer neuen Simulation 104
A.3. Komponenten der Visualisierung 105
A.4. Paket-Abhängigkeiten in WalkSim 106

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 12. Februar 2013	
Ort, Datum	Unterschrift