Ein datenzentriertes Programmiermodell für verteilte virtuelle Welten

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Michael Sonnenfroh
aus Heidenheim

Düsseldorf, Oktober 2010

Aus dem Institut für Informatik
der Heinrich-Heine Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Erstgutachter: Prof. Dr. Michael Schöttner
Zweitgutachter: Prof. Dr. Martin Mauve
Drittgutachter: Prof. Dr. Peter Schulthess

Tag der mündlichen Prüfung: 18.11.2010

Kurzfassung

Virtuelle Welten sind inzwischen ein fester Bestandteil des gesellschaftlichen Lebens geworden. Allen bisherigen Ansätzen ist gemeinsam, dass sie auf nachrichtenorientierte Technik für die Verteilung der Welt zurückgreifen, die eng mit der verwendeten Netzwerkarchitektur gekoppelt ist. Das verwendete nachrichtenorientierte Programmiermodell reicht dabei bis in den Aufbau der für die Inhalte einer virtuelle Welt verwendeten Algorithmen, wodurch Änderungen an der Netzarchitektur auch Änderungen an den Algorithmen nach sich ziehen. Auch die Umsetzung der für die Welt wichtigen Konsistenzmodelle erfolgt in enger Kopplung mit den verwendeten Nachrichten und ist ebenfalls sowohl abhängig von der Netzarchitektur, als auch von der gewählten Implementierung einer Verteilungskomponente.

Um diesen Nachteilen zu begegnen, wurde im Rahmen der Dissertation ein neues, datenzentriertes Programmiermodell namens TGOS (Typed Grid Object Sharing) entworfen. Das TGOS-Modell integriert in neuer Weise die Konzepte speicherbasierter Verteilungssysteme (engl. distributed shared memory, kurz DSM) mit den Vorteilen der bisher verwendeten nachrichtenorientierten Technik. Neben einer strikten Abstraktion der Netzarchitektur werden auch allfällige Konsistenzmodelle innerhalb des Programmiermodells definiert und nur mit den im Programmiermodell definierten Operationen umgesetzt. Ebenso definiert das TGOS-Modell alle wesentlichen Mechanismen, welche für die Verteilung einer virtuellen Welt unabdingbar sind, wie beispielsweise Persistenz und Freispeichersammlung oder Gruppenkommunikation und Lastverteilung.

Das vorgestellte Programmiermodell wurde erfolgreich im Rahmen der prototypischen virtuellen Welt Wissenheim Worlds sowohl konzeptionell, als auch praktisch erprobt. Ferner zeigen die durchgeführten Messungen, dass die resultierende datenzentrierte Architektur effizient ist.

Abstract

Virtual worlds are becoming more and more a part of the everyday life. For data distribution, current approaches are commonly using message passing mechanisms, which are closely coupled to the underlying network architecture. The message-based programming model even impacts the algorithms that describe the content of a virtual world, so that a change to the network architecture requires altering the algorithms as well. The implementation of the consistency models, which are very important for virtual worlds, is not only closely coupled with the messages and the network architecture used, but depends also heavily on the implementation details of the component in charge of data distribution.

In order to compensate the formerly mentioned problems, this thesis defines a new, data-centric programming model called TGOS (Typed Grid Object Sharing). The TGOS-model presents a new approach by integrating concepts known from distributed shared memory systems with the advantages of the currently used message-based mechanisms. Another novelty is the definition of new consistency models within the boundaries of the TGOS model using only the mechanism provided by it. The TGOS programming model also defines all techniques necessary for distributing virtual worlds, like persistency and garbage collections or group communications and load balancing.

The proposed programming model has been successfully evaluated with a prototype of a virtual world call ed Wissenheim Worlds. The presented test results show that the approach is feasible and efficient.

Danksagung

Meinen herzlichsten Dank möchte ich Herrn Prof. Dr. Michael Schöttner aussprechen, der mir die Möglichkeit zu dieser Dissertation eröffnet hat. Ich bin sehr dankbar für die konsequente und großzügige Unterstützung und den damit verbundenen persönlichen Einsatz, der entscheidend zu dieser Arbeit beigetragen hat.

Ich möchte außerdem Herrn Prof. Dr. Peter Schulthess vielmals für die freundliche Aufnahme an der Universität Ulm und seine Unterstützung danken.

Mein Dank gilt auch Herrn Prof. Dr. Martin Mauve für die Begutachtung der Arbeit.

Meinen Düsseldorfer Kollegen Florian Müller, Kim Rehmann, Dr. John Mehnert-Spahn und Michael Braitmeier sowie Steffen Gerhold, Tobias Bäuerle und den anderen Kollegen an der Universität Ulm danke ich für ihren fachlichen und persönlichen Austausch.

Ganz besonderer und herzlichster Dank gilt meiner zukünftigen Frau Lene für ihre liebevolle Unterstützung meiner Anstrengungen und ihren unermüdlichen Korrektureinsatz.

Nicht zuletzt möchte ich mich ganz herzlich bei meinen Eltern, Irena und Günter Sonnenfroh und meinen Großeltern bedanken, die mir immer hilfreich mit Rat und Tat zur Seite stehen.

Inhaltsverzeichnis

1 Einleitung **9**
1.1 Historie . . . 10
1.2 Kategorisierung virtueller Welten . . . 11
1.3 Virtuelle Welten in der Gesellschaft . . . 13
1.4 Zielsetzung der Arbeit . . . 14
1.5 Struktur der Arbeit . . . 14

2 Architektur verteilter Welten **16**
2.1 Programmiermodelle . . . 16
2.1.1 Basismodelle . . . 16
2.1.2 Netzwerk-Protokolle . . . 19
2.1.3 Ereignis-gesteuerte Programmierung . . . 20
2.2 Daten und Engines . . . 21
2.2.1 Klassifikation . . . 21
2.2.2 Strukturen . . . 23
2.2.2.1 Szenengraphen . . . 23
2.2.2.2 Enginestrukturen . . . 24
2.3 Lastverteilung . . . 24
2.3.1 Partitionierung . . . 24
2.3.2 Cluster & Grid . . . 26
2.3.3 Multiple Welten . . . 28
2.4 Konsistenz . . . 29
2.4.1 Konsistenzmodelle . . . 30
2.4.2 Konsistenzdomänen . . . 31
2.5 Zuverlässigkeit . . . 31
2.5.1 Verfügbarkeit . . . 31
2.5.2 Persistenz . . . 33
2.6 Sicherheit . . . 33
2.7 Zusammenfassung . . . 34

3 Das TGOS-Programmiermodell **36**
3.1 Nachrichten oder
verteilter gemeinsamer Speicher . . . 37

3.2 Das TGOS-Programmiermodell 40
3.2.1 Grundkonzept 40
3.2.2 Typisierung 42
3.2.3 Basisoperationen & Ereignisse 42
3.2.4 Hüllenbildung & Referenzen 47
3.2.5 Terminologie 50
3.2.6 Ordnung 50
3.2.7 Atomare Operationen 52
3.2.8 Verzeichnisdienst 53
3.2.9 Persistenz & Speicherbereinigung 54
3.2.10 Gruppenkommunikation 57
3.2.10.1 Gruppenzugehörigkeit 57
3.2.10.2 Gruppenverwaltung 58
3.2.10.3 Gruppendefinition im TGOS-Modell 59
3.2.11 Multi-Threading 60
3.2.12 Anforderungen an die Replikationsschicht 61
3.2.12.1 Funktionale Anforderungen 61
3.2.12.2 Nichtfunktionale Anforderungen 64
3.2.13 Ereignisse & Serialisierung 65
3.2.14 Funktionsübersicht 66
3.3 Verwandte Arbeiten 66
3.4 Zusammenfassung 69

4 Architektur einer verteilten Welt mit TGOS 71
4.1 Datenzentriertes Entwurfsmuster 71
4.1.1 Verteilte Datenstruktur 72
4.1.2 Interaktion und lokaler Kontext 75
4.1.3 Datenklassifikation 77
4.2 Komponenten einer verteilten Welt 78
4.2.1 Programmablauf 84
4.2.2 Aktualisierungsformen 85
4.3 Konsistenzmanagement 86
4.3.1 Strikte und schwache Konsistenz 86
4.3.2 Transaktionale Konsistenz 88
4.3.2.1 Transaktion bei hoher Latenz 89
4.3.2.2 Transaktionale Konsistenz mit TOGS 90
4.3.2.3 Automatische Lese- & Schreibmengen-Aufzeichnung 92
4.3.3 Konsistenzdomänen 94
4.3.4 Szenenspezifische Konsistenz mit TGOS 95
4.4 Entwurf einer Szene 95
4.4.1 Voraussetzungen 95
4.4.2 Aufbau des Szenengraphens 97
4.4.3 Integration der Spiellogik 98

4.4.4 Zusammenfassung 102
4.5 Last- und Replikationsmanagement 102
4.5.1 Lastverteilung durch Area-of-Interest 102
4.5.1.1 Statische AoI-Zuordnung 103
4.5.1.2 Dynamische AoI-Zuordnung 103
4.5.2 Die Infrastruktur einer verteilten Welt 106
4.5.2.1 Auswahl der Netzarchitektur 107
4.5.2.2 Replikationsschicht 107
4.5.2.3 Dienste der verteilten Welt 111
4.5.2.4 Verknüpfung von Diensten und Replikation . 113
4.6 Ein nachrichtenbasierter Ansatz im Vergleich 115
4.6.1 Das RedDwarf-Framework 115
4.6.2 Volleyball mit RedDwarf 116
4.6.3 Bewertung 118
4.7 Verwandte Arbeiten 118
4.8 Zusammenfassung 119

5 Messungen 121
5.1 Evaluation der Basiskomponenten 121
5.1.1 Serialisierungsaufwand 122
5.1.2 Replikationsschichten 123
5.1.2.1 Planare Replikation 124
5.1.2.2 Hierarchische Replikation 127
5.1.3 TGOS-Operationen 128
5.1.4 Transaktionale Konsistenz 129
5.2 Evaluation einer verteilten virtuellen Welt 132
5.2.1 Area-of-Interest Messung 133
5.2.2 Messung der Lastverteilung 134
5.2.3 Transaktionale Konsistenz 138
5.2.4 Mobile Klienten 139
5.2.5 Feldtests 140
5.3 Bewertung 142

6 Zusammenfassung 144
6.1 Resultat 144
6.2 Ausblick 146

Literaturverzeichnis 148

Abbildungsverzeichnis 157

Tabellenverzeichnis 160

Listing 161

Kapitel 1

Einleitung

Virtuelle Welten existieren in unterschiedlichsten Formen und haben bereits in vielen Bereichen des täglichen Lebens Einzug gehalten. Obwohl weit verbreitet, fehlt es bisher an einer einheitlichen Definition des Begriffes der virtuellen Welt. Eine mögliche Definition wird von Castronova in[CCE+07] gegeben und bezieht sich explizit auf 3-dimensionale Welten, eine weitere, mit dem Schwerpunkt auf den Begriff der *Welt*, findet sich in [Bar03].

Eine Kombination aus den in der Literatur beschriebenen möglichen Definitionen ist in 1 gegeben und definiert die im Rahmen der Arbeit verwendete Bedeutung des Begriffs.

Definition 1: Virtuelle Welten
Eine virtuelle Welt ist eine computergenerierte, persistente Umgebung, in der ein Teilnehmer durch ein virtuelles Abbild, einen Avatar, präsent ist. Die Welt ist in sich selbst logisch geschlossen und die Aktionen eines Avatars spiegeln sich, im Rahmen der angebotenen Möglichkeiten, in der Welt wieder.

Definition 1 beinhaltet noch nicht den verteilten Charakter, der vielen Welten zugrunde liegt, weshalb in 2 die virtuelle Welt zu einer verteilten virtuellen Welt erweitert wird. In der Literatur werden beide Bezeichnungen jedoch meist synonym verwendet und auch im Rahmen der Arbeit ist eine virtuelle Welt gleichbedeutend mit einer verteilten virtuellen Welt.

Definition 2: Verteilte virtuelle Welten
Ein verteilte virtuelle Welt ist eine virtuelle Welt, in der mehrere Teilnehmer gleichzeitig am selben Ort zusammenkommen und in unterschiedlichster Weise miteinander oder mit der Welt interagieren und kommunizieren können.

1.1 Historie

Als die wohl erste verteilte virtuelle Welt kann das in Abbildung 1.1 dargestellte und 1988 entwickelte Multi-User Dungeon (kurz MUD [CN93]) angesehen werden. Die Welt bot eine textbasierte Oberfläche, residierte auf einem einzelnen Server und war für mehrere Spieler via Telnet zu erreichen. In MUD konnten die Spieler zwischen verschiedenen, textuell beschriebenen "Räumen"wechseln und mit den darin enthaltenen Gegenständen und Personen interagieren.

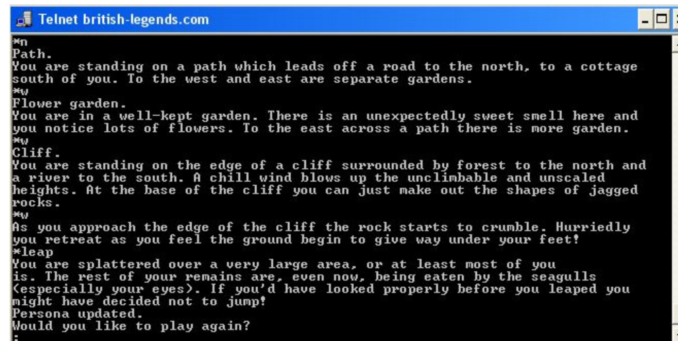


Abbildung 1.1: MUD 1

Viele Konzepte, die für die Verteilung aktueller virtueller Welten Verwendung finden, wurden im Rahmen mehrspielerfähigen Computerspielen (Definition 3) entwickelt.

Definition 3: Mehrspielerfähige Computerspiele
Unterhaltungssoftware in Form eines Spiels, in dem mehrere Spieler gemeinsam mit anderen spielen können. Die maximal mögliche Spielerzahl liegt meistens im unteren zweistelligen Bereich, typischerweise zwischen acht oder sechzehn. Das Spiel ist auf eine kurzzeitige Spielerfahrung (wenige Stunden) ausgelegt, die Persistenz ist meistens nicht gegeben und Speicherungsmöglichkeiten, um später in der gleichen Konstellation fortzufahren können, sind die Ausnahme.

Die ersten verteilten Spiele waren meist rundenbasierte Strategietitel (beispielsweise Spaceward HO!), bei denen die Daten einer neue Runde via Dateien verteilt wurden, wodurch diese entweder elektronisch (beispielsweise FTP oder eMail) oder physikalisch mit Disketten in Postbriefen an entfernte Spieler verschickt wurden. Der nächste Schritt waren Echtzeittitel, wie beispielsweise der Ego-Shooter Doom, bei denen Spieler sich über ein LAN oder mit Modem zu einem gemeinsamen Spiel verbinden konnten. Mit dem 1998 erschienen Echtzeitstrategiespiel Starcraft, führte Blizzard das kostenlose

Battle.net ein, eine internetbasierten Plattform, die es ermöglicht, weltweit mit Anderen gemeinsam zu spielen. Das Battle.net bietet dabei keine konsistente virtuelle Welt, sondern es erlaubt, einzelne Spiele mit bis zu acht Spielern durchzuführen, die in sich abgeschlossen sind.

Als eine der ersten 3-dimensionalen verteilten virtuellen Welten kann Everquest angesehen werden, welches 1999 erschien und in Grundzügen alle wesentlichen Merkmale moderner Ansätze bot. Der wohl bekannteste Titel dürfte das 2004 von Blizzard Entertainment veröffentlichte und in Abbildung 1.2 dargestellte World of Warcraft (kurz WoW) sein. Mit inzwischen über zwölf Millionen zahlenden Nutzern stellt es zudem bis heute die größte kommerzielle virtuelle Welt dar.

Abbildung 1.2: World of Warcraft

1.2 Kategorisierung virtueller Welten

Virtuelle Welten lassen sich nach einer ganzen Reihe verschiedener Kriterien einteilen und, analog zur Definition des Begriffes, gibt es auch hier im Moment keine einheitliche Klassifikation. Die nachfolgend beschriebene Kategorisierung nach Präsenz, Inhalt und Bezahlmodellen kann daher auch nicht als allgemeingültig oder vollständig erachtet werden und dient ausschließlich dazu, einen Einblick in das komplexe Thema zu gewinnen.

Präsenz

Als Präsenz bezeichnet man den Grad des Empfindens eines Nutzers, in der virtuellen Welt anwesend (präsent) zu sein. Dabei kann zwischen einer *physischen Präsenz* und einer *sozialen Präsenz* unterschieden werden.

Die physische Präsenz, oder auch Telepräsenz [Ste92], bezieht sich auf den Grad der physischen Teilnahme, die sowohl audio-visuell als auch

physisch sein kann. Der Grad der physischen Immersion richtet sich dabei nach der, von der virtuellen Welt bereitgestellten, audio-visuellen Gestaltung. Beispielsweise haben Welten, die mit Hilfe 3-dimensionaler Grafik und Surround-Sound dargestellt werden, einen höheren Grad an Telepräsenz, als HTML-basierte Welten mit einfacher oder abstrakter Grafik.

Die soziale Präsenz [SWC76] bezieht sich in der ursprünglichen Definition auf die Güte der bereitgestellten Kommunikationskanäle. Viele kommerzielle Welten bieten Spielern nicht nur textuelle Kommunikation via Text-Chats an, sondern ermöglichen auch Voice-Chats. Zusätzlich zu den Kommunikationskanälen bilden sich in aktuellen Welten soziale Bindungen und/oder Zwänge aus, die durch die Spielmechanik und/oder andere Teilnehmer hervorgerufen werden. Viele virtuelle Welten stellen Aufgaben bereit, die nur von einer Gruppe von Spieler gelöst werden können und bieten daher auch spezielle Funktionen, um Interessengemeinschaften (beispielsweise Gilden [DYNM07]) zu bilden, zu verwalten und zu bewerben. Auch ist es nicht unüblich, Aufgaben so schwierig zu gestalten, dass Gruppen vorher gemeinsam trainieren müssen, um diese Aufgaben zu lösen.

Als eine Verbindung zwischen physischer und sozialer Präsenz kann der durch Goffman [Gof66] geprägte Begriff der *Co-Präsenz* gesehen werden. Goffman beschreibt den Begriff als Zustand in dem Teilnehmer " sense that they are close enough to be perceived in whatever they are doing, including their experiencing of others, and close enough to be perceived in this sensing of being perceived"(*Goffman*).

Inhalt

Neben den Präsenzfaktoren lassen sich virtuelle Welten auch nach ihrem Inhalt klassifizieren. Die bekannteste und umfangreichste Kategorie sind die *Massivly Multiplayer Role Playing Games* (MMORPG), zu der beispielsweise World of Warcraft zählt . Eine weitere Kategorie stellen die freien Welten oder virtuellen *Get-Aways* dar, deren wohl bekanntester Vertreter *Second Life* (siehe Abbildung 1.3) ist. Auch virtuelle Chat-Räume, wie beispielsweise *Google Lively*, lassen sich zu dieser Kategorie zählen. Eine weitere Klasse bilden die sogenannten *Social Games*, wie beispielsweise *Farmville* in Facebook, bei dem die Spieler nicht direkt miteinander interagieren, sich jedoch an den, im Spiel errungenen, Erfolgen messen und vergleichen können.

Bezahlmodell

Virtuelle Welten gliedern sich grundsätzlich in freie Welten (beispielsweise Second Life), meist Free-to-Play genannt, und Welten, für deren Zugang eine monatliche Gebühr zu entrichten ist (WoW, Aion, etc.). Mit dem Aufkommen großer virtueller Welten war das Abo-Modell

das vorherrschende Bezahlmodell, da es den Entwicklern eine konstante Einnahmequelle versprach, welche die Kosten für den Betrieb der Welt (Server-Cluster, Netzanbindung, Service & Support, etc.) decken sollte. Mit zunehmender Dichte an verfügbaren Titeln sinkt für viele Projekte jedoch der Abonnentenstamm deutlich, was zu entsprechenden Einbußen führt.

Als Alternative entwickelte sich das *Micro-Payment-Modell*, bei dem der Zugang zur Welt frei ist, der Spieler aber die Möglichkeit besitzt, Spielfortschritte und/oder besondere Gegenstände für reales Geld zu kaufen. Inzwischen stellen viele virtuelle Welten, die auf dem Abo-Modell basieren, mangels Nutzer auf eine Free-to-Play-Variante um - und das mit durchaus beträchtlichem Erfolg (beispielsweise *Turbine* mit Dungeons & Dragons und Herr der Ringe Online). Eine weitere Möglichkeit, die beispielsweise in SecondLife Verwendung findet, ist der Verkauf oder die Verpachtung von Grundbesitz in der virtuellen Welt. Ein Teilnehmer erwirbt mit dem Kauf eines Stückchen virtuellen Lands auch das Recht, eigene Objekt darauf zu erschaffen und/oder auszustellen.

Abbildung 1.3: Second Life

1.3 Virtuelle Welten in der Gesellschaft

Virtuelle Welten sind längst aus der Nische der Spiele für Kinder in die Mitte der Gesellschaft gelangt. Insbesondere in den asiatischen Ländern erfreuen sich virtuelle Welten großer Beliebtheit und Verbreitung. Selbst in Deutschland gewinnen sie immer mehr mediale Aufmerksamkeit, wie 2007 am Beispiel von SecondLife zu sehen war.

Mit der zunehmenden Nutzung der virtuellen Welten im alltäglichen Leben treten, bei manchen Teilnehmern auch vermehrt Suchterscheinungen

auf. Insbesondere in virtuellen Spielwelten, wie WoW, Aion, etc. ist dies deutlich zu beobachten. Spieler verbringen oftmals Tage in der virtuellen Welt, ohne Kontakt zur realen Außenwelt. Die Grenze zwischen Suchtverhalten und einer bewussten Verlagerung sozialer Aktivitäten in die Virtualität [NWH05, Gri10] sind dabei nicht einfach zu ziehen. Bei extremen Formen der Sucht verloren Spieler beispielsweise ihren Beruf und den sozialen Kontakt zu ihren Familien. Neben den Studien, die sich mit diesen Auswirkungen befassen, gibt es auch eine Reihe von Arbeiten (beispielsweise [Yee06]), welche die Attribute einer virtuellen Welt untersuchen, die zum einen motivieren, aber auch in extremer Weise zu einer Sucht führen können.

1.4 Zielsetzung der Arbeit

Obwohl die Zahl der entwickelten und angebotenen 3-dimensionalen verteilten virtuellen Welten stetig wächst, hat sich seit den Anfängen mit Everquest wenig am zugrunde liegenden Modell der Verteilung geändert. Insbesondere die Konzentration auf Nachrichten und eine oftmals starke Fixierung auf die jeweilige Netzarchitektur kennzeichnen die bisherigen Ansätze.

Im Rahmen der Arbeit wird daher, ausgehend von den Unzulänglichkeiten bestehender Ansätze, ein neuartiges Programmiermodell entworfen, welches speziell auf die Anforderungen verteilter virtueller Welten zugeschnitten ist. Neben einer Vereinfachung der Programmierung soll durch den neuen Ansatz auch eine Abstraktion der Netzarchitektur stattfinden. Dadurch soll diese austauschbar sein, ohne dass, wie bisher, bestehende Algorithmen der Welt angepasst werden müssen. Zusätzlich soll die Integration neuer Konsistenzmodelle mit Hilfe der Basisfunktionalität des Programmiermodells erfolgen, anstatt wie bisher üblich fest in das Programmiermodell integriert zu werden. Auch dürfen die Konsistenzmodelle nicht durch den Austausch der Netzarchitketur beeinflusst werden und sollten außerdem leicht modularisierbar sowie im laufenden Betrieb austauschbar sein.

1.5 Struktur der Arbeit

In Kapitel 2 wird ein Überblick über den Stand der Technik verteilter virtueller 3-dimensionaler Welten, vergleichbar mit SecondLife oder World of Warcraft, gegeben. Neben einer allgemeinen Beschreibung der benötigten Komponenten, liegt der Schwerpunkt dabei auf den für die Verteilung verwendeten Konzepten. Zusätzlich zur Beschreibung der verwendeten Netzarchitekturen erfolgt auch ein Diskurs über gängige Lastverteilungsalgorithmen sowie eine Übersicht über die, in virtuellen Welten gängigen, Konsistenzmodelle.

Ausgehend von den in Kapitel 4 beschriebenen Ansätzen werden in Kapitel 3 deren bestehenden Nachteile und Unzulänglichkeiten identifiziert, sowie

Anforderungen für ein verbessertes Programmiermodell definiert. Diese Anforderungen erfüllend, wird eine neues, datenzentriertes Programmiermodell vorgestellt, das *Typed Grid Object Sharing*-Modell (kurz TGOS-Modell oder nur TGOS). Dessen Konzepte, Eigenschaften und Anforderungen werden im Verlauf des Kapitels im Detail definiert und erläutert.

Mit Hilfe des in Kapitel 3 vorgestellten Programmiermodells wird in Kapitel 4 auf konzeptioneller Ebene die Architektur einer verteilten virtuellen Welt definiert, welche das TGOS-Modell verwendet. In einem ersten Schritt werden die benötigten verteilten Datenstrukturen konzipiert und darauf aufbauend die für die virtuelle Welt erforderlichen Komponenten und deren Adaption im Rahmen des TGOS-Modells beschrieben. Besonderes Augenmerk wird auf den Entwurf von Konsistenzmodellen mit Hilfe des neuen Programmiermodells gelegt, wobei der Fokus auf den Einsatz und die mögliche Umsetzung transaktionaler Konsistenz innerhalb einer verteilten virtuellen Welt gelegt wird.

Die in Kapitel 4 vorgestellte Architektur wurde im Rahmen der Arbeit prototypisch implementiert und für die quantitative Analyse des Ansatzes im Rahmen von Messungen verwendet, die in Kapitel 5 vorgestellt werden.

Abschließend werden in Kapitel 6 die wesentlichen Neuerung des in der Arbeit verfolgten Ansatzes noch einmal rekapituliert, sowie ein Ausblick auf weiterführende Forschungsmöglichkeiten gegeben.

Kapitel 2

Architektur verteilter Welten

Dieses Kapitel soll eine Einführung ins den Aufbau, die Struktur und Konzeption virtueller Welten bieten. Da virtuelle Welten in sehr unterschiedlichen Ausprägungen vorkommen und deren Aufbau sich oftmals stark unterscheidet, kann im Folgenden nur ein konzeptioneller Überblick gewährt werden. Im Fokus liegen dabei Welten, die eine 3-dimensionale Repräsentation bieten.

2.1 Programmiermodelle

Betrachtet man die Entwurfsmuster, welche in verteilten Welten vorkommen, so weisen sie große Gemeinsamkeiten mit mehrspielerfähigen Computerspielen auf. Diese Nähe ist nicht zufällig, vielmehr stellen verteilte Welten im Wesentlichen nur einen Weiterentwicklung dieser Spiele dar, im Kern verwenden beiden die selben Mechanismen.

2.1.1 Basismodelle

Theoretisch betrachtet, kann man zwischen zwei verschiedenen Basismodellen bei der Konzeption von Verteilung im Umfeld verteilter Welten und Computerspiele unterscheiden. Dies ist zum einen das *befehlsorientierte* Modell und zum anderen das *aktualisierungsorientierte* Modell.

Befehlsorientierte Modelle können mit *aktiver Replikation* [Sch90] verglichen werden, da die einzelnen Knoten nicht einen neuen Zustand zugeschickt bekommen, sondern ihre Zustände mit Hilfe empfangener Befehle weiterentwickeln. Jedoch sind die Knoten beim befehlsorientierten Modell nicht nur Empfänger, sondern gleichzeitig auch Versender von Befehlen. Abbildung 2.1 illustriert schematisch die Vorgehensweise, unter Verwendung eines Client/ Server-Ansatzes. Knoten eins möchte bei ❶ eine Aktion (im Beispiel eine Bewegung) ausführen und schickt den Befehl(auch Kommando genannt) für diese Aktion an den Server. Dieser prüft bei ❷, ob die Ausführung einen

gültigen neuen Zustand erzeugt und schickt, nach erfolgreicher Prüfung, das Kommando zurück an alle Knoten, die dann bei ❸ die Bewegung ausführen. Knoten 1 führt den Befehl nicht direkt bei ❶ aus, sondern erst, wenn das Echo des Servers zum Zeitpunkt ❸ eintrifft. Bei diesem Modell ist es wichtig, dass alle Knoten zwingend immer alle Befehle ausführen und dies - im Falle von Abhängigkeiten - auch in der richtigen Reihenfolge tun. Die trivialste Abhängigkeit ist zum Beispiel die Position eines Avatars, falls Avatare miteinander kollidieren können. Möchten beide an die gleiche Position gelangen, ist es wichtig, wessen Kommando zu erst berechnet wird. Die Kommandos müssen deshalb einer totalen Ordnung unterliegen. Im Falle von Client/ Server-Systemen wird dies meist erreicht, indem der Server die Kommandowünsche seiner Clients sammelt und geordnet an diese zurück schickt. Neben der Reihenfolgenproblematik stellt das Modell auch eine hohe Anforderung an die numerische Stabilität [dJ77] der verwendeten Berechnungsmethoden, da alle Knoten ihre Datenbasis nur mit Hilfe der Kommandos von einem gültigen Zustand in den nächsten überführen. Treten bei der Berechnung nicht deterministische Ergebnisse oder Rundungsfehler auf, verlieren die Knoten ihre Synchronität und entwickeln sich unterschiedlich weiter. Dieses Verhalten wird auch als *Schmetterlingseffekt*, ein Begriff eingeführt von Edward Lorenz ([RA00], Seite 91ff), bezeichnet, falls ein kleiner Fehler enorme und unvorhersehbare Auswirkungen auf die Weiterentwicklung des Gesamtsystems haben kann. Eine der häufigsten Fehlerquellen für solche Nichtdeterminismen sind unsynchronisierte Zufallsvariablen, nicht initialisierte Kellervariablen und verlorene oder umsortierte Netzwerkpakete. Auch der Einsatz mehrerer Threads für die Berechnung eines neuen Zustandes kann zu diesem Effekt führen. Eine Schwierigkeit beim befehlsorientierten Modell besteht beim nachträglichen Beitritt eines Knotens, da dieser sowohl einen konsistenten Schnappschuss des aktuellen Zustandes benötigt, als auch sämtliche seit dem Schnappschuss aufgetretenen Kommandos in der korrekten Reihenfolge.

Abbildung 2.2 zeigt das aktualisierungsorientierte Modell, ebenfalls mit einem Client/Server-Aufbau und am Beispiel einer Positionsänderung. Zum Zeitpunkt ❶ hat der erste Knoten die Position seines Avatars verändert und möchte diese Änderung nun allen anderen teilnehmenden Knoten bekannt geben. Je nach gewähltem Netzwerkprotokoll schickt er dafür eine Aktualisierungs-Nachricht entweder an einen Server, der die Nachricht weiterverteilt, oder er versendet im Falle eines Peer-to-Peer Systems die Nachricht selbst an alle anderen Teilnehmer. Im Gegensatz zum befehlsorientierten Modell ist die Überprüfung des neuen Zustandes bei ❷ bezüglich seiner Gültigkeit deutlich schwieriger, da der Übergang erst wieder rekonstruiert werden muss, um eine Prüfung durchzuführen, sieht man von einfachen Tests ab (beispielsweise, ob die Position in einem gültigen Bereich liegt). Die empfangenden Knoten aktualisieren daraufhin bei ❸ die lokale Position ihres Avatars. Im Gegensatz zum befehlsbasierten Ansatz sind hier

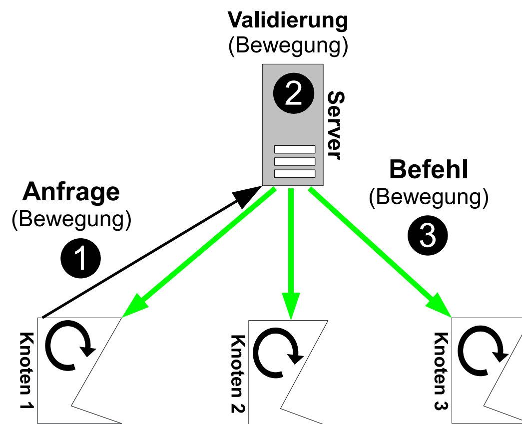


Abbildung 2.1: Befehlsorientiertes Modell

die Anforderungen an den Determinismus der Berechnungen geringer, da jeder Knoten Änderungen an der Zustandsmenge an alle weiteren Teilnehmer repliziert. Der bei befehlsorientierten Ansätzen problematische Schmetterlingseffekt tritt hier nicht in Erscheinung.

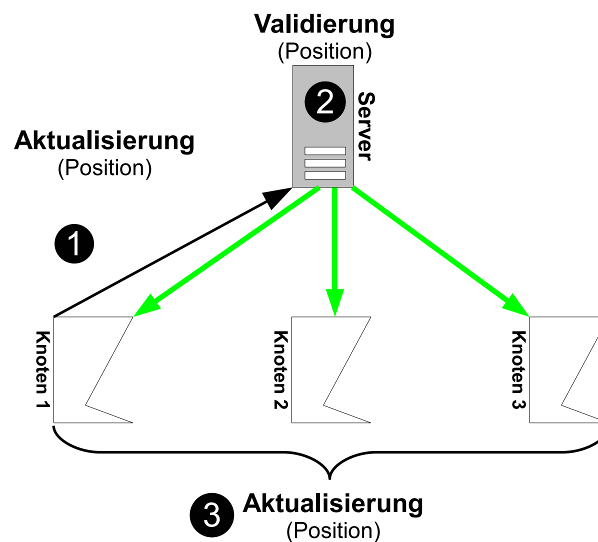


Abbildung 2.2: Aktualisierungsorientiertes Modell

Obwohl beide Modelle mit einem Client/Server-Ansatz vorgestellt wurden, sind beide Varianten auch mit einem P2P-Ansatz möglich. Das Aktualisierungs-Modell ist dafür am einfachsten geeignet, da dort lediglich alle Zustandsaktualisierungen direkt an alle beteiligten Knoten geschickt werden. Beim befehlsorientierten Modell muss, neben der korrekte Reihenfolge

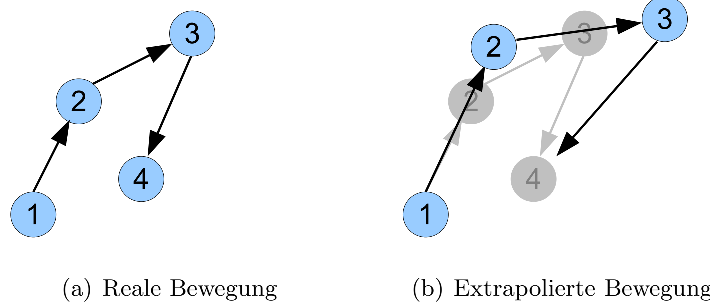
der Befehle, auch eine globale Ordnung derselben erfolgen, um beispielsweise bei der Kollisionsbestimmung zwischen zwei Avataren bestimmen zu können, wessen Befehl von allen Knoten zuerst ausgeführt werden soll.

2.1.2 Netzwerk-Protokolle

Die vorgestellten Basismodelle stellen die beiden theoretischen Extreme dar; im realen Einsatz kommen hauptsächlich hybride Formen der beiden vor, die sich im Netzwerk-Protokoll des Spiels, beziehungsweise der virtuellen Welt, manifestieren. Dabei ist es sehr vom Spielprinzip oder vom Verwendungszweck abhängig, welche Ausprägung gewählt wird. Ein typisches Beispiel wär ein Echtzeitstrategiespiel (engl. Real-time strategy, kurz RTS), in dem hunderte von Avataren gesteuert werden. Ein Aktualisierungs-Protokoll würde mit dieser großen Anzahl an Einheiten einen enormen Bandbreitenbedarf aufweisen, da die benötigte Datenrate $(AnzahlanTeilnehmer) * (AnzahlanEinheiten)*(AktualisierteDatenmenge)$ betragen würde. Nimmt man ein gepacktes Format für die Position und Rotation einer Einheit, indem jeweils 16-Bit pro Koordinate für die Position und Rotation reserviert sind, so müssen bei 100 bewegten Einheiten mindestens 1,2KB pro Teilnehmer übertragen werden. Deshalb würde man in diesem Fall ein reines befehlsorientiertes Modell bevorzugen. Eine bekannte Form stellen die *Lock-Step-Protokolle* dar, die gerade bei RTS-Anwendungen stark verbreitet sind. Ein Beispiel für ein solches Protokoll lieferte der Post-Mortem-Bericht zweier Entwickler von Age of Empires I & II [BT01].

Ego-Shooter dagegen setzen eher auf eine Kombination aus beiden Protokolle. Insbesondere die Position der Avatare wird hier via Aktualisierungs-Mechanismus übertragen. Dies geschieht jedoch nicht periodisch, sondern nur, falls der Avatar seine Bewegungsrichtung oder Geschwindigkeit ändert. Damit die anderen Knoten die Position eines Avatars berechnen können, wird zusätzlich auch der aktuelle Bewegungsvektor übertragen. Dadurch können die zukünftigen Positionen eines Avatars auf anderen Knoten extrapoliert werden; der Vorgang wird in der Fachliteratur auch als *Koppelnavigation* [PW02] (engl. Dead Reckoning) bezeichnet.

Abbildung 2.3 gibt einen kurzen Einblick in die grundlegende Funktionsweise von Dead Reckoning, welches auch zur Verdeckung der Netzwerklatenz (engl. latency hiding) eingesetzt wird. Teilbild 2.3(a) zeigt die Orginalbewegung des Avatars, während Teilbild 2.3(b) die Extrapolation auf einem Klienten zeigt. Der Mechanismus eignet sich sehr gut für gleichförmige Bewegungsänderungen mit einem kleinen Delta, wie man beim Schritt von 1 nach 2 sehen kann. Bei stärkeren Abweichungen wird oftmals versucht, das Überschießen, sprich das zu weit Berechnen einer Position, ausgelöst durch die Latenz, wie bei Position 2, zu kompensieren, indem man den Bewegungsvektor entsprechen anpasst. Bei starken Abweichungen oder sehr hohen Latenzen gibt es aber Sprünge, die vom Spieler als Lags (hinken; der



(a) Reale Bewegung (b) Extrapolierte Bewegung

Abbildung 2.3: Konzept des Dead-Reckoning

wahren Position hinterherhinken) wahrgenommen werden. Durch die periodische Aktualisierung der Position können Nicht-Determinismen nicht zu Schmetterlingseffekten führen und die Abweichungen, die zwischen zwei Aktualisierungen entstehen, sind meistens tolerierbar.

2.1.3 Ereignis-gesteuerte Programmierung

Allen Basismodelle gemein ist der ereignisorientierte Charakter, welcher sich auch aus der Netzwerkschicht ergibt, an die sie angelehnt sind. Der ereignisorientierte Charakter manifestiert sich in den oftmals komplexen Zustandsmaschinen, die hinter den Algorithmen stehen. Ereignisse lösen dabei einen Übergang von einem Zustand zum nächsten aus. Dieses Paradigma beschränkt sich aber nicht nur auf die interne Logik verteilter Welten, sondern erstreckt sich auch auf die Scriptingschnittstellen, welche für benutzergenerierte Inhalte zur Verfügung gestellt werden. Ein Beispiel ist die von SecondLife zur Verfügung gestellte *Linden Script Language* [Mel08], welche von ihrer Struktur und Syntax auf die Implementierung interaktiver Automaten getrimmt ist. Das Codebeispiel 2.1 zeigt einen Ausschnitt aus einem solchen LSL-Skript.

Jedes dieser Skripte besitzt eine Standardzustand, genannt *default*; in unserem Beispiel existiert ein weiterer Zustand, *state_touched*. Das Skript besteht also aus eine Liste von Zuständen, welche jeweils auf verschiedene Ereignisse reagieren können. Das Ereignis *state_entry* wird zum Beispiel beim Eintritt in das Ereignis gerufen, *touch_start* wenn ein Avatar das Objekt berührt. Jedes Ereignis kann neben einfachen Ausgaben und Berechnungen auch einen Zustandsübergang bewirken, wie im Beispiel der Aufruf *state touched*. Ein LSL-Skript beschreibt also einen Automaten, der durch definierte Ereignisse seine Zustände verändert.

Ein weiterer Aspekt, der durch die enge Kopplung des Programmiermodells mit der ereignis- und nachrichtenbasierten Programmierung ersicht-

```
default {
    state_entry() {llSay(0, "Hello, Avatar!");}
    touch_start(integer total_number){state touched;}
}

state touched {
    state_entry() {llSay(0, "Don't touch me!");}
}
```

Listing 2.1: Linden Scripting Language

lich wird, ist, dass bei Client/Server-Architekturen auch die Programme in einen Server- und Clientteil aufgespaltet sind. Dies ist auch bei Peer-to-Peer-basierten Ansätzen zu beobachten, da auch dort teilweise Client/Server-Strukturen vorkommen (siehe Abschnitt 2.3). Diese Aufspaltung sorgt für zusätzlichen Aufwand und Komplexität bei der Programmierung der Weltlogik.

2.2 Daten und Engines

Jeder Knoten, der an einer virtuellen Welt teilnimmt, benötigt eine Vielzahl unterschiedlichster Daten. Dazu zählen neben Mediendaten, wie 3D-Modellen, Texturen oder Tönen, auch Kommunikationsinformationen aus Text- beziehungsweise Voice-Chats, lokale Optimierungsstrukturen, sowie Informationen zur gegenwärtig angezeigten Szenerie. Abbildung 2.4 gibt einen Eindruck, welche Daten auf einem teilnehmenden Knoten im Allgemeinen vorhanden sind. Die spezifische Ausprägung unterscheidet sich von virtueller Welt zu virtueller Welt, am Grundprinzip ändert sich jedoch nichts.

2.2.1 Klassifikation

Gerade für virtuelle Welten ist es sehr interessant, die verwendeten Daten und Strukturen, welche sie benutzen, im Hinblick auf die Verteilung zu klassifizieren. Im Folgenden ist eine mögliche Klassifikation dargestellt:

Privat / Verteilt
: Private Daten sind nur auf einem Knoten verfügbar, während verteilte Daten auch von anderen Knoten genutzt werden können. Zum Beispiel sind Grafikdaten, die im Speicher der knotenlokalen Grafikkarte liegen, immer privat.

Online / Offline
: Bei dieser Klasse wird zwischen Daten, welche nur bei einer Verbin-

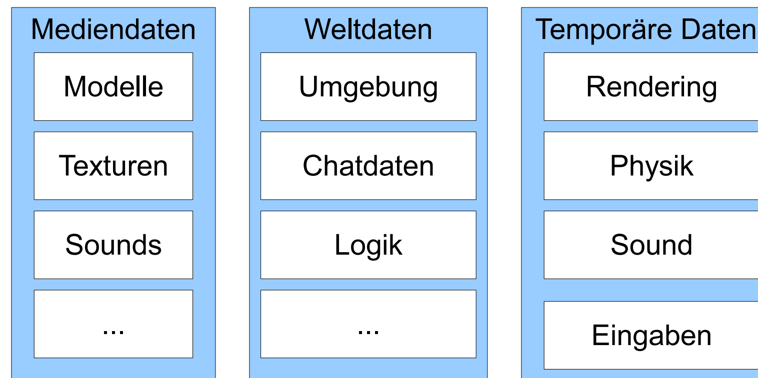


Abbildung 2.4: Daten eines Knotens

dung zur Welt vorhanden sind und Daten, welche auch ohne Verbindung auf dem lokalen Knoten verfügbar sind, unterschieden. Insbesondere Mediendaten, wie beispielsweise Texturen oder Modelle, können offline vorgehalten werden, um Ladezeiten zu verkürzen und die benötigte Netzwerkbandbreite zu minimieren. Gerade bei großen Online-Welten, mit mehreren Gigabyte an Mediendaten, sind Offline-Daten unerlässlich. Wie immer gibt es auch hybride Formen, dazu zählt zum Beispiel das Caching von SecondLife, welches im Prinzip alle benötigten Medien-Daten online anfordert, aber einen lokalen Zwischenspeicher bereithält, der mit den Caches von Webbrowsern vergleichbar ist. Dort werden alle heruntergeladenen Daten zwischengespeichert, um sie bei einer erneuten Anfrage nicht wieder über das Netz laden zu müssen.

Konstant / Variabel
Die Häufigkeit der Aktualisierungen ist ein weiteres Unterscheidungskriterium. Daten, welche sich im Laufe der Nutzung gar nicht oder nur äußerst selten ändern, werden als konstant bezeichnet, Daten mit hoher Aktualisierungsfrequenz als variabel.

Große / Kleine Änderungen
Daten lassen sich auch nach der Größe ihrer Aktualisierungsinformationen klassifizieren. Änderungen in Texturen oder Audiodateien benötigen größeren Datenmengen, während Aktualisierungen von Transformationen oder neue Befehle eher kleiner sind.

Die Klassifikation ist, hilfreich um beispielsweise eine Abschätzung der benötigten Bandbreite machen zu können oder um Einschränkungen der Funktionalität beschreiben zu können, die durch die Verteilung auftreten können.

Zum Beispiel ist es bei der ausschließlichen Verwendung von Offline-Mediendaten nicht mehr möglich, dynamisch zur Laufzeit neue Grafikinformationen hinzuzufügen oder bestehende Daten im laufenden Betrieb zu ändern.

2.2.2 Strukturen

Neben der Klassifikation sind insbesondere die Struktur und die Verknüpfung der Daten einer virtuellen Welt von Interesse. Im Folgenden sollen die gängigsten Strukturtypen kurz vorgestellt werden.

2.2.2.1 Szenengraphen

In objektorientierten Systemen finden häufig graph-ähnliche Strukturen Verwendung, um Beziehungen zwischen Objekten zu modellieren, beziehungsweise um einen effizienten Zugriff auf Daten zu gewährleisten. Diese Strukturen werden in der Literatur häufig als *Szenengraphen* bezeichnet und kommen meistens in Form von azyklisch gerichteten Graphen vor. Allgemein können Szenengraphen folgendermaßen definiert werden:

Definition 4: Szenengraph
Ein Szenengraph ist eine Datenstruktur in Form eines azyklischen und gerichteten Graphen, welche meist für die Visualisierung graphischer und vektorbasierter Szenen eingesetzt wird.

Die Entwicklung von Szenengraphen begann in den 90er Jahren, als Zusammenführung von 3D-Grafik und Objektorientierung. Einer der frühen Vertreter war Strauss [SC92], der einen Graphen mit Transformationshierarchie und Ereignisbehandlung entwickelte. Diese anfänglichen Szenengraphen waren rein für die Visualisierung 3-dimensionaler Szenen konzipiert und enthielten noch keine Verteilungsaspekte.

Ein weiterer sehr bekannter Vertreter der unverteilten Graphen stellt Java-3D dar, welcher standardmäßig in die Java-Laufzeitumgebung integriert ist. OpenSceneGraph [BO04] ist ein weiterer Szenengraph, der sich aber sehr speziell auf das effiziente Visualisieren großer Objektmengen spezialisiert hat.

Verteilte Szenengraphen entstanden Mitte der 90er Jahre und erweiterten die bestehenden Graphen. Eine der ersten Arbeiten in dieser Richtung präsentierten Blair MacIntyre und Steven Feiner [MF98], die ein DSM-System für die Verteilung des Szenengraphen nutzten.

Neben dem bereits erwähnten Ansatz von MacIntyre und Feinder existieren noch eine ganze Reihe weiterer **verteilter Szenengraphen**. Ein sehr bekannter Vertreter ist Blue-C [NLSG03], entwickelt von der ETH-Zürich, DIVE [FS98] oder auch Avaocado [Tra99].

Von einem einzelnen Szenengraphen zu sprechen, ist bei heutigen Systemen eher irreführend, da es dort meist mehrere graphartige Strukturen gibt, welche bestimmte Aspekte einer virtuellen Welt definieren. Zum Beispiel gibt es Graphen, welche speziell im Hinblick auf Visualisierungen optimiert sind, andere enthalten nur Informationen über Animations- beziehungsweise Bewegungsabläufe.

2.2.2.2 Enginestrukturen

Unter dem Begriff *Engine* versteht man - sowohl bei virtuellen Welten als auch bei Computerspielen - die Gesamtheit der Komponenten, welche die technischen Grundvoraussetzung bereitstellen, um die virtuelle Welt auf einem Knoten darzustellen und zu simulieren. Dazu zählen beispielsweise Grafik- und Physikengine, sowie allfällig bereitgestellte Werkzeuge zum Erstellen von Inhalten. Die Engine kann dabei als Werkzeug beziehungsweise Framework aufgefasst werden, mit welchem die Welt, beziehungsweise deren Inhalt, verwaltet und synthetisiert wird.

Jede Komponente verfügt intern über eine Vielzahl von Hilfsstrukturen, die für den Betrieb zwingend notwendig sind. Dazu zählen, neben gerätespezifischen Strukturen wie Grafik- oder Soundpuffer, auch Strukturen zur Zugriffsoptimierung beziehungsweise Ausführungskoordination. Insbesondere die Anforderungen an eine Renderengine erfordern umfangreiche zusätzliche Strukturen.

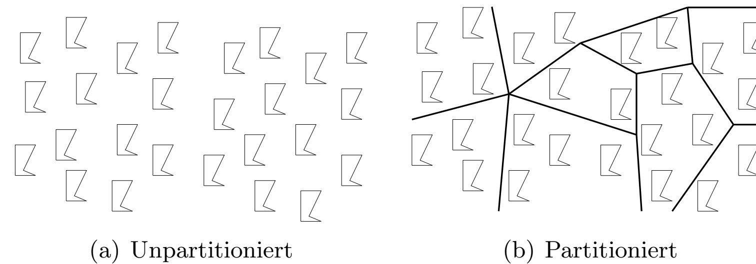
2.3 Lastverteilung

Betrachtet man die Werbeaussagen kommerzieller virtueller Welten, so wird behauptet, das tausende von Nutzern gleichzeitig online sein können und es wird suggeriert, dass jeder mit jedem interagieren könnte. Betrachtet man jedoch den Bedarf an Rechenzeit und Netzwerkbandbreite, so steigt dieser quadratisch mit der Anzahl der Nutzer und führt die Werbeaussagen ad absurdum. Um den Bedarf handhabbar zu machen, werden deshalb zwei in der Informatik sehr gebräuchliche Ansätze verwendet: Divide & Conquer und Lokalität.

2.3.1 Partitionierung

Die virtuelle Welt wird gemäß dem Divide & Conquer-Ansatz in Partitionen unterteilt, die *Areas-of-Interest* (kurz AoI) oder auch *Zonen* genannt werden und als abgeschlossene Einheit behandelt werden. Abbildung 2.5 demonstriert das Grundkonzept hinter dem Area-of-Interest-Ansatz, ein Vergleich verschiedener Varianten findet sich in [MZP+94]. Die Partitionierungsfunktion, mit der die Knoten in einzelne Bereiche unterteilt werden, kann dabei nach unterschiedlichen Gesichtspunkten gestaltet werden. Die häufigste

Funktion nutzt dabei die *virtuelle Lokalität* aus. Da sich Avatare meistens nur auf der Oberfläche einer virtuelle Welt bewegen (von Welten im Weltall abgesehen), genügt es oftmals, nur zwei Dimensionen der



(a) Unpartitioniert (b) Partitioniert

Abbildung 2.5: Area of Interest

Umgebungen zu betrachten, die sich mit einem einfachen Schachbrettmuster leicht in disjunkte Bereiche unterteilen lassen. Die Bereiche können dabei statisch [BF93, MZP+94] - beim Entwurf oder dynamisch - [BKV06] zur Laufzeit definiert werden. Vorteilhaft ist bei den dynamischen Varianten, dass sie auf Fluktuationen in der Benutzerdichte reagieren können, in dem sie den Partitionierungsgrad anpassen können. Eine hybride Variante definiert statisch alle sinnvollen Partitionen und legt dann, dynamisch zur Laufzeit, die Granularität der Areas-of-Interest fest.

Neben der Partitionierung besteht zusätzlich noch die Möglichkeit, zwischen einem Sicht- und einem Aktionsbereich [HSSB09] zu unterscheiden, wobei der Aktionsbereich eine Teilmenge des Sichtbereiches darstellt. Abbildung 2.6 demonstriert eine mögliche Unterteilungsform. Je nach Bereich wird eine stärkere, beziehungsweise schwächere, Konsistenz verwendet oder es findet eine Filterung der zu übertragenden Daten statt. Zum Beispiel sind die weiter entfernten Objekte in den grauen Bereichen meistens sehr klein, wodurch Bewegungsänderungen gar nicht oder nur für große Deltas wahrnehmbar sind und somit die Aktualisierungsrate dieser Objekte deutlich gesenkt werden kann. Alle Objekte in den schwarzen Bereiche sind für den Avatar nicht mehr sichtbar, weshalb Änderungen an diesen nicht übertragen werden müssen.

Eine weitere Variante, die bei serverbasierten Systemen möglich ist, berechnet anhand statischer Sichtbarkeitsinformationen der Szene (beispielsweise eine Wand, an dieser Stelle befindet sich der Spieler im Tunnel, etc.), welche anderen Teilnehmer ein Knoten überhaupt sehen kann und schickt nur Informationen über die Sichtbaren.

Eine Reihe von Veröffentlichungen beschäftigen sich mit den Charakteristiken des bei virtuellen Welten aufkommenden *Datenverkehrs*. Die von Kim [KCC+05] veröffentlichten Ergebnisse basieren auf einem MMORPG mit Client/Server-Ansatz und zeigen einen deutlich asymmetrischen Cha-

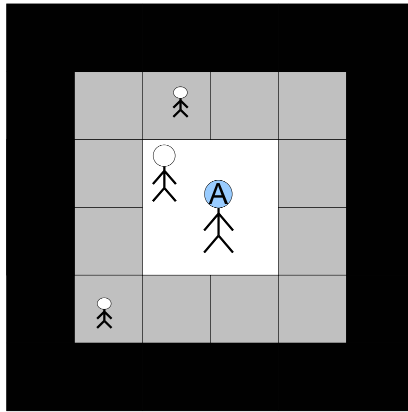


Abbildung 2.6: Unterteilung in Sichtbereiche

rakter der Datenströme, der für diese Art Systeme typisch ist. Eine weitere Abhandlung [FRS05] beschäftigt sich mit dem MMORPG Everquest 2. Kinicki beschreibt in [KC08], wie sich das Verhalten der virtuellen Avatare auf den zu beobachtenden Netzwerkverkehr auswirkt.

Eine weitere Arbeit, die sich sehr stark mit der Skalierbarkeit von virtuellen Welten beschäftigt, ist das Projekt Darkstar [Wal08], ein Labor-Projekt von Sun Microsystems. Darkstar setzt bei der Kommunikation zwischen Server-Cluster und Klient auf Nachrichten, bietet aber im Server-Backend die mit transaktionaler Konsistenz gesicherte Ausführung von verteilten Server-Threads. Mehr Informationen zum Projekt Darkstar sind in Kapitel 4 Abschnitt 4.6.1 zu finden.

2.3.2 Cluster & Grid

Neben der logischen Partitionierung, welche durch das Area-of-Interest-Management erfolgt, muss auch eine Aufteilung der benötigten Hardware-Ressourcen vorgenommen werden. Kommerzielle virtuelle Welten nutzen dabei große Server-Cluster beziehungsweise Grid-Strukturen, um die erforderliche Bandbreite und Rechenzeit bereitzustellen. Abbildung 2.7 skizziert den schematisch-logischen Aufbau eines Server-Clusters, wie er von den meisten kommerziellen Welten verwendet wird.

Jeder Cluster besitzt einen oder mehrere Login-Server, welche die Authentifizierung und Autorisierung der Teilnehmer übernehmen. Des Weiteren existieren mehrere Server, welche einzelne Zonen in der Welt simulieren sowie die Kommunikation mit den teilnehmenden Knoten realisieren. Alle Teilnehmer verteilten sich damit auf verschiedene Server. Im Idealfall sind alle Zonen gleichmäßig ausgelastet. Will ein Teilnehmer nun von einer Zone in die nächste wechseln, so kann es vorkommen, dass die Kapazität der Zone

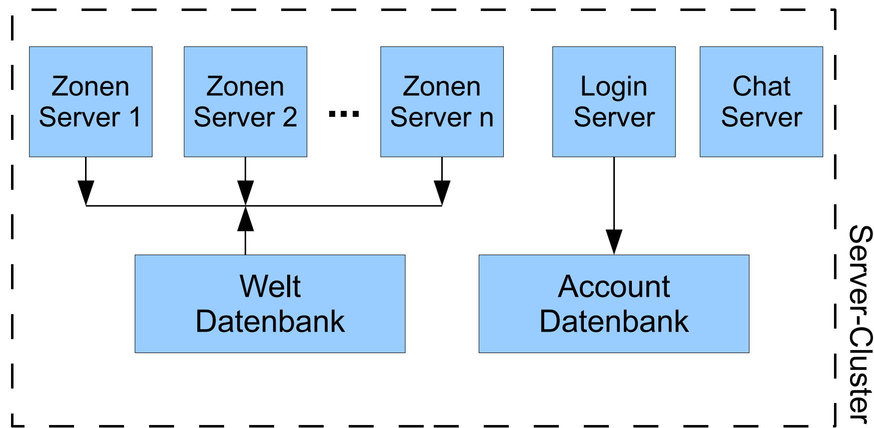


Abbildung 2.7: Schematischer Server-Cluster-Aufbau

erschöpft ist und ein Wechsel nicht stattfinden kann.

SecondLife ist dabei einer der wenigen Anbieter, der einen Einblick in den Aufbau und die Struktur seiner Welt bietet und der seinen Klienten als Open Source zur Verfügung stellt. Mehr zu der Struktur und dem Aufbau des SecondLife Grids findet sich im Wiki [Sec] von Linden Labs.

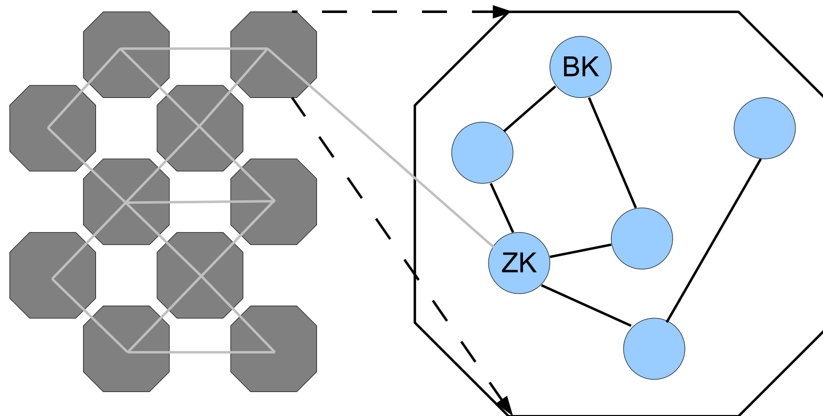


Abbildung 2.8: P2P-Overlay-Struktur

Peer-to-Peer-Ansätze, wie beispielsweise [HBH06], nutzen Overlay-Techniken, um eine Aufteilung in einzelne Zonen zu realisieren. Dabei sorgt ein Zonen-Koordinator (ZK) dafür, dass alle einer Zone zugeordneten Knoten immer eine konsistente Welt sehen. Abbildung 2.8 zeigt exemplarisch den Aufbau eines Peer-to-Peer-Overlay-Netzes, welches sich in verschiede-

ne Zonen gliedert. Die einzelnen Zonen-Koordinatoren sind untereinander verbunden und sorgen für die Kommunikation zwischen verschiedenen Zonen. Um Ausfälle des Zonen-Koordinators tolerieren zu können, existieren noch Backup-Koordinatoren (BK), welche im Fehlerfall übernehmen können. Knoten wechseln dynamisch zwischen verschiedenen Zonen, wobei der Zonen-Koordinator das Hand-Off übernimmt. Die Kommunikation kann dabei entweder zentral über den Zonen-Koordinator erfolgen oder über Gruppenkommunikationsmechanismen.

2.3.3 Multiple Welten

Selbst mit ausgefeilten AoI-Mechanismen und entsprechenden Grid-Systemen stoßen diese Systeme, abhängig von den Anforderungen an das Spielerlebnis, an ihre Grenzen. Einige kommerzielle Anbieter stellen deshalb nicht eine große Welt für alle Teilnehmer bereit, sondern mehrere parallele und disjunkte Welten. Die oberste Ebene stellt dabei eine Aufteilung nach geographischen Gesichtspunkten dar. Der Marktführer World of Warcraft gliedert sich zum Beispiel in drei Zonen [Act], Europa, Amerika und Asien, wobei jede dieser Zonen wiederum aus mehreren parallelen Welten besteht. Dabei sind die Welten nicht völlig isoliert, da es durchaus weltübergreifende Aktivitäten gibt. Abbildung 2.9 skizziert den Aufbau einer solchen hierarchischen Struktur.

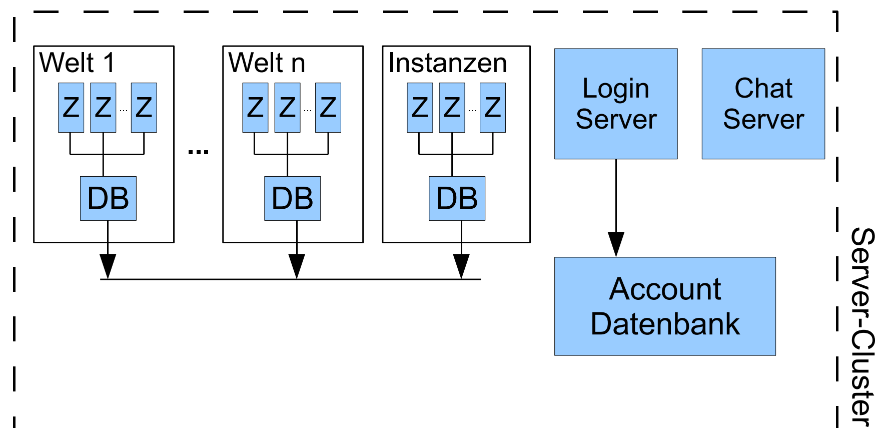


Abbildung 2.9: Aufbau mit multiplen Welten

Die wohl bekannteste weltübergreifende Aktivität stellen die sogenannten Instanzen dar, welche insbesondere in MMORPGs Verwendung finden. Dabei handelt es sich um abgeschlossene Umgebungen, welche insbesondere für Gruppenaktivitäten genutzt werden. Jeder Gruppe wird hierfür eine eigene private Instanz zugewiesen und die Gruppe kann dabei aus Teilnehmern

aus verschiedenen Welten bestehen. MMORPG verwenden dieses Konzept sowohl zur Lastverteilung, als auch um mehrere Gruppen von Spielern den selben virtuellen Ort mehrfach exklusiv betreten zu lassen.

2.4 Konsistenz

Die folgende klassisch Definition datenzentrierter Konsistenz soll im Folgenden verwendet werden:

Definition 5: Konsistenz
Die Konsistenz ist ein Vertrag zwischen parallel laufenden Prozessen, welche auf einem gemeinsamen (verteilten) Speicher operieren. Der Vertrag regelt dabei, welchen Werte eine Leseoperation zurückliefert, beziehungsweise wann und in welcher Reihenfolge Schreiboperationen sichtbar werden.

Betrachtet man nun verteilte Welten, so ist es unmöglich, eine Konsistenz für die gesamte Welt zu definieren. Vielmehr existieren innerhalb der Welt mehrere Konsistenzmodelle, die für verschieden Daten und Strukturen zum Einsatz kommen. Dabei kann die Konsistenz für ein Datum durchaus im Laufe der Zeit wechseln, beziehungsweise durch den Ablauf bestimmt sein.

Yu und Vahdat [YV02] beschreiben eine Klassifikation von Konsistenzmodellen, die auch numerische Abweichung innerhalb gelesener Werte berücksichtigt und sich somit sehr gut für verteilte Welten eignet, da dieser Umstand hier häufig auftritt. Abbildung 2.10 veranschaulicht dies anhand eines Beispieles. Avatar A wird dort von Knoten 1 gesteuert und bewegt sich an eine neue Position. Während die Position von Avatar A bei Knoten 2 innerhalb der durch die Konsistenz erlaubten Abweichung liegt, verletzt Knoten 3 diese.

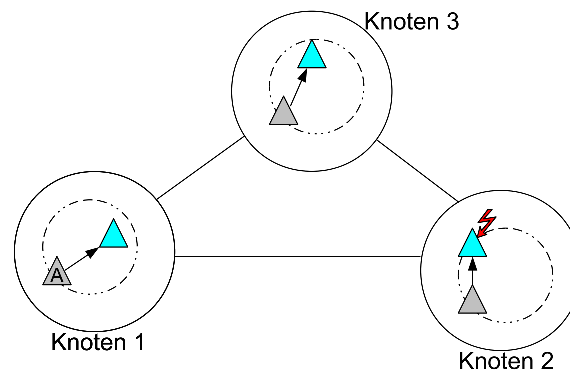


Abbildung 2.10: Konsistenz von Positionsinformationen

Eine weitere Eigenheit von verteilten Spielen und Welten ist der Um-

stand, dass das Verletzen eines Konsistenzkriterium durchaus toleriert wird, solange garantiert werden kann, dass nach einer gewissen Zeit ΔT wieder ein konsistenter Zustand erreicht wird. Das Beispiel der Avatarposition ist wiederum typisch für diesen Fall; die Verletzung der Konsistenz wird dabei vom Teilnehmer als *Lag* wahrgenommen, bei dessen Auftreten andere Avatare zu springen scheinen. Ist es nicht möglich, nach ΔT wieder einen konsistenten Zustand zu erreichen, so wird der Knoten meist von der virtuellen Welt getrennt und muss sich neu verbinden. Damit soll sichergestellt werden, dass der globale Zustand der Welt konsistent bleibt.

2.4.1 Konsistenzmodelle

Die Wahl der Konsistenzmodelle einer verteilten Welt wird stark vom Design und den Anforderungen der virtuellen Umgebung beeinflusst. Bei Ego-Shootern beispielsweise toleriert eine Spieler meist nur kleine Abweichungen, da die Position eines Spieler dort für den Spielerfolg sehr wichtig ist (Treffen eines Ziels), während beim Bummel durch eine virtuelle Einkaufswelt die Position der anderen Avatare weniger strengen Kriterien unterliegt. Eine Liste mit verwendeten Konsistenzmodellen zu präsentieren, wie es zum Beispiel für den gemeinsamen Zugriff auf verteilten Speicher möglich wäre, ist deshalb für verteilte Welten nur schwer möglich.

Konsistenz in verteilten Welten überschneidet sich stark mit Konsistenzansätzen, die in verteilten Spielen gebräuchlich sind. Zum Beispiel gibt es einige Post-Mortem-Artikel bekannter Spiele, die das verwendete Netzwerkprotokoll offen legen und erläutern. Eine der ersten Veröffentlichungen, die sowohl unterhaltsam zu lesen als auch sehr informativ ist, kam von P. Lincroft [Lin99]. Ein weiteres Beispiel für unterschiedliche Konsistenz stellt das SimMud-Projekt [BKH04] dar. Bei dessen Ansatz werden Positionsinformationen der Avatare durch einen P2P-Verbund der teilnehmenden Knoten ausgetauscht, mit entsprechend schwacher Konsistenz. Für den konsistenten Zugriffe auf Objekte in der Welt wird jedoch eine strikte Konsistenz benötigt, weswegen diese über eine Client/Server-Modell (bei SimMud Coordinater/Peer genannt) synchronisiert wird.

Einige Autoren beschäftigen sich auch mit theoretischen Überlegungen, welche Konsistenzmodelle speziell für verteilte Welten nötig, beziehungsweise geeignet wären. Ein Beispiel hierfür wäre [SSW+07], welches insbesondere Konsistenzanforderungen im P2P Umfeld beleuchtet.

In aktuellen verteilten Welten wird das Konsistenzmodell durch das verwendete Programmiermodell und das damit implementierte Netzwerkprotokoll bestimmt. Befehlsorientierte Programmiermodelle implementieren aufgrund der verwendeten Simulationsmechanik zumeist strikte Konsistenzmodelle, während aktualierungsbasierte Ansätze in der Regel schwächere Modelle umsetzen.

2.4.2 Konsistenzdomänen

Unter Konsistenzdomänen versteht man die Aufteilung des Konsistenzraumes in, wenn möglich disjunkte, Teilbereiche, welche die Konsistenz der ihnen zugewiesenen Daten regeln können, ohne dabei andere Konsistenzdomänen zu beeinflussen. Durch die Partitionierung soll die Anzahl der an einer Synchronisierung beteiligten Knoten reduziert werden, um sowohl den Datenverkehr als auch die durch die Synchronisierung auftretenden Latenzen zu verringern. Das im Abschnitt 2.3 vorgestellte Area-of-Interest-Management stellt eine solche Aufteilung für eine bestimmte Untermenge an Daten dar (in diesem Fall die Position und Orientierung des Avatars). Viele Operationen machen Synchronisierung zwischen verschiedenen Konsistenzdomänen nötig. Abbildung 2.11 zeigt ein typisches Beispiel für die Interaktion verschiedener Konsistenzdomänen in verteilten Welten. Avatar A und B sind in unterschiedlichen Bereichen der virtuellen Welt und A möchte B einen Gegenstand per virtueller Post zukommen lassen (virtuelle Postfächer sind beispielsweise in World of Warcraft oder anderen kommerziellen Welten zu finden). Das Postfach von B ist dabei in einer eigenen Konsistenzdomäne. Bei der Übergabe muss also erst die Konsistenzdomäne von As Inventar (grau gefärbt), in der sich das Objekt befindet, mit der des Postfaches von B kooperieren und dann Bs Inventar-Konsistenzdomänen mit der seines Postfaches. Ein weiteres Beispiel für Konsistenzdomänen stellen die verschieden Chatkanäle in einer virtuellen Welt dar.

2.5 Zuverlässigkeit

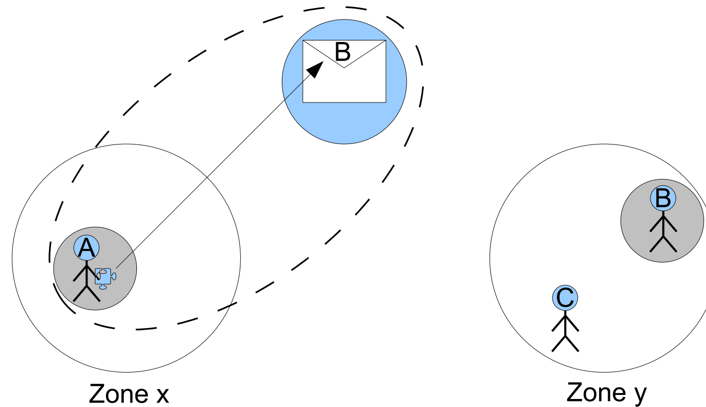
Zuverlässigkeit ist ein wichtiges Kriterium für verteilte virtuelle Welten. Die folgenden zwei Kategorien können dabei unterschieden werden:

Verfügbarkeit
: Verfügbarkeit beschäftigt sich mit der Ausfallwahrscheinlichkeit der Welt, wie robust sich die Welt als Ganzes gegenüber Hard- und Softwarefehlern verhält.

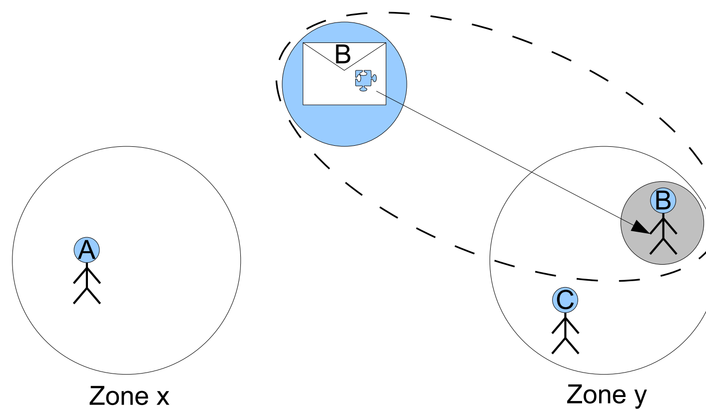
Persistenz
: Neben der Verfügbarkeit ist es für einen Teilnehmer einer virtuellen Welt wichtig, dass die Fortschritte, Erfahrungen und Konfigurationen seines Avatars bestehen bleiben, auch wenn es zu Fehlern kommt. Gerade bei MMORPGs ist es sehr wichtig, dass die Charakterentwicklung und der Spielfortschritt gespeichert werden.

2.5.1 Verfügbarkeit

Bei einem System mit tausenden beteiligten Knoten ist sowohl die Ausfallwahrscheinlichkeit eines Knotens, als auch das Auftreten von unerkannten



(a) A legt Objekt in Bs Postfach



(b) B holt Objekt aus seinem Postfach

Abbildung 2.11: Kooperation unterschiedlicher Konsistenzdomänen

Netzfehlern sehr groß. Dennoch muss eine verteilte Welt sicherstellen, dass der Ausfall eines Knotens nicht das gesamte System beeinflusst, insbesondere sollte der unerwartete Absturz eines teilnehmenden Knotens keine Auswirkung auf andere Teilnehmer haben. Genauso sollten Ausfälle in der Hardware der Hintergrunddienste keinen Ausfall der ganzen virtuellen Welt zur Folge haben. Dabei wird besonderer Wert auf die Robustheit dieser Systeme gelegt.

Auch hier gibt es starke Unterschiede zwischen Client/Server-basierten Systemen und Peer-to-Peer-Ansätzen. Gerade letztere wurden mit dem Ziel entwickelt, die Ausfallsicherheit zu erhöhen, in dem der Single-Point-of-Failure eliminiert wird.

2.5.2 Persistenz

Die Herausforderungen für Persistenz in einer virtuellen Welt sind, unter Verwendung der gegenwärtigen Programmiermodelle, stark von der verwendeten Netzwerkarchitektur abhängig.

Bei Client/Server-basierten Welten wird Persistenz meistens durch eine den Simulationsdiensten (Zonen) nachgelagerte Datenbankschicht realisiert, wie sie zum Beispiel in Abbildung 2.7 dargestellt wird. Ein Beispiel für einen solchen Ansatz stellt SecondLife dar, welches eine verteilte SQL-Datenbank für die Persistenz einsetzt.

Peer-to-Peer Systeme setzen, statt auf einen zentralen Speicherdienst, verstärkt auf Replikationsstrategien zwischen den teilnehmenden Knoten. In vielen Systemen ist deshalb für die Persistenz der Daten eine minimal verfügbare Knotenanzahl zwingend notwendig, da ansonsten Replikate verloren gehen würden. Zusätzlich zur Verfügbarkeit der Daten kommt hier auch noch das Problem der Integrität zum Tragen, da die Replikate auch auf Knoten residieren, die nicht unbedingt als vertrauenswürdig einzustufen sind.

Alle Modelle haben jedoch gemeinsam, dass die Häufigkeit, in der Daten persistent gespeichert werden, starken Einfluss auf die Gesamtleistung der Welt hat. Die dabei auftretenden Probleme und Fragestellungen ähneln den bei Checkpointingverfahren [TvS08] vorkommenden Problemstellungen. Durch die Verwendung von Area of Interest und Konsistenzdomänen kann die Granularität der „Checkpoints“ zwar eingeschränkt werden, dafür müssen aber Aktionen, die mehrere Konsistenzdomänen beeinflussen, gesondert behandelt werden.

2.6 Sicherheit

Sicherheit ist ein Thema, welches, gerade für kommerzielle Anbieter von virtuellen Welten, enorm an Bedeutung gewonnen hat. Insbesondere, da schon eine kleine Zahl böswilliger Teilnehmer das Erlebnis aller anderen stören kann. Hinzukommt, dass verteilte Welten, sei es in Form von Spiel-, Lern- oder Businesswelten, in zunehmendem Maße zu einem wichtigen Wirtschaftsfaktor werden. Wie wichtig virtuelle Welten gerade für den Wirtschaftsstandort Europa geworden sind, zeigt eine Veröffentlichung [NA08] der *European Network and Information Security Agency* (kurz ENISA), die sich ebenfalls sehr ausführlich mit Sicherheits- und Datenschutzaspekten in verteilten Welten auseinander setzt.

Neben klassischen Sicherheitsaspekten, wie Authentifizierung und Autorisierung beim Einloggen in die virtuelle Welt, sind insbesondere Anti-Cheating-Maßnahmen ein bedeutsamer Faktor. Anti-Cheating reicht dabei von der Erkennung und Vermeidung ungültiger Operationen, beziehungsweise Zustandsübergänge, bis hin zur Erkennung unerlaubten Nutzungs-

verhaltens. Ein sehr bekanntes Beispiel für solche unerlaubten Nutzungsarten stellen in kommerziellen virtuellen Spielwelten die sogenannten Farmbots [CDW07] dar, welche automatisch Ressourcen sammeln, um damit virtuelle Währung zu verdienen. Diese virtuelle Währung wird dann über Online-Auktionshäuser an andere Spieler verkauft, die so reales Geld gegen virtuellen Aufwand tauschen. Dadurch ist mehr virtuelles Geld verfügbar, was wiederum zu einer virtuellen Inflation führt, die starke Auswirkungen auf die virtuelle Wirtschafts- und Weltmechanik haben kann.

Eine Übersicht über aktuelle Cheating-Ansätze sowie deren Gegenmaßnahmen, sowohl für Client/Server-Architekturen als auch für Peer-to-Peer-Ansätze, wurde von Steven Webb [WS07] veröffentlichte.

Kommerzielle Systeme setzen, aufgrund der Sicherheitsproblematik, verstärkt auf Client/Server-Systeme, da dort Sicherheit oftmals einfacher zu bewerkstelligen ist, als bei reinen Peer-to-Peer-orientierten Systemen. Insbesondere die Identifikation von vertrauenswürdigen Knoten ist bei letzteren ein großes Problem. Als Programmiermodell finden insbesondere befehlsorientierte Varianten starke Verwendung, da sich damit sehr gut ungültige Operationen erkennen und verhindern lassen. Ein Server kann zum Beispiel a priori prüfen, ob ein vom Klienten empfangener Befehl gültig ist. Bekommt der Server, beziehungsweise die anderen Klienten, nur einen neuen Zustand, ist es deutlich schwieriger, diesen Zustand zu verifizieren. Reine Peer-to-Peer-Systeme setzen auch auf Abstimmungsverfahren, um byzantinische Fehler zu maskieren. Dies setzt aber voraus, dass die Mehrzahl der abstimmenden Knoten vertrauenswürdig ist.

2.7 Zusammenfassung

Sowohl bei den aktuell verfügbaren, als auch den zukünftig geplanten virtuellen Welten, stellt der Client/Server-Ansatz das vorherrschende Paradigma für die Verteilung dar. Bei kommerziellen Welten, wie beispielsweise World of Warcraft, Aion oder auch SecondLife, wird sogar ausschließlich dieses Paradigma benutzt. Bei den verwendeten Programmiermodellen wird meistens eine Variante des befehlsorientierten Modells gewählt, da dort sowohl striktere Konsistenzen deutlich einfacher zu implementieren sind, als auch die benötigte Netzwerkbandbreite geringer ausfällt. Allfällige Peer-to-Peer Ansätze finden sich hauptsächlich in der akademischen Welt oder in Open-Source-Projekten wieder, haben aber in der kommerziellen Welt bisher nicht Fuß fassen können (von der Peer-to-Peer-basierten Verteilung von Programmaktualisierungen abgesehen).

Ein sehr wichtiger Aspekt verteilter virtueller Welt ist, neben allfälliger Algorithmen zur Latenzverdeckung, insbesondere die große Anzahl gleichzeitig aktiver Nutzer und der daraus resultierende Bedarf an Lastverteilungs- und Partitionierungsmechanismen (siehe Abschnitt 2.5). Zusätzlich ist auch

die effiziente Nutzung der vorhandenen Bandbreiten sehr wichtig, um einerseits möglichst viele Nutzer gleichzeitig verarbeiten zu können und andererseits die Bandbreiten-Anforderungen an diese entsprechend gering zu halten.

Zusätzlich findet sich in gängigen Welten eine große Diversität an Interaktionsformen, die auch eine Vielzahl von unterschiedlichen Konsistenz- und Kommunikationsmechanismen benötigen. In zunehmenden Maße haben Teilnehmer einer virtuellen Welt auch die Möglichkeit, eigene Inhalte zu generieren und auszustellen (beispielsweise in SecondLife). Deshalb genügt es, im Gegensatz zu mehrspielerfähigen Computerspielen, nicht mehr, nur einige wenige Konsistenzmodelle durch ein optimiertes Netzwerkprotokoll bereitzustellen, vielmehr sollte eine virtuelle Welt einfache Mechanismen bieten, um neuen Konsistenzanforderungen gerecht zu werden, beziehungsweise dem Benutzer die Möglichkeit bieten, eigene Modelle zu implementieren.

Ausgehend von den, in diesem Kapitel, vorgestellten Anforderungen einer verteilten virtuellen Welt werden im folgenden Kapitel 3 die Unzulänglichkeiten der bestehender Ansätze beschrieben und darauf aufbauend die Anforderungen an ein verbessertes Programmiermodell erläutert und im TGOS-Programmiermodell theoretisch definiert.

Kapitel 3

Das TGOS-Programmiermodell

Ausgehend vom aktuellen Stand der Technik bei verteilten Welten wird deutlich, dass bei diesen üblicherweise die Kommunikations-, Konsistenz- und Logikkomponenten eng miteinander verzahnt sind.

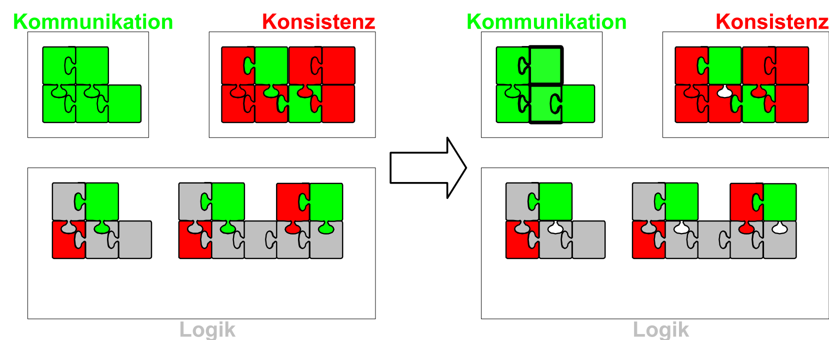


Abbildung 3.1: Probleme bei ungenügender Kapselung

Beispielsweise würden Änderungen an einem Teil der Kommunikationskomponente, in Abbildung 3.1 durch eine fehlende Verbindung zwischen zwei Puzzlestücken dargestellt, Anpassungen an allen anderen Komponenten nach sich ziehen. Insbesondere müssten alle Algorithmen, welche die Logik der Welt definieren, angepasst werden, falls sie sich auf die geänderten Teile der Kommunikationskomponente abstützen.

Um aber einzelne Komponenten austauschen zu können, ohne dabei das Gesamtsystem anpassen zu müssen, müssten diese über abstrakte Schnittstellen entkoppelt werden. Im Idealfall wäre es dadurch möglich, Teile der Kommunikationskomponente zur Laufzeit auszutauschen, ohne dabei den Betrieb der virtuellen Welt zu beeinträchtigen oder gar die für die virtuelle Weltlogik verwendeten Algorithmen verändern zu müssen. Beispielsweise

könnte so eine Client/Server-basierte Kommunikation beim Ausfall des Servers auf einen Peer-to-Peer-basierte Variante dynamisch umschalten.

Eines der Hauptziele des TGOS-Programmiermodells ist daher, eine geeignete Abstraktion zwischen den Komponenten zu definieren, die den Anforderungen der Austauschbarkeit genügt, aber dabei weder die Flexibilität noch die Erweiterbarkeit einschränkt.

Ein weiteres Ziel ist, die Entwicklung und Programmierung verteilter Welten zu vereinfachen. Insbesondere die Notwendigkeit, eigene Nachrichtenformate entwickeln zu müssen oder Daten und Strukturen selbsttätig serialisieren und über Rechnergrenzen hinweg konsistent halten zu müssen, stellt einen enormen Aufwand dar.

3.1 Nachrichten oder verteilter gemeinsamer Speicher

Bisher wurde die Kommunikations- beziehungsweise Verteilungskomponente nur in abstrakter Form beschrieben. Jedoch hat das verwendete Kommunikationsparadigma große Auswirkungen auf die Definition der abstrakten Schnittstelle. Seit den Anfängen der Computernetzwerke ist die Entwicklung verteilter Anwendungen auf Basis von Nachrichten und Paketen die gebräuchlichste Form. Neben dem rein nachrichtenbasierten Ansatz entwickelten sich Ende der 80er Jahre auch speicherbasierte Verteilungsverfahren. Bei diesen verteilten, gemeinsam benutzen Speicher-Systemen (engl. Distributed Shared Memory, kurz DSM) wird dem Programmierer die Illusion vermittelt, dass alle teilnehmenden Knoten einen großen gemeinsamen Speicher aufspannen, der wie bei einer Einzelplatz-Maschine genutzt werden kann, während das DSM-System transparent die paketorientierte Netzwerkkommunikation übernimmt.

In Abbildung 3.2 ist das Konzept des DSM in einfacher Form dargestellt. Die Knoten eins bis n spannen dabei einen globalen, virtuellen Adressraum auf, der von Anwendungen wie lokaler Speicher verwendet werden kann. Dabei ist aus Anwendungssicht nicht ersichtlich, ob sich die Daten an einer virtuellen Adresse auf dem lokalen oder einem entfernten Knoten befinden. Ein weiterer wichtiger Aspekt stellt die Granularität der verteilten Daten dar. Wie in Abbildung 3.2 zu sehen, ist der Beispiel-DSM aus vier Byte großen Einheiten aufgebaut. Obwohl eine Anwendung auf jedes einzelne Byte zugreifen kann, tauschen die Knoten immer vier Byte große Einheit untereinander aus. Ändert eine Anwendung ein Byte im DSM, so werden für diese Änderung immer vier Bytes übertragen. Die Granularität des DSM ist dabei für den Anwendungsprogrammierer transparent, kann sich aber durch Seiteneffekte wie zum Beispiel *False-Sharing* [ACRZ97] bemerkbar machen. False-Sharing tritt auf, falls Knoten für sie disjunkte Daten ändern, diese aber beispielsweise gemeinsam innerhalb einer Verteilungseinheit liegen. Ob-

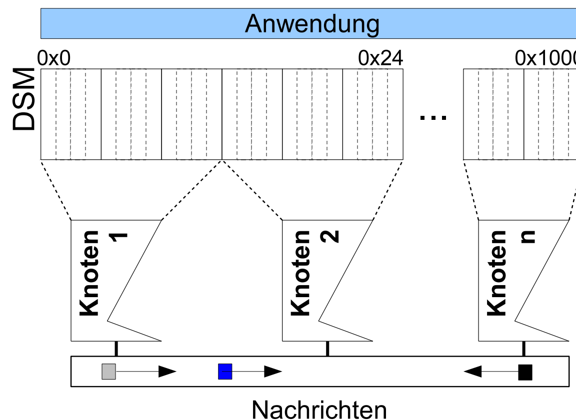


Abbildung 3.2: Konzeptioneller Aufbau eines DSM

wohl die Zugriffe logisch unabhängig sind, ergeben sich, bedingt durch die Granularität, Konflikte.

Eine weitere wichtige Eigenschaft neben der Granularität ist das verwendete Konsistenzmodell [Mos93], welches bei konkurrierenden Zugriffen durch mehrere Knoten für einen wohldefinierten Ablauf sorgt. Die ersten Systeme boten meistens nur ein Konsistenzmodell, spätere Systeme wie *Unify* [GYF95] oder Munin [BCZ90] boten verschiedene vorgegebene Modelle an. Die Konsistenz wird dabei pro Verteilungseinheit definiert, üblicherweise bereits bei der Allokation eines verteilten Speicherbereiches.

Eines der ersten Systeme stellt IVY [Li88] dar, welches von Kai Li in Princeton entwickelt wurde. IVY nutzt als Granularität der Verteilung 1KB-Seiten und bietet ein striktes Konsistenzmodell. Ein weiteres System ist Thor [LDS93], das von Barbara Liskov am M.I.T. entwickelte wurde und aus dem Bereich der objektorientierten Datenbanken hervor ging. Im Gegensatz zu IVY nutzte Thor für das Schreiben eine Objektgranularität und aus Cachinggründen für das Lesen eine Seitengranularität. Als Konsistenzmodell kommt bei Thor die transaktionale Konsistenz zum Einsatz. Ein anderer bekannter Vertreter der DSM-Systeme ist Treadmarks [ACD+96].

Eine aktuelle Weiterentwicklung der klassischen DSM-Systeme stellen die *in-memory data grids* (kurz IMDG) dar, welche sich speziell mit den Problemen und Herausforderungen im Grid-Umfeld befassen. Ein bekannter Vertreter der IMDG ist beispielsweise GigaSpaces [Sha07], welches insbesondere für den Einsatz in Firmen konzipiert wurde. Auch die schon bei Liskov vorgestellten Transaktionen werden im Rahmen von *distributed transactional memory* [DD09, RRCC10] wieder aufgegriffen und weiterentwickelt.

Die Verwendung eines DSM für die Verteilungskomponente einer virtuellen verteilten Welt erscheint auf den ersten Blick ein vielversprechender Ansatz zu sein. Insbesondere die starke Abstraktion der Netzwerkschicht

und das datenzentrierte Programmierbarmodell ermöglichen eine einfache Schnittstelle zwischen den eingangs erwähnten Basiskomponenten. Nachteilig jedoch ist, dass im Gegensatz zu nachrichtenbasierten Ansätzen, wo das Eintreffen einer Nachricht in der Regel einen Zustandsübergang signalisiert, Aktualisierungen im DSM, analog zu lokalem Speicher, transparent erfolgen. Dies bedeutet, dass Änderungen eines Knotens an Daten im DSM den anderen Knoten nicht explizit gemeldet werden, sondern diese nur durch kontinuierliches Vergleichen der sich ändernden Stelle eine Änderung erkennen können. Auch ist es bei DSM-Systemen für den Programmierer meist nicht oder nur schwer möglich, die Granularität der versendeten Daten, beziehungsweise die Anzahl und Größe der tatsächlich verschickten Pakete, zu ermitteln oder diese direkt zu beeinflussen. Gerade für Anwendungen, die Netzwerke mit sehr unterschiedlichen Charakteristiken verwenden, ist aber das Wissen und die Kontrolle der Pakete von entscheidender Bedeutung.

Eine Verteilungskomponente, die aus einem klassischen DSM besteht, scheidet aus den eben genannten Gründen aus. Jedoch könnte ein DSM-Ansatz, in Kombination mit den feingranularen Steuerungsmöglichkeiten und dem Ereignischarakter eines nachrichtenbasierten Ansatzes, eine Alternative zu den bisher verwendeten paketorientierten Ansätzen bieten. Die folgenden Punkte muss ein Konzept, welches Synergien aus den beiden Ansätzen zieht, bereitstellen:

Datenzentriert

Der Zugriff auf die verteilten Daten sollte, analog zum Modell des verteilten Speichers, transparent möglich sein. Die Verteilung der Daten soll dabei nicht implizit erfolgen, sondern explizit ausgelöst werden. Die Granularität, mit der Daten verteilt werden, muss durch die Anwendung vorhersagbar sein. Eine gegebenenfalls benötigte Serialisierung muss ebenfalls vom Modell automatisch bereitgestellt werden.

Variable Konsistenz

Das System muss die Möglichkeit bereitstellen, Konsistenzmodelle nicht nur dynamisch auszuwählen, sondern auch Neue einfach zu implementieren. Die Konsistenzmodelle dürfen dabei kein Teil der Kommunikationskomponente sein oder durch Änderungen an dieser beeinflusst werden, sondern sich nur der Funktionen der abstrakten Schnittstelle bedienen. Auch muss es möglich sein, die Konsistenz, im Gegensatz zu DSM-Systemen, nicht schon bei der Allokation eines Bereiches festlegen zu müssen, sondern diese dynamisch im Programmablauf festzulegen.

Ereignis gesteuert

Das System muss die Möglichkeit bieten, sich über Änderungen im Speicher, die durch andere Knoten vorgenommen werden, informieren

zu lassen. Als kleinste Einheit soll die verwendete Granularität dienen. Eine Anwendung sollte ferner die Möglichkeit haben, auch einen nachrichtenbasierten Programmierstil verwenden zu können.

3.2 Das TGOS-Programmiermodell

Das *Typed Grid Object Sharing* (kurz TGOS) stellt ein neues Paradigma bei der Realisierung verteilter Welten bereit, indem es die in 3.1 beschriebenen Anforderungen in einem neuen Programmiermodell umsetzt. TGOS basiert daher analog zu DSM-Systemen auf der Idee, dass Benutzer nicht in Nachrichten denken, sondern in Objekten, welche in unterschiedlichen Versionen auf verteilten Knoten vorhanden sind. Dabei ist es zulässig, dass verschiedene Knoten ganz unterschiedliche Versionen von Objekten haben. Ein kleiner Satz an Grundoperationen ermöglicht es einem Benutzer, Änderungen an einem Objekt zu verteilen, beziehungsweise Objekte, die noch nicht in seinem Objektraum vorhanden sind, zu laden. A priori wird von TGOS kein Konsistenzmodell definiert und umgesetzt, außer der grundlegenden Konsistenz, die sich durch die Replikationsschicht ergibt (beispielsweise Aktualisierungsreihenfolge). Einige Beispiele für mit TGOS implementierte Konsistenzmodelle finden sich in Abschnitt 4.3.2.

Das TGOS-Modell ist zwar nicht speziell für die Verwendung mit einer bestimmten Programmiersprache konzipiert, jedoch wurde beim Entwurf darauf geachtet, dass die Konzepte mit einer typsicheren objektorientierten Sprache, beispielsweise Java, umsetzbar sind. Zusätzlich muss die Sprache über Reflection-Mechanismen [Ibr92] verfügen, um beispielsweise dynamisch auf Klassendefinitionen zugreifen zu können.

3.2.1 Grundkonzept

Bevor die Funktionsweise von TGOS im Detail beschrieben wird, soll zuerst die Umgebung, in welcher TGOS zur Anwendung kommt, genauer erläutert werden. Abbildung 3.3 zeigt den konzeptionellen Aufbau eines TGOS-Systems. Wie eingangs beschrieben, besitzt jeder Knoten eine eigene Sicht auf die Objekte, welche auch unterschiedliche Versionen haben können. Dies ist in der Grafik durch die innerhalb der Objekte gezeigten Versionsnummern verdeutlicht. Lediglich die globale Replikationsschicht besitzt eine konsistente Sicht auf alle verteilten Objekte. Dabei ist zu beachten, dass die Sichten (auch Objektsichten) der einzelnen Knoten stark typisierte Objekte verwenden, während die Replikationsschicht konzeptionell nur mit binären Datenblöcken arbeitet, deren interne Struktur dieser nicht bekannt ist. Durch diese Abstraktion wird die Replikationsschicht unabhängig von dem in der Objektsicht verwendeten Objektmodell, beziehungsweise dessen Umsetzung. TGOS bildet nun das Bindeglied zwischen den Objektsichten und der Replikationsschicht. Durch die Basisoperationen kann ein Knoten via TGOS andere Sich-

ten und die Replikationsschicht aktualisieren. Wichtig ist dabei, dass TGOS nicht die Replikationsschicht selbst definiert oder implementiert, sondern lediglich die funktionalen und nicht-funktionalen Anforderungen festlegt. Alle Objekte besitzen einen global eindeutigen Identifikator, im Bild durch die Buchstaben dargestellt. Dieser Identifikator ist sowohl TGOS als auch der Replikationsschicht bekannt und wird für die Verbindung zwischen typsicherem Objekt in der Objektsicht und binärem Datenblock in der Replikationsschicht verwendet. Die Bereitstellung des Identifikators sowie dessen Format obliegt allein der Replikationsschicht. TGOS definiert lediglich die Vergleichseigenschaften, die ein Identifikator bieten muss. Dabei gilt, dass zwei Objekte, die in jeweils unterschiedlichen Sichten residieren, identisch sind, falls sie den selben Identifikator besitzen. Dies trifft in Abbildung 3.3 auf die Objekt **B** in Knoten 2 und n zu. Haben beide zusätzlich noch die gleiche Version, so sind sie Klone. Im Beispiel ist dies bei den Objekten **C** in Knoten 2 und n der Fall.

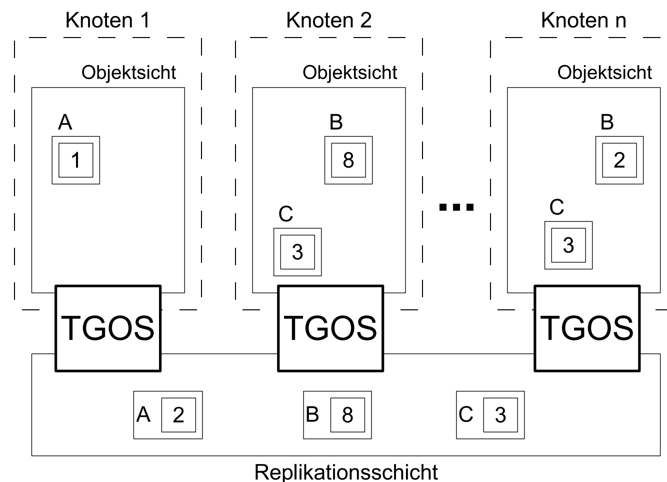


Abbildung 3.3: Sichten und Replikationsschicht

Zusätzlich ist TGOS auch für die Serialisierung von Objekten in binäre Datenblöcke und umgekehrt verantwortlich. Für die Deserialisierung spezifiziert TGOS eine Verfahren, welches, im Gegensatz zu üblicherweise in typsicheren Sprachen verwendeten Mechanismen (zum Beispiel Java Serialization [Kur97]), kein neues Objekt erstellt, sondern den Inhalt des bestehenden Objektes mit den aktualisierten Daten überschreibt. Durch diese *integrierende Deserialisierung* wird sichergestellt, dass alle Referenzen gültig bleiben. Mehr Informationen zur Referenzenbehandlung finden sich in Abschnitt 3.2.4. Ein weiterer Vorteil der integrierenden Deserialisierung ist die Möglichkeit, die neuen Daten nur partiell zu integrieren, sprich nur Teile des Objektes zu aktualisieren.

3.2.2 Typisierung

Eine wichtige Entwicklung im Hinblick auf objektorientierte Programmiermethodiken stellt die strenge Typisierung von Objekten dar. Im Gegensatz zu schwach getypten Sprachen wie C oder Pascal verbieten streng typisierte Sprachen das selbstständige Erzeugen von Zeigern sowie Zeigerarithmetik jeglicher Form. Durch die Typisierungen ist das Format eines referenzierten Objektes immer eindeutig bestimmt. Dies ist insbesondere für die im TGOS-Modell definierte Serialisierungsfunktionalität von Bedeutung, da diese Informationen über die Struktur der Objekte benötigt, welche in schwach getypten Sprachen gar nicht oder nur sehr eingeschränkt vorhanden sind.

Einen detaillierten Einblick bietet ein Artikel von Luca Cardelli und Peter Wagner [CW85], der mit zu den meist zitierten Abhandlungen über Typisierung, Abstraktion und Polymorphismus gehört.

3.2.3 Basisoperationen & Ereignisse

TGOS unterscheidet zwischen Ereignissen und Operationen. Operationen sind dabei TGOS-spezifische Funktionen, welche ein Benutzer auf Objekte in einer Objektsicht anwendet und die mit der Replikationsschicht interagieren. Diese wiederum kann bei anderen teilnehmenden Knoten Ereignisse auslösen, die sich aus den vorher beschriebenen Operationen ergeben. Im TGOS-Modell sind zwei Ereignisse definiert, die von der Replikationsschicht aus auftreten können und an die Objektsichten weitergeleitet werden.

Aktualisierungsereignis
: Dieses Ereignis wird ausgelöst, falls ein Objekt in der Replikationsschicht geändert wurde und diese Änderung an alle teilnehmenden Knoten weiterverteilt werden soll. Das Ereignis beinhaltet die aktualisierten Objektdaten.

Invalidierungsereignis
: Das Invalidierungsereignis tritt analog zum Aktualisierungereignis auf. In diesem Fall spezifiziert das Ereignis jedoch nur, welches Objekt geändert wurde, die aktuellen Objektdaten sind nicht Bestandteil des Ereignisses und müssen, falls gewünscht, separat nachgeladen werden.

Neben den Ereignissen sind in TGOS fünf Basisoperationen definiert, welche hier nun vorgestellt und erläutert werden sollen. Alle Operationen sind, falls nicht anders vermerkt, synchrone Aufrufe. Das heißt, ein Aufruf blockiert, bis seine Aufgabe vollständig abgeschlossen wurde. Da TGOS für typsichere Programmiersprachen entworfen ist, die in den meisten Fällen auch das Überladen von Methoden bereitstellen, können die TGOS-Operationen mit verschiedenen Signaturen auftreten.

Push

Mit Hilfe der *Push*-Operation kann ein Objekt an die Replikationsschicht übertragen werden. Sollte das Objekt dort bereits existieren, so wird der entsprechende Datenblock aktualisiert und ein Aktualisierungereignis bei alle anderen Knoten ausgelöst. Abbildung 3.4 veranschaulicht eine solche Push-Operation und ihr Zusammenspiel mit der Replikationsschicht.

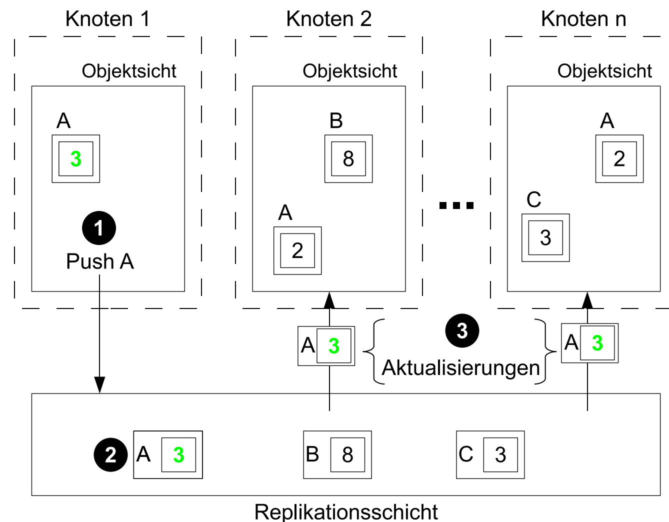


Abbildung 3.4: Push-Operation

Bei ❶ serialisiert TGOS die Objektdaten und übergibt der Replikationsschicht die Daten in binärer Form, welche an die global eindeutige ID gekoppelt sind. Die Replikationsschicht aktualisiert ihre Replik bei ❷ und löst dann bei ❸ ein Aktualisierungsereignis, welches eine Kopie der neuen Daten enthält, bei den übrigen Knoten aus. TGOS löst daraufhin ein Aktualisierungsereignis aus, welches vom Benutzer gegebenenfalls weiterverarbeitet werden kann.

Invalidate

Die *Invalidate*-Operation funktioniert analog zur Push-Operation, mit dem einzigen Unterschied, dass keine Aktualisierungsereignisse ausgelöst werden, sondern stattdessen Invalidierungsereignisse. Der Vorteil der Invalidierung liegt im deutlich geringeren Datenaufkommen, falls Knoten sich nur für das Auftreten des Änderungsereignisses interessieren, jedoch nicht für die Änderungen selbst. Auch kann es für Objekte sinnvoll sein, die nur sehr selten gelesen, aber häufig geschrieben werden.

Abbildung 3.5 stellt den Ablauf schematisch dar, der für die Schritte ❶

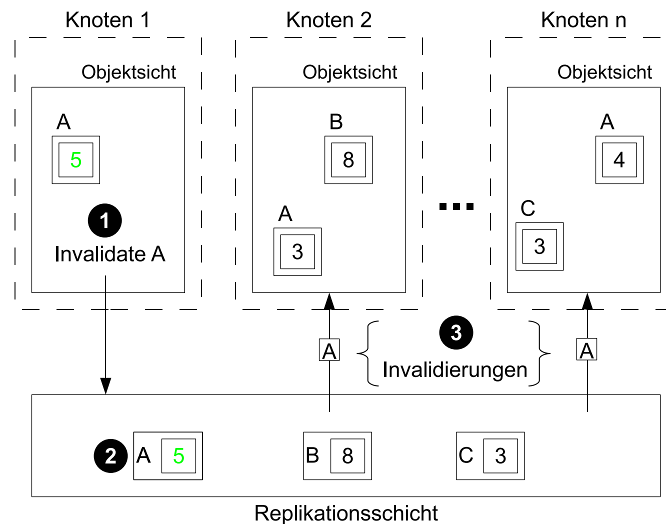


Abbildung 3.5: Invalidate-Operation

und ❷ identisch zur Push-Operation verläuft. Lediglich bei ❸ wird, anstatt der aktualisierten Daten, nur ein Hinweis auf das sich veränderte Objekt übertragen und TGOS löst dementsprechend ein vom Anwender verarbeitbares Invalidierungsereignis aus. Das invalidierte Objekt ist in der Replikationsschicht weiterhin gültig und kann innerhalb der Objektsicht verwendet werden.

Pull

Durch die *Pull*-Operation kann eine Sicht die aktuellste Version eines Objektes anfordern. Dieser Aufruf läuft synchron und blockiert, bis TGOS von der Replikationsschicht die angeforderten Daten erhalten und das Objekt aktualisiert hat. Abbildung 3.6 zeigt den Ablauf einer Pull-Operation, die bei ❶ mit einer Anfrage an die Replikationsschicht startet. Diese schickt daraufhin die neueste ihr bekannte Version bei ❶ an den anfragenden Knoten zurück, der diese dann bei ❸ entweder in ein bestehendes Objekt integriert oder es neu erzeugt. Des Weiteren ist zu beachten, dass diese Operation ausschließlich eine der Replikationsschicht bekannte Version des Objektes zurück liefert, allfällige neuere Versionen in den Objektsichten anderer Knoten, wie in der Abbildung bei Knoten n zu sehen, sind für die Replikationsschicht nicht sichtbar.

Sync

Die *Sync*-Operation fordert eine freiwillige Sperre für ein Objekt an. Diese freiwilligen Sperren verhalten sich ähnlich wie die bei UNIX-Dateisystemen gebräuchlichen *advisory locks* [MJLF84]. Abbildung 3.7

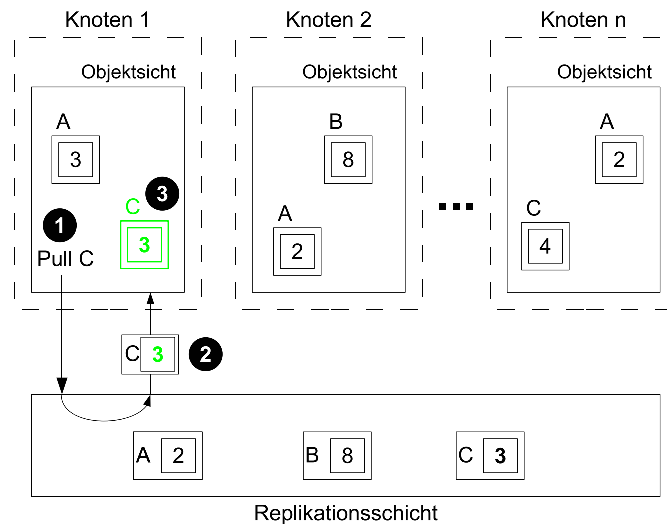


Abbildung 3.6: Pull-Operation

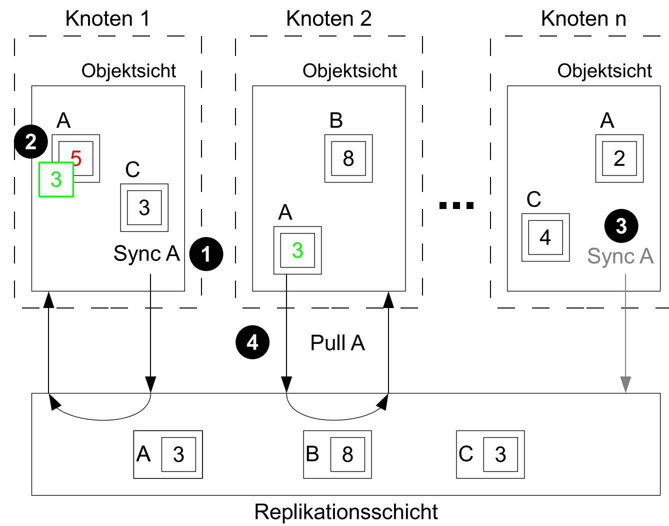
zeigt exemplarisch die Funktionsweise und den Ablauf einer Sperranforderung.

In 3.7(a) beginnt Knoten 1 bei ❶ mit der Anforderung einer Sperre, welche von der Replikationsschicht gewährt wird. Dabei wird automatisch die aktuellste Version des gesperrten Objektes geladen, was in diesem Beispiel zur Folge hat, dass die in der Objektsicht vorhandene neuere Version bei ❷ überschrieben wird. Die nachfolgende Sperranfrage ❸ von Knoten n blockiert, da Knoten 1 im Moment die Sperre hält. Die Pull-Operation ❹ von Knoten 2 wird jedoch ausgeführt, obwohl Knoten 1 immer noch die Sperre für Objekte A hält, da, wie eingangs erwähnt, die Sperren nur freiwilliger Natur sind. Knoten 2 könnte sogar eine Push-Operation auf das Objekt *A* ausführen.

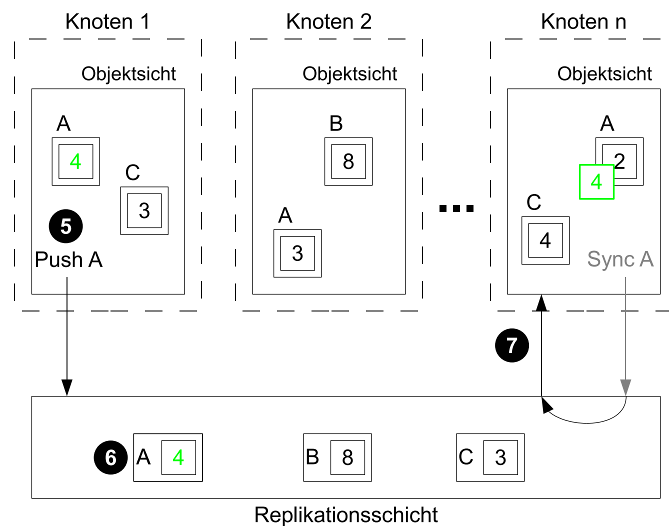
In 3.7(b) wird bei ❺ das Freigeben der Sperre mit Hilfe einer Push-Operation dargestellt. Sobald die Replikationsschicht die Anfrage ❻ bearbeitet hat, wird die Sperranforderung von Knoten n bei ❻ gewährt.

Order

Die *Order*-Operation ist die einzige asynchrone Operation und kann als eine Art Bestellvorgang verstanden werden, bei der TGOS die aktuellsten Daten für ein Objekt bei der Replikationsschicht bestellt. Abbildung 3.8 verdeutlicht den Ablauf einer solchen *Order*-Operation. Bei ❶ wird die „Bestellung“ aufgegeben, die dann bei ❷ im Rahmen eines Aktualisierungsereignisses eintrifft. Dabei wird, im Gegensatz zur Pull-Operation, weder ein bestehendes Objekt automatisch überschrieben,



(a) Sperren setzen



(b) Sperren aufheben

Abbildung 3.7: Sync-Operation

noch ein fehlendes neu erzeugt. Vielmehr obliegt es der Ereignisbehandlung, die Daten geeignet in die Objektsicht zu integrieren.

Diese Operation ist durch ihren asynchronen Charakter besonders für das Nachladen von Objektstrukturen im Hintergrund geeignet. Da die angeforderten Daten auf dem gleichen Weg wie Aktualisierungs- beziehungsweise Invalidierungsereignisse, welche durch Push- oder In-

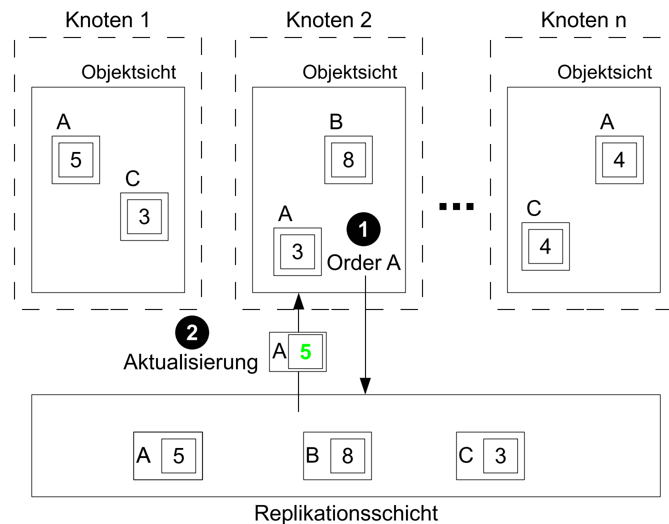


Abbildung 3.8: Order-Operation

validierungsoperationen ausgelöst werden, ankommen, muss eine korrekte Reihenfolge sichergestellt werden. Beispielsweise wäre es fatal, wenn das Ereignis einer Order-Operation eine Version zurückliefert, die älter ist, als eine Version des selben Objektes, welche durch eine reguläre Aktualisierungsnachricht empfangen wurde. Weitere Informationen zur Reihenfolgeproblematik von Ereignissen finden sich in Abschnitt 3.2.6.

Die beiden Operationen *Push* und *Invalidate* werden im TGOS-Modell auch als Schreiboperationen bezeichnet, da sie die Daten eines Objektes gewissermaßen in den Datenspeicher der Replikationsschicht schreiben.

3.2.4 Hüllenbildung & Referenzen

Wäre ein Benutzer gezwungen, jedes einzelne Objekt von Hand zu aktualisieren, so wäre dies in vielen Fällen sehr umständlich und zeitraubend. Das trifft insbesondere auf Java oder ähnliche Sprachen zu, da dort selbst einfachste Datenstrukturen als Objekte realisiert werden müssen, die in anderen Programmiersprachen, wie beispielsweise Pascal, als Datenverbund umsetzbar wären. Konzeptuell gehören diese *Daten*-Objekte zu dem sie referenzierenden Objekt.

Um den Umgang mit eben diesen Objekten zu vereinfachen, definiert TGOS einen dynamischen Hüllenbildungsmechanismus, mit dem solche logischen Objektstrukturen automatisch als ein einzelnes, verteilbares Objekt interpretiert werden können. Zu diesem Zweck unterscheidet TGOS zwi-

schen *Wurzelobjekten* und *Datenobjekten*; alle Objekte, die von einem Wurzelobjekt aus erreichbar sind und selbst kein Wurzelobjekt sind, werden als Datenobjekte bezeichnet und gelten als zu dieser Wurzel gehörend. Alle Objekttypen in der Objektsicht sind standardmäßig Datenobjekte; erst eine durch den Anwender erfolgte Markierung spezialisiert einzelne Objekte zu Wurzelobjekten. Die Markierung erfolgt dabei auf Typebene und kann entweder durch Vererbung oder mit Hilfe einer Schnittstellendefinition erfolgen, wobei die letztere Variante deutlich flexibler ist, da die meisten objektorientierten und typsicheren Sprachen zwar keine Mehrfachvererbung von Klassen zulassen, diese aber meist für Schnittstellendefinitionen erlauben. Abbildung 3.9 gibt eine Eindruck, wie sich die Hüllenbildung bei einfachen Objekten gestaltet.

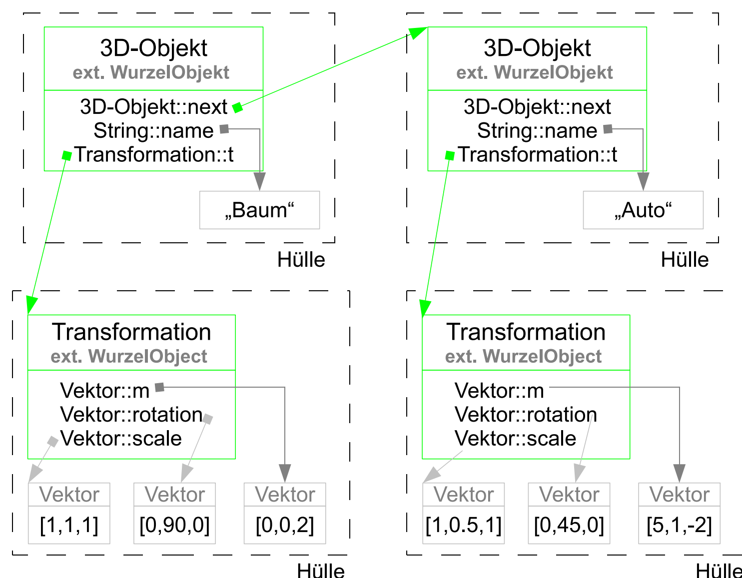
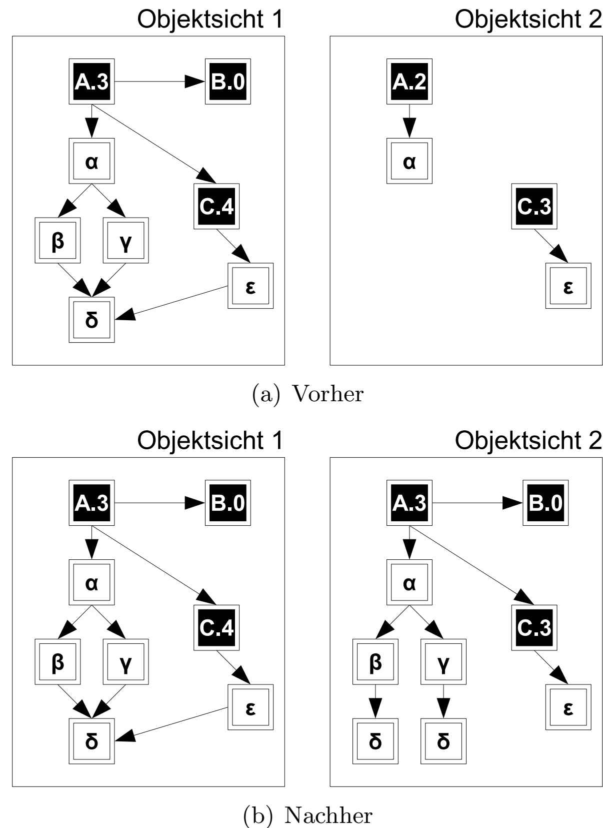


Abbildung 3.9: Hüllenbildung

Dabei findet die Markierung der Wurzelobjekte durch Vererbung statt; alle Objekte, die vom Typ *WurzelObjekt* erben, werden zu Wurzelobjekten. Im Umkehrschluss sind damit alle anderen Objekte Datenobjekte. Findet nun eine Push-Operation auf das linke *3D-Objekt* statt, so traversiert der Hüllenbildungsmechanismus alle Referenzen des Objektes und prüft, ob das dadurch referenzierte Objekt ein Wurzelobjekt oder ein Datenobjekt ist. Handelt es sich um ein Datenobjekt, so wird die Hülle des Wurzelobjektes um dieses Datenobjekt erweitert. Die Traversierung der Datenobjekte erfolgt dabei transitiv. Zeigt die Referenz hingegen auf ein Wurzelobjekt, dann wird dieses nicht zur Hülle hinzugefügt und auch nicht weiter untersucht. Die komplette Hülle definiert dabei auch die kleinste mögliche

Verteilungseinheit, da immer die komplette Hülle serialisiert wird. Da die Replikationsschicht nur mit binären Datenblöcken arbeitet, kann der global eindeutige Identifikator auch keine Einheit kleiner als ein Wurzelobjekt und seine zugehörige Hülle identifizieren. Die bereits definierten Basisoperationen und alle weiteren im TGOS-Modell definierten Operationen können deshalb ausschließlich auf Wurzelobjekte angewendet werden, da nur diese eindeutig global identifizierbar sind. Das Anwenden auf ein Datenobjekt ist weder zulässig noch möglich.



(a) Vorher

(b) Nachher

Abbildung 3.10: Globale und lokale Referenzen

Die Unterscheidung in Wurzel- und Datenobjekte wird daher von TGOS auch benutzt, um zwischen globalen und lokalen Referenzen zu unterscheiden. Referenzen auf Wurzelobjekte sind global gültig und werden von TGOS in den unterschiedlichen Objektsichten zwischen Objekten gleicher Version konsistent gehalten. Referenzen auf Datenobjekte hingegen sind nur innerhalb einer Objektsicht eindeutig. Wurzelobjekte werden deshalb im TGOS-Terminus auch globale Objekte genannt, die Datenobjekte entsprechend lokale Objekte.

Abbildung 3.10 verdeutlicht nochmals das Konzept der globalen und lo-

kalen Objekte, sowie die daraus resultierenden Besonderheiten der TGOS-Serialisierungsfunktionalität. Der Einfachheit halber wurde auf die Darstellung der Replikationsschicht, welche sich unterhalb der Objektsichten befindet, verzichtet. Abbildung 3.10(a) zeigt den Ausgangszustand, in der Objektsicht 1 alle drei Objekte verändert hat, aber diese Änderungen noch nicht via Push-Operation verteilt hat. Die Objekte mit schwarzem Hintergrund sind globale, jene mit weißem Hintergrund lokale Objekte.

Objektsicht 1 führt nun eine Push-Operation auf das globale Objekt A aus und Objektsicht 2 hat die durch das Aktualisierungsereignis erhaltenen Änderungen integriert. Wie man in Abbildung 3.10(b) sehen kann, sind die Referenzen zwischen den globalen Objekten konsistent geblieben, Objektsicht 2 hat das fehlende Objekt B automatisch nachgeladen und die Referenz von A nach B hergestellt. Bei den lokalen Objekten kann man insbesondere bei der Referenz auf das Objekt δ sehen, dass die Referenzen lokaler Objekte nicht konsistent gehalten werden müssen. Die TGOS-Serialisierung definiert, analog zur gängigen Parameterdefinition bei Methoden, Referenzen auf globale Objekte als Referenzparameter, Referenzen auf lokale Objekte als Wertparameter. Bei globalen Objekten wird als nur der global eindeutige Identifikator vermerkt, die Datenobjekte werden komplett serialisiert.

3.2.5 Terminologie

Für die Definition der Operationen wird eine eigenständige Terminologie verwendet, die in der nachfolgenden Tabelle 3.1 erläutert wird. Da es sich um ein sehr einfaches Modell handelt, wurde darauf verzichtet, eine bestehende Syntax zu verwenden, die zumeist deutlich ausdrucksstärker, aber auch komplexer ist.

Gemäß der Syntaxdefinition haben die TGOS-Basisoperationen damit die folgende Signatur:

```
Push (wObjekt)
Invalidate (wObjekt)
Pull (wObjekt)
Sync (wObjekt)
Order (wObject)
```

Zu beachten ist, dass die Operationen als Parameter immer ein Objekt vom Typ Wurzelobjekt, abgekürzt durch *wObjekt*, besitzen.

3.2.6 Ordnung

Obwohl TGOS den Benutzer von einer allfällig zu verwendenden Replikationsschicht abstrahieren soll, gibt es dennoch einige Punkte, die zu beachten sind. Insbesondere die Reihenfolge, in der Schreiboperationen sichtbar werden, muss für alle teilnehmenden Knoten eindeutig definiert sein. Die

Syntax	Semantik
$[Typ]$	Definierte eine Liste aus Elementen eines bestimmten Typs.
$\{Typ1, Typ2\}$	Definiert ein Tupel, welches aus den mit Kommata getrennten Typen besteht. Ein Tupel beinhaltet immer alle darin definierten Typen. Typen können auch mehrfach vorkommen.
Op (Typ1) → Typ2	Definition einer Operation Op. Parameter werden durch Kommata getrennt innerhalb der runden Klammern definiert. Eine Operation ohne Paramter besteht nur aus den runden Klammern. Ein optionaler Rückgabewert kann durch → und den Typ definiert werden.

Tabelle 3.1: Syntaxauflistung

Auswahl der Ordnung hat dabei großen Einfluss auf die mit TGOS implementierbaren Algorithmen. Im Wesentlichen kann zwischen totaler, partieller oder gar keiner Ordnung der Schreiboperationen unterschieden werden. Denkbar sind auch hybride Formen, bei denen via TGOS festgelegt werden kann, ob eine Schreiboperation einer Ordnung unterliegen soll oder nicht. Des Weiteren kann unterschieden werden zwischen der Ordnung der Schreiboperationen an die Replikationsschicht und der Ordnung der daraus resultierenden Aktualisierungs- bzw. Invalidierungsereignisse.

Standardmäßig setzt TGOS eine partielle Ordnung aller *Push*- und *Invalidate*-Operation sowie deren Ereignisse voraus. Dies bedeutet, dass sowohl die Replikationsschicht als auch alle teilnehmenden Knoten die Schreiboperationen eines Knotens genau in der Reihenfolge sehen, in der dieser sie ausgeführt hat. Diese Ordnung gilt auch für eine allfällige *Pull*-Operation des ausführenden Knotens, die direkt nach Schreiboperationen erfolgt.

Zusätzlich bietet TGOS die Möglichkeit, der Replikationsschicht einen Hinweis zu geben, dass die partielle Ordnung für die Ereignisse - und nur für die Ereignisse - nicht zwingend erforderlich ist. Zu diesem Zweck wird die Signatur der *Push*- und *Invalidate*-Operation durch einen Booleschen-Parameter folgendermaßen erweitert:

```
Push        (wObjekt, boolean)
Invalidate  (wObjekt, boolean)
```

Wird die Operation mit einem Wert *Wahr* aufgerufen, so wird eine partielle Ordnung der Ereignisse garantiert. Bei einem Aufruf mit *Falsch* wird zwar garantiert, dass die Schreibreihenfolge in der Replikationsschicht der Aufrufreihenfolge entspricht, jedoch können die Aktualisierungs- bezie-

hungsweise Invalidierungsereignisse in unterschiedlicher Reihenfolge bei den anderen Knoten eintreffen. Beispielsweise kann es vorkommen, dass ein Ereignis mit einer neueren Version vor einem älteren eintrifft.

3.2.7 Atomare Operationen

Neben der Notwendigkeit, kleine Aktualisierungs- beziehungsweise Invalidierungsoperationen zu größeren und effizienteren Einheiten zu akkumulieren, bietet die Verkettung von Schreiboperationen auch die Möglichkeit, eine Gruppe von Aktualisierungen oder Invalidierungen als atomare Einheit zusammen zu fassen. Damit ergibt sich eine einfache Möglichkeit, zusammengehörende Aktualisierungen zu bündeln, ohne auf Sperren angewiesen zu sein. Die Ausführung solcher verketteter Operationen erfolgt dann als eine atomar Operation, sprich, es werden entweder alle beteiligten Objekte verändert oder keines. Um dies zu erreichen, muss die Replikationsschicht garantieren, dass die Änderungen einer atomaren Operation nur dann gespeichert werden, wenn sie vollständig vorhanden sind. Zusätzlich müssen die resultierenden Ereignisse andere Knoten als untrennbare Einheit erreichen, um die Atomarität auch in diesem Fall zu gewährleisten.

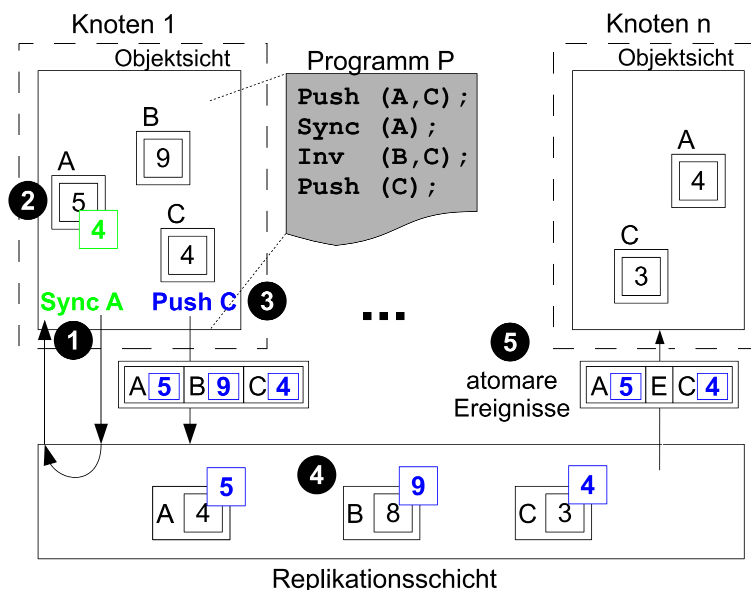


Abbildung 3.11: Verkettete Push-Operationen

Für die atomaren Operationen wird sowohl die Push-Operation als auch die Invalidate-Operation um eine zusätzliche Signatur, die einen zweiten Parameter erfordert, erweitert. Dieser Parameter gibt an, mit welchem Objekt das zu Schreibende verknüpft werden soll. Die Signatur der Operation sieht damit folgendermaßen aus:

```
Push        (wObjekt, wObjekt)
Invalidate  (wObjekt, wObjekt)
```

In Abbildung 3.11 wird die Funktionsweise einer solchen atomaren Operation dargestellt. Die in *Programm P* ausgeführte Operation **Push** (A,C) löst keinen Zugriff auf die Replikationsschicht aus, viel mehr serialisiert TGOS die aktuellen Daten des Objektes *A* und hängt sie an das Referenzobjekt, in diesem Fall *C*, an. Es können dabei beliebig viele Operationen mit einem Objekt verknüpft werden. Auch ist es, wie im Programm zu sehen, zulässig, Aktualisierungen und Invalidierungen zu mischen. Die darauf folgende Sperranforderung **Sync** (A) wird bei ❶ ausgeführt und überschreibt zwar bei ❷ A mit der Version der Replikationsschicht, die vorher serialisierten Daten von A bleiben davon aber unberührt. Sobald auf Objekt C bei ❸ die Schreiboperation **Push** (C) ausgeführt wird, werden auch die vorher serialisierten Daten von Objekt A in der Version 5 und B in Version 9 übertragen. Atomare Operationen werden im Hinblick auf Sperren genauso gehandhabt, als wären es einzelne Schreiboperationen, sprich die bei ❶ angeforderte Sperre von A wird durch die in der atomare Operation enthaltene Push-Operation auf Objekt A bei ❸ wieder freigegeben. Dadurch können auch mehrere Sperren gleichzeitig atomar freigeben werden. Die Änderungen der atomaren Operation werden dann bei ❹ ebenfalls atomar in der Replikationsschicht vorgenommen. Das durch die Schreiboperation ausgelöste Ereignis bei ❺ enthält die Änderungs- beziehungsweise Invalidierungsinformationen aller an der atomaren Operation beteiligten Objekte.

3.2.8 Verzeichnisdienst

Um bestehende Objekte in einem verteilten System zu finden, bedarf es eines Verzeichnis- oder Namensdienstes. Theoretisch wäre es möglich, einen Verzeichnisdienst mit Hilfe der gegebenen Basisoperationen zu implementieren. Für den Einsatz von TGOS, zum Beispiel in einem reinen Java-Umfeld oder ähnlichen Laufzeitumgebungen, ergeben sich dabei einige Probleme und Nachteile, die am Beispiel von Java erläutert werden.

Mit Standard-Java ist es sehr aufwendig, den Zugriff auf ein Objekt abzufangen, um dieses beispielsweise bedarfsgerecht nachzuladen. Eine Möglichkeit wäre, die Zugriffe auf ein Objekt abzufangen, indem statt des Objektes nur ein Proxy zur Verfügung gestellt wird, der die Aufrufe an das eigentliche Objekt weiterleitet und Zugriffe dabei protokolliert. Zusätzlich dürfen Objektvariablen nicht öffentlich sichtbar sein, da sonst ein Zugriff nicht erkannt werden kann. Stehen automatische Zugriffserkennungsmechanismen nicht zur Verfügung, müssten alle erreichbaren Objekte auf jeden Klienten vorhanden sein. Dies würde für einen Verzeichnis- oder Namensdienst bedeuten, dass alle darin gespeicherten Objekte auf allen angeschlossenen Knoten präsent sein müssten. Im Sinne der Ausfallsicherheit würde dieses Verfahren eine optimale Redundanz garantieren, jedoch den Speicher- und

Synchronisierungsbedarf der Klienten unverhältnismäßig in die Höhe treiben. Insbesondere bei großen Datenmengen ist dieser Ansatz durch Speicherlimitierungen der Klienten nicht umsetzbar.

Um dieses Problem zu lösen, definiert das TGOS-Modell eine eigene, schlanke Verzeichnisdienstschnittstelle, die nur aus den folgenden Operationen besteht:

```
Put (wObjekt, Name)
Get (Name) -> wObjekt
Get (Ausdruck) -> [Name]
```

Die *Put*-Operation hat als Parameter das zu speichernde Objekt und einen Namen, in Form eines hierarchielosen textuellen Identifikators, unter welchem das Objekt registriert werden soll. Für die Get-Operation sind zwei Varianten definiert. Die Erste bekommt den Name, nach dem gesucht werden soll, übergeben und liefert, falls gefunden, das registrierte Objekt zurück. Die zweite Variante bekommt einen regulären Ausdruck nach Ken Thompson [Tho68] übergeben und liefert eine Menge aller dem Ausdruck entsprechenden Namen des Namensraums zurück.

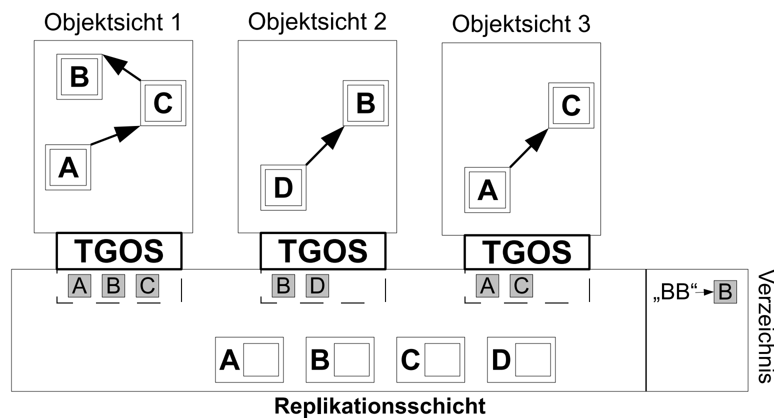
Analog zu den Basisoperationen, definiert TGOS kein Konsistenzmodell für den Namensdienst, außer das für *Put*-Operationen ebenfalls eine partielle Ordnung gelten muss.

3.2.9 Persistenz & Speicherbereinigung

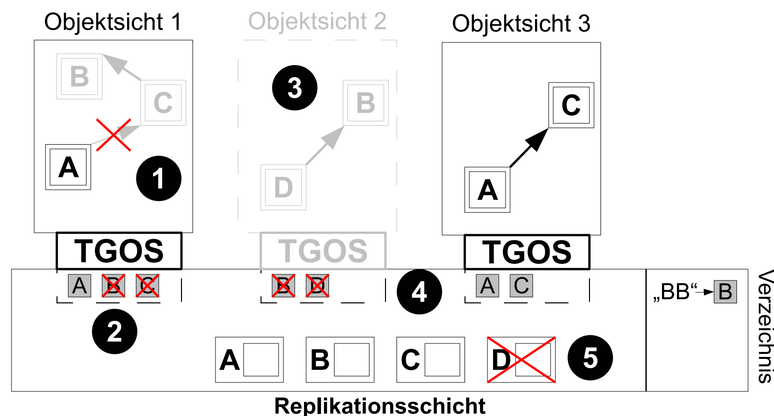
Im Gegensatz zu klassischen nachrichtenorientierten Ansätzen findet die Kommunikation bei TGOS über verteilte Objekte statt. Dabei muss jedes Objekt, anders als bei Nachrichten, global eindeutig identifizierbar und erreichbar sein, weshalb ein ebenfalls global eindeutiger Identifikator benötigt wird. Zusätzlich ist jedes verteilte Objekt innerhalb der Replikationsschicht durch einen binären Datenblock präsent. Dies trifft auch auf Objekte zu, die nur eine kurze Lebenszeit haben, wodurch die Notwendigkeit erwächst, nicht mehr benötigte Objekte freizugeben, um sowohl deren Identifikatoren als auch deren benötigten Speicherplatz zurück zu gewinnen. Da die Identifikatoren mit dem belegten Speicherplatz verknüpft sind, bedingt die Freigabe des einen auch die Freigabe des anderen.

Ein weiterer Aspekt stellt die Persistenz eines Objektes dar. Selbst wenn keine Objektsicht mehr mit der Replikationsschicht verbunden ist, sollten manche Objekte persistent bleiben und nicht automatisch freigegeben werden. Um dies zu gewährleisten, gibt es wiederum zwei mögliche Verfahren. Zum einen die manuelle Markierung persistent zu haltender Objekte durch den Benutzer oder die Einbeziehung des Verzeichnisdienstes, bei der die in diesem enthaltenen Referenzen auf Objekte zur Wurzelmenge einer eventuellen automatischen Freispeichersammlung hinzu kommen.

Durch die im TGOS-Modell definierte Teilung in Objektsicht und Replikationsschicht, wird auch eine mögliche Freispeichersammlung aufgeteilt. Innerhalb der Objektsichten wird eine stark getypte Programmiersprache vorausgesetzt, welche meistens bereits eine automatische Freispeichersammlung integriert, beziehungsweise die Voraussetzung bietet, eine eigene zu implementieren. Zusätzlich zur Freispeichersammlung müssen auch Mechanismen existieren, die es TGOS erlauben, den Lebenszyklus eines Objektes zu verfolgen; beispielhaft seien hier die Referenz-Objekte von Java genannt. Sind diese Vorgaben erfüllt, kann TGOS die in der Objektsicht verwendete Objektmenge bestimmen und mit der Replikationsschicht synchronisieren. Die Freispeichersammlungen laufen in diesem Fall lokal für jede Objektsicht und unabhängig voneinander ab.



(a) Ausgangszustand



(b) Nach Freigabe

Abbildung 3.12: Exemplarische Freispeichersammlung

Die Replikationsschicht hingegen muss eine globale Freispeichersammlung für ihre Replikate anbieten und kann dazu sowohl die Informationen darüber nutzen, welche Replikate TGOS für eine Objektsicht angefordert hat, als auch welche inzwischen als nicht mehr benutzt markiert wurden. Da der Basis-Verzeichnisdienst Teil der Replikationsschicht - beziehungsweise einer angegliederten Schicht ist - können die darin enthaltenen Referenzen ebenfalls verwendet werden.

Abbildung 3.12 zeigt exemplarisch den konzeptionellen Ablauf einer globalen automatischen Freispeichersammlung, wie sie für das TGOS-Modell definiert ist. Teilbild 3.12(a) zeigt die beteiligten Objektsichten sowie die Replikationsschicht, bevor irgendwelche Operationen ausgeführt wurden. Wie gut zu sehen ist, besitzt die Replikationsschicht Informationen darüber, welche Replikate TGOS angefordert hat und zum anderen einen angegliederten, beziehungsweise integrierten, Verzeichnisdienst, in dem ein Eintrag *“BB“* auf ein Objekt B verweist. Wie in Abbildung 3.12(b) bei ❶ zu sehen, wurde die Referenz von A nach B gelöscht, worauf hin TGOS durch die Freispeichersammlung der Laufzeitumgebung der Objektsicht mitgeteilt bekommt, dass die Objekte B und C freigeben werden. TGOS informiert bei ❷ die Replikationsschicht, die daraufhin die Liste der von Objektsicht 1 verwendeten Objekte anpasst. Da sowohl B und C noch durch die Objektsicht 2 und 3 referenziert werden, können diese nicht freigegeben werden. Wenn nun bei ❸ Objektsicht 2 beendet wird, so kann die Replikationsschicht bei ❹ ermitteln, dass keine Objektsicht mehr eine Referenz auf Objekt D oder B hält. Da Objekt B jedoch im Verzeichnisdienst referenziert wird, könnte nun bei ❺ nur Objekt D freigegeben werden.

Neben den Informationen, welche Objekte und daraus resultierend, welche Replikate von den Objektsichten benutzt werden, müssen zusätzlich noch die nicht von den Objektsichten verwendeten, aber durch die Replikationsschicht gespeicherten, weiteren Objekte/Replikate in Betracht gezogen werden. Jedes von diesen kann eine Referenz auf ein anderes Objekt enthalten. Da aber der Replikationsschicht die interne Struktur der Daten nicht bekannt ist, muss TGOS diese Information bereitstellen. Dies kann entweder bei jeder Aktualisierung eines Replikates in Form einer Liste von referenzierten Identifikatoren erfolgen oder TGOS stellt der Replikationsschicht eine Funktion bereit, die alle von einem Replikat referenzierten Replikate, beziehungsweise deren Identifikator, ermittelt. Erst durch die Kombination aus Objektsichtverfolgung und Referenzbehandlung der Replikate kann eine funktionierende globale Freispeichersammlung realisiert werden.

Das Beispiel stellt nur eine mögliche Variante zur Freispeichersammlung durch eine Replikationsschicht dar. Die genaue Implementierung ist wiederum der konkreten Ausprägung der Replikationsschicht überlassen. Lediglich der Informationsfluss zwischen dem Freispeichermechanismus der Objektsicht in Richtung Replikationsschicht, wie in Abbildung 3.12(b) bei ❷ zu sehen, als auch die Rolle des Verzeichnisdienstes werden durch TGOS defi-

niert.

3.2.10 Gruppenkommunikation

Wie in Abschnitt 2.3 beschrieben, sind Area-of-Interest-Algorithmen für eine verteilte virtuelle Welt unverzichtbar. Hauptziel dieser Algorithmen ist es, den Kreis der Empfänger einer Aktualisierung zu beschränken, um dadurch die Netzlast zu verringern. Obwohl nachrichtenorientierte Ansätze direkt auf Gruppenkommunikationsmechanismen der verwendeten Transportschicht, beispielsweise UDP oder TCP, zurückgreifen könnten, werden diese nur selten verwendet, da die Nutzung im Internet nicht oder nur eingeschränkt möglich ist. Die benötigten Mechanismen zur Steuerung des Nachrichtenflusses sind daher sowohl in der Kommunikations- als auch der Logikkomponente der Welt integriert und auf eine bestimmte Netzarchitektur zugeschnitten.

Auch im TGOS-Modell müssen Mechanismen vorgesehen werden, die eine Partitionierung des Objektraumes hinsichtlich des Empfängerkreises von Ereignissen erlauben. Da eine Anwendung im TGOS-Modell nur Zugriff auf die Operationen der Objektsicht hat, müssen auch die Gruppenkommunikationsmechanismen auf dieser Abstraktionsebene bereitgestellt werden. TGOS übernimmt damit auch für die Gruppenkommunkation eine Mittlerrolle zwischen Objektsichten und Replikationsschicht. Im Folgenden sollen sowohl die funktionalen, als auch die nicht-funktionalen Anforderungen an einen Gruppenkommunikationsmechanismus definiert und in das TGOS-Modell integriert werden.

3.2.10.1 Gruppenzugehörigkeit

Da, wie eingangs beschrieben, Gruppenkommunikationsmechanismen speziell für die spezifische Verteilung von Ereignissen genutzt werden, müssen diese daher mit einer Gruppe verknüpft, beziehungsweise einer zugehörig, sein. Dies gilt auch für die Empfänger der Ereignisse, welche im Falle des TGOS-Modells die Objektsichten sind. Die *Gruppenzugehörigkeit* ist daher eine Eigenschaft, die spezifiziert, welcher Gruppe man angehört. Ereignisse werden dabei nur an Objektsichten verteilt, die auch einer der Gruppen des Ereignisses angehören.

Bei der Definition der Gruppenzugehörigkeit der Objektsicht können zwei Ansätze unterschieden werden; ob sie pro Objektsicht definiert wird oder ob jedes Objekt selbst beschreibt, zu welchen Gruppen es gehört. Die letztere Variante, im Weiteren auch *objektbasierte Granularität* genannt, ist dabei eine Verfeinerung der ersten Variante, des *sichtbasierten Ansatzes.* Würde eine Objektsicht ein ihr vorher unbekanntes Objekt mit Hilfe der Pull-Operation laden, so würde die Gruppenzugehörigkeit der Objektsicht um die Gruppen des Objektes erweitert. Analog würde die Gruppen-

zugehörigkeit bei der Freigabe von Objekten verkleinert, falls es Gruppen ohne korrespondierende Objekte in der Objektsicht gibt.

Die Definition der Zugehörigkeit der Ereignisse kann dabei analog zur Objektsicht erfolgen. Beispielsweise könnte das Ereignis einer Push-Operation automatisch die Gruppen des auslösenden Knotens oder die Gruppen des betroffenen Objektes erben. Zusätzlich besteht noch die Möglichkeit, einen *flussbasierten Ansatz* zu verwenden, bei dem bei jeder Schreiboperation spezifiziert wird, welchen Gruppen die resultierenden Ereignisse zugehörig sind.

3.2.10.2 Gruppenverwaltung

Zusätzlich zur Bestimmung der Gruppenzugehörigkeit müssen auch geeignete Funktionen zum Beitritt, Austritt und zur Verwaltung von Gruppen durch TGOS definiert werden. Dabei stellt sich die Frage, welche Funktionen benötigt werden und in welchem Maße auf die bereits im TGOS-Modell integrierten Basisoperationen und Konzepte zugegriffen werden kann. Zur Verwaltung einer Gruppe werden die folgenden abstrakten Funktionen benötigt:

Identifikation
: Eine Gruppe muss sowohl innerhalb der Objektsichten als auch für die Replikationsschicht eindeutig identifizierbar und referenzierbar sein. Das Erstellen eines solchen global eindeutigen Identifikators kann entweder durch die Anwendung innerhalb der Objektsicht erfolgen oder automatisch durch TGOS oder die Replikationsschicht. Die automatische Generierung hat dabei den Vorteil, sowohl weniger fehleranfällig (beispielsweise automatische Vermeidung von Duplikaten), als auch komfortabler zu sein.

Erstellen
: Es muss eine Funktion existieren, mit der eine neue Gruppe angelegt und der Replikationsschicht bekannt gemacht werden kann.

Beitritt / Austritt
: Es müssen Funktionen existieren, die es ermöglichen, einer Gruppe beizutreten und gegebenenfalls diese auch wieder zu verlassen. Sobald der Gruppe beigetreten wurde, müssen alle entsprechenden Gruppenereignisse empfangen werden, nach einem Austritt sollten keine Ereignisse der Gruppe mehr auftreten.

Löschen
: Sollte eine bestimmte Gruppe nicht mehr benötigt werden, so muss es möglich sein, diese zu löschen, um eventuell reservierte Ressourcen der Replikationsschicht (beispielsweise UDP-Multicast Gruppen) wieder freizugeben.

Um die geforderte Funktionalität in TGOS zu integrieren, bietet sich, als schlanke und elegante Variante, eine Erweiterung des TGOS-Objektmodells an. Die bereits vorhandenen globalen Objekte bieten sich als eindeutiger Identifikator für eine Gruppe an, da jedes Objekt bereits einen eindeutigen Identifikator besitzt.

3.2.10.3 Gruppendefinition im TGOS-Modell

Eine Gruppe wird im TGOS-Modell nun durch ein beliebiges, globales Objekt realisiert, welches eine bestimmte, durch TGOS definierte, Schnittstelle implementiert. Lädt ein Knoten nun dieses *Gruppenobjekt* in seine Objektsicht, so tritt er automatisch der Gruppe bei und erhält ab diesen Zeitpunkt alle Ereignisse, die für diese Gruppe bestimmt sind. Wird das Gruppenobjekt durch die automatische Freispeichersammlung aus der Objektsicht entfernt, so ist der Knoten nicht länger Mitglied in der betreffenden Gruppe. Die Gruppe wird gelöscht, falls es keine Mitglieder mehr gibt und das Gruppenobjekt nicht im Verzeichnisdienst registriert oder anderweitig referenziert ist.

Zur Bestimmung des Empfängerkreises definiert das Modell eine knotenbasierte Granularität, da eine automatische objektbasierte Granularität zwingend für jedes Objekt auch einen Verweis auf die zugehörige Gruppe benötigen würde, was sowohl zu einem höheren Speicherbedarf, als auch Verwaltungsaufwand führt. Es steht einem Anwender jedoch frei, in seinen Objekten eine Referenz auf ein Gruppenobjekt zu integrieren. Dabei ist jedoch zu beachten, dass eventuell Ereignisse verloren gehen, da das Gruppenobjekt, abhängig von der Implementierung des TGOS-Modells, nach dem Datenobjekt geladen wird und in der Zwischenzeit keine gruppenspezifischen Ereignisse empfangen werden. Dadurch kann es beispielsweise vorkommen, dass mit einem veralteten Objekt gearbeitet wird, da das Aktualisierungsereignis auf Grund der fehlenden Gruppenzugehörigkeit nicht empfangen wurde. Ein möglicher Lösungsweg ist, das Objekt nach Erhalt der Gruppe ein zweites Mal zu laden, da dadurch das aktuellste, der Replikationsschicht bekannte Objekt geholt wird, welches eventuell verpasste Aktualisierungen enthält.

Für die Bestimmung der Gruppenzugehörigkeit der Ereignisse sind im TGOS-Modell zwei Varianten vorgesehen. Die Erste definiert eine Standardmenge an auszulösenden Gruppen, die bei jeder *Push*- oder *Invalidate*-Operation Verwendung findet. Es existiert eine Operation, um eine Liste an Gruppen zu setzen, sowie eine weiter, die die aktuell gesetzten Standardgruppen zurück liefert:

```
setGroups ([Gruppenobjekt])
getGroups () -> [Gruppenobjekt]
```

Bei der zweiten Variante werden die Schreiboperationen um einen zusätzlichen

Parameter erweitert, der eine Liste aller zu verwendenden Gruppen spezifiziert und die Vorgaben der ersten Variante überschreibt:

```
Push        (wObjekt, [Gruppenobjekt])
Invalidate  (wObjekt, [Gruppenobjekt])
```

Durch das Überladen der Methode können Gruppen bei einer atomaren Schreiboperation (siehe Abschnitt 3.11) nur für die finale Operation angegeben werden. Dies ist auch sinnvoll, da auch die ausgelösten Ereignisse eine atomare Einheit bilden und als ganzes bei den jeweiligen Empfängern ankommen müssen.

Der von einer Replikationsschicht zu erbringende Funktionsumfang wird in Abschnitt 3.2.12 näher erläutert.

3.2.11 Multi-Threading

Im Gegensatz zur parallelen Ausführung der TGOS-Operationen auf verschiedenen Knoten, bei der die Operationen auf disjunkten Objektsichten angewendet werden, arbeiten im *Multi-Threaded*-Fall mehrere Operationen gleichzeitig innerhalb der selben Objektsicht. Die Basisoperationen von TGOS ändern nicht nur die Daten der Replikationsschicht und lösen Ereignisse aus, sondern verändern auch die lokalen Daten der sie ausführenden Objektsicht (beispielsweise die Pull-Operation). Um einen deterministischen Ablauf zu gewährleisten, müssen daher sowohl die Operationen thread-sicher, als auch ihr Verhalten bei paralleler Ausführung klar definiert sein. Im Folgenden wird nun das Verhalten von TGOS nach Basisoperationen getrennt definiert.

Push

Im Sinne einer Ordnung von Multi-Threaded-Operationen verhält sich TGOS, als wäre jeder Thread ein eigener Knoten. Das heißt, die standardmäßig verwendete partielle Ordnung aller Schreiboperationen wird jeweils aus Sicht eines Threads garantiert.

Sync

Prinzipiell gibt es zwei mögliche Varianten, *threadspezifische* Sperren oder *knotenspezifische* Sperren.

Bei threadspezifischen Sperren wird garantiert, dass, falls zwei Threads eine Sperre anfordern, genau ein Thread diese Sperre besitzen kann. Push- oder Invalidate-Operationen auf das gesperrte Objekte, welche von anderen Threads ausgeführt würden, können die Sperre nicht aufheben. Nur eine Schreiboperation des sperrenden Threads kann diese freigeben.

Bei knotenspezifischen Sperren führt eine parallele Anforderung, analog zum threadspezifischen Ansatz, dazu, dass genau ein Thread die

Sperre erhält, jedoch würde eine Schreiboperation eines beliebigen anderen Threads die Sperre wieder aufheben.

In einer *hybriden* Variante wird die knotenspezifische Sperre dahingehend erweitert, dass, falls ein Thread, der nicht in Besitz der Sperre ist, eine Schreiboperation auf diese ausführt, so lange blockiert, bis der Thread im Sperrbesitz diese freigeben hat.

TGOS definiert als Standard das hybride Modell, da es die Intuitivität des threadspezifischen Ansatzes besitzt, ohne dass die Replikationsschicht threadspezifische Sperren bereitstellen muss.

Pull

Parallele Pull-Operationen sind aus Sicht von TGOS sowohl auf unterschiedliche Objekte als auch auf ein einziges Objekt unkritisch, auch wenn im letzteren Fall die Reihenfolge der Objektaktualisierungen nicht definiert ist.

3.2.12 Anforderungen an die Replikationsschicht

Ein wichtiger Aspekt des TGOS-Modells ist die Austauschbarkeit der Replikationsschicht. Um dies zu gewährleisten, definiert TGOS einige Anforderungen, welche eine kompatible Replikationsschicht erfüllen muss. Hierzu zählt neben der Bereitstellung global eindeutiger Identifikatoren, die den in Absatz 3.2.1 beschriebenen Anforderungen genügen müssen, auch eine partielle Ordnung von Nachrichten, beziehungsweise Replikationsoperationen. Im Folgenden werden sowohl die funktionalen als auch die nicht-funktionalen Anforderungen und deren Semantik erläutert.

3.2.12.1 Funktionale Anforderungen

Neben den Funktionen zur Replikation von Daten benötigt TGOS zusätzlich einen einfachen Verzeichnisdienst, sowie Möglichkeiten zur Verwaltung von Gruppenkommunikationsmechanismen. Die benötigten Funktionen sind deshalb nochmals unterteilt, in *replikationsspezifische*, *verzeichnisdienstspezifische* und *gruppenspezifische* Bereiche.

TGOS definiert einen binären, beliebig großen Datenblock, der durch einen Identifikator global eindeutig identifiziert und referenziert werden kann, als Basisobjekt der Replikation. Alle abstrakten Funktionen, welche für die Replikation definiert werden, haben als Parameter entweder den Identifikator des Datenblocks oder eine Tupel aus Block und Identifikator.

Erstelle Block erzeugt einen leeren Datenblock, der durch einen global eindeutigen Identifikator spezifiziert wird.

Block freigeben teilt der Replikationsschicht mit, dass der Knoten den betreffenden Datenblock nicht länger referenziert, sprich, das korrespondierende TGOS-Objekt wurde auf diesem Knoten freigegeben. Dies bedeutet jedoch nicht, dass der Block global gelöscht werden kann, da er durchaus noch auf anderen Knoten in Verwendung oder entweder durch den Verzeichnisdienst oder einen anderen Block referenziert sein kann.

Lade Block fordert einen oder mehrere Datenblöcke, spezifiziert durch die eindeutigen Identifikatoren, an. Die Funktion muss sowohl eine synchrone, als auch eine asynchrone Semantik ermöglichen.

Schreibe Block überträgt einen oder mehrere Blöcke an die Replikationsschicht. Werden mehrere Blöcke übertragen, so muss sowohl die Übertragung, die Speicherung, als auch die Aktualisierung jeweils atomar erfolgen. Des Weiteren muss spezifiziert werden können, ob die Schreiboperation eine Aktualisierungsnachricht oder eine Invalidierungsnachricht nach sich zieht. Zusätzlich sollte bei der Funktion eine Liste von Gruppen spezifiziert werden, die Ereignisse, welche durch die Schreiboperation ausgelöst werden, erhalten sollen. Wird keine Gruppe angegeben, müssen die Ereignisse an alle teilnehmenden Knoten verschickt werden. Zur Identifikation der Gruppen muss der gleiche Identifikatortyp verwendet werden, der auch bei den Datenblöcken zum Einsatz kommt.

Sperre Block fordert eine Sperre für einen Datenblock an. Der Aufruf muss synchroner Natur sein und liefert als Ergebnis die aktuelle Version des gesperrten Datenblockes zurück. Eine Unterscheidung in Lese- und Schreibsperre findet nicht statt.

Die genaue Implementierung der Gruppenkommunikations wird durch TGOS nicht spezifiziert, lediglich die Basisfunktionen zur Gruppenverwaltung sowie deren Verhalten aus Sicht des TGOS-Modells werden hier definiert. Analog zu den Datenfunktionen dient auch hier der eingangs beschriebene Identifikator als Weg zur eindeutigen Identifikation von Gruppen.

Gruppe beitreten informiert die Replikationsschicht, dass der Knoten einer durch einen Identifikator spezifizierten Gruppe beitreten möchte. Ist unter dem gegebenen Identifikator zum Zeitpunkt des Aufrufes noch keine Gruppe registriert, so wird diese erstellt. Die Replikationsschicht muss dabei Sorge tragen, parallele Zugriffe geeignet zu synchronisieren.

Gruppe verlassen ermöglicht es einem Knoten, sich aus einer Gruppe, welche wiederum über einen Identifikator identifiziert wird, auszutragen, um fortan keine weiteren, diese Gruppe betreffenden, Ereignisse

zu erhalten. Gruppen, die keine Mitglieder mehr haben und nicht durch den Verzeichnisdienst oder andere Blöcke referenziert werden, können von der Replikationsschicht gelöscht werden.

Für den Verzeichnisdienst definiert TGOS keine expliziten Konsistenzkriterien, außer dass, analog zur Replikation, eine partielle Ordnung für das Registrieren von Einträgen aus Sicht einer Objektsicht beziehungsweise eines Threads gelten muss.

Registrieren registriert einen Replikat-Identifikator unter einem alphanumerischen Namen im Verzeichnisdienst. Der Aufruf mit einem leeren oder ungültigen Identifikator löscht den vorhandenen Eintrag.

Suche stellt eine Anfrage an den Verzeichnisdienst, mit dem Namen des gesuchten Eintrags als Parameter. Im Erfolgsfall wird der gespeicherte Identifikator zurückgeliefert, im Fehlerfall ein in der verwendeten Programmiersprache definierter *NULL*-Wert.

Suche Namen startet eine Suche im Namensraum und liefert alle registrierten Namen zurück, auf die das Muster passt. Es werden dabei keine Identifikatoren zurückgeliefert, sondern alle Namen, welche dem Suchmuster entsprechen.

Neben den Funktionen, die durch TGOS gerufen werden können, muss eine Replikationsschicht auch zwei Ereignisse zur Verfügung stellen, die von TGOS empfangen werden können.

Aktualisierung
Das Ereignis wird ausgelöst, falls innerhalb der Replikationsschicht ein Replikat aktualisiert wurde und eine Aktualisierungsbenachrichtigung gewünscht wurde. Das Ereignis beinhaltet sowohl den Replikat-Identifikator, als auch den aktualisierten Datenblock.

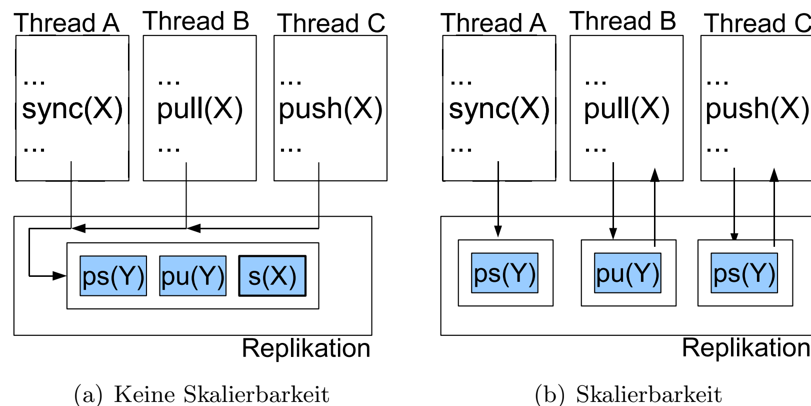
Invalidierung
Das Invalidierungsereignis verhält sich analog zum Aktualisierungsereignis und wird ausgelöst, falls statt der Aktualisierungsbenachrichtigung eine Invalidierungsbenachrichtigung spezifiziert wurde. Das Invalidierungsereignis beinhaltet nur den Replikat-Identifikator des aktualisierten Replikats.

Die hier vorgestellten Funktionen stellen die kleinste Menge an benötigten Funktionen dar. Zusätzliche Funktionen zur Behandlung von Fehlersituationen, Verbindungsauf- und Abbau, Initialisierung der Replikationsschicht und dergleichen wurden nicht behandelt, da sie für das theoretische Verständnis des TGOS-Modells nicht relevant sind.

3.2.12.2 Nichtfunktionale Anforderungen

Zu den drei wichtigsten nichtfunktionalen Anforderungen gehören die Persistenz, Robustheit und Skalierbarkeit der Replikationsschicht. Es muss sichergestellt sein, dass der Ausfall eines Knotens weder die Integrität noch die Verfügbarkeit der gespeicherten Daten schwächt, noch dass es zu Verklemmungen kommt. Dies könnte beispielsweise auftreten, falls der abgestürzte Knoten eine Sperre hielt. Das TGOS-Modell verlangt in diesem Fall, dass die gehaltene Sperre durch die Replikationsschicht freigegeben wird und der nächste Knoten, welcher auf die Sperre wartet, diese erhält.

Neben der Skalierbarkeit einer Replikationsschicht in Bezug auf Anzahl der teilnehmenden Knoten und Replikate, ist auch die Skalierbarkeit der parallelen Anfragen auf einem Knoten von besonderem Interesse. Wie bereits erwähnt, sind die meisten TGOS-Basisoperationen synchroner Natur, das heißt, sie blockieren nach einer Anfrage an die Replikationsschicht, bis diese das Ergebnis zurückliefert. Werden nun mehrere Anfragen durch unterschiedliche Threads an die Replikationsschicht gestellt, wie in Abschnitt 3.2.11 beschrieben, so muss sichergestellt sein, dass diese mit parallelen und eventuell blockierenden Anfragen zurecht kommt.



(a) Keine Skalierbarkeit (b) Skalierbarkeit

Abbildung 3.13: Knotenlokale Skalierbarkeit der Replikationsschicht

Abbildung 3.13 verdeutlicht die Problematik des Multi-Threaded-Ansatzes für die Replikationsschicht. In Teilbild 3.13(a) wird die nicht-parallele Situation dargestellt, bei der alle Anfragen durch die Replikationsschicht in einer Warteschlange verwaltet werden. Die Sperranfrage von Thread A, welche beispielsweise durch eine bereits bestehende Sperranfrage nicht gewährt werden kann, blockiert dadurch alle Anfragen der anderen Threads. In Teilbild 3.13(b) kann die Replikationsschicht mehrere Anfragen parallel bearbeiten und die Threads B und C werden durch die Sperranfrage von A nicht weiter blockiert. Wie viele thread-parallele Zugriffe eine Replikationsschicht verarbeiten kann, wird im TGOS-Modell nicht spezifiziert, sondern es obliegt

dem Programmierer, eine für seine Anwendung geeignete zu verwenden.

3.2.13 Ereignisse & Serialisierung

Bisher wurden Funktionen vorgestellt, die es erlauben, Objekte in der Replikationsschicht zu aktualisieren und Ereignisse bei anderen Knoten auszulösen. Die Signatur der Ereignisse oder die Art und Weise, wie die von ihnen gemeldeten Aktualisierungen, beziehungsweise Invalidierungen, in die Objektsicht integriert werden können, wurde bisher nicht definiert. Zu diesem Zweck wird auf die im TGOS-Modell in Abschnitt 3.2.1 spezifizierte Serialisierungsfunktionalität zurückgegriffen. Zuerst sollen jedoch die Ereignisse mit ihren Signaturen definiert werden; wie für objektorientierte Ansätze typisch, sind diese als Beobachtermethoden (engl. listener) eines klassischen Beobachter-Musters [EGJ95] (engl. observer pattern) definiert. Aus Platzgründen wird der globale Identifikator durch GID (global identifier) abgekürzt:

```
onUpdate      ( [{GID, [byte]}] )
onInvalidate  ( [GID] )
```

Das Aktualisierungsereignis *onUpdate* liefert eine Liste mit Tupeln, bestehend aus einem Identifikator und einem binären Datenblock. Das Invalidierungsereignis *onInvalidate* wiederum besitzt lediglich eine Liste von Identifikatoren. Mit Hilfe der nun definierten Serialisierungsfunktionen können die durch die Beobachtermethoden bereitgestellten Ereignisdaten in die Objektsicht integriert werden.

```
setObject     ( GID, [byte] ) -> [GID]
getObject     ( wObjekt ) -> {GID, [byte]}
```

Die Funktion *setObject* aktualisiert das mit dem Identifikator spezifizierte Objekt mit den serialisierten Daten mit Hilfe der in Abschnitt 3.2.1 definierten integrierenden Deserialisierung. Wie bereits erwähnt, kann auch nur eine partielle Deserialisierung stattfinden. Zusätzlich liefert es gegebenenfalls noch eine Liste von Identifikatoren zurück, welche alle Objekte spezifizieren, die von dem gerade aktualisierten Objekt zwar referenziert werden, jedoch noch nicht in der Objektsicht vorhanden sind. Die Funktion *getObject* serialisiert das übergebene Objekt und liefert ein Tupel zurück, welches analog zu dem des Aktualisierungsereignisses aufgebaut ist. Diese Funktion ist nützlich, um beispielsweise Sicherungen von Objekten anzulegen.

Mit den vorgestellten Funktionen kann der Anwender nun selbst entscheiden, zu welchem Zeitpunkt und in welcher Form, partiell oder komplett, er die Daten der Ereignisse integrieren möchte.

3.2.14 Funktionsübersicht

Abschließend soll ein vollständiger Überblick über die von TGOS bereitgestellte Schnittstelle zur Manipulation des Objektraumes gezeigt werden, da die in Abschnitt 3.2.3 definierten Basisoperationen im Laufe des Kapitels teils deutlich erweitert wurden und zusätzliche Funktionen hinzu kamen. Tabelle 3.2 gibt einen Überblick über alle Funktionen, die das TGOS-Modell zur Modifikation der Objektsicht bereitstellt, inklusive deren überladene Varianten.

3.3 Verwandte Arbeiten

Das TGOS-Modell lässt sich nur schwer in eine bestimmte Kategorie einordnen. Vielmehr überschneidet sich das Konzept mit mehreren bereits existierenden Ansätzen, welche im weiteren Verlauf des Abschnitts erläutert und auf deren Unterschiede zu TGOS eingegangen werden soll.

Tuple Spaces, welche David Gelernter als Teil der Sprache Linda [Gel85] 1985 in einen ersten Variante präsentierte, und als deren Pionier er gilt, sind das naheliegendste Konzept für eine Vergleich mit TOGS. Ein Tupel Space ist ein verteilter, gemeinsam genutzter Speicherbereich, der beliebige Tupel enthält, die über wenige Operationen zugegriffen und verändert werden können. Eine weitere Besonderheit von Tuple Spaces sind ihre Assoziativität. Will ein Prozess ein Tupel bearbeiten, so muss er dieses erst aus dem Tupel Space extrahieren und kann nicht, wie bei DSM-Systemen üblich, direkt darauf zugreifen.

Prinzipiell bestehen zwischen TGOS und Tuple Spaces viele Gemeinsamkeiten, wie beispielsweise die Abstraktion zwischen logischem Zugriff und der daraus resultierenden Netwerkkommunikation. Jedoch bieten die Tuple Spaces und deren Derivate im Gegensatz zu TGOS kein feingranulares Ereignissmodell an. Des Weiteren ist für den Zugriff auf die Tuples immer nur ein spezifisches Konsistenzmodell definiert. Ein weiterer Unterschied liegt in der Verwendung der Objekte. Während bei Tuple Spaces die Objekte aus dem Tuple Space entfernt werden müssen, um sie zu schreiben, erlaubt TGOS die direkte Bearbeitung der Objekte. Die Synchronisationssemantik der Tuple Spaces ist bei TGOS somit nicht gegeben. Auch stellt TGOS keine assoziative Suchlogik bereit, wie sie bei Tuple Spaces zum Auffinden von Objekten verwendet wird.

Eine Erweiterung der Tuple Spaces stellen Object Spaces [Pol93] oder Java Spaces [FHA99] dar, welche zum Beispiel bei Java in Form der Jinis Verwendung finden. Eine Erweiterung der JavaSpaces hinsichtlich Grid-Computing sind wiederum die GigaSpaces [SA05], welche auch den Begriff des *in-memory data grids* geprägt haben.

Verteilte Objektmodelle sowie deren unterschiedliche Ausprägungen sind ebenfalls eng mit TGOS verwand. Eine allgemeine Beschreibung verteilter

Operation	Signatur	Beschreibung
Push	(wObjekt)	Objekt schreiben, Sperre lösen
	(wObjekt, wObjekt)	Operation verketten
	(wObjekt, Ordnung)	Ordnung der Ereignisnachrichten bestimmen
	(wObjekt, wGruppe)	Standardgruppen übergehen
Invalidate	(Objekt)	Objekt invalidieren, Sperre lösen
	(wObjekt, wObjekt)	Operation verketten
	(wObjekt, Gruppe)	Standardgruppen übergehen
Pull	(wObjekt)	Objekt aktualisieren
	(ID)	Objekt anfordern/aktualisieren
Sync	(wObjekt)	Sperre anfordern & aktualisieren
Order	(wObjekt)	asynchron aktualisieren
	(ID)	asynchron anfordern/aktualisieren
Put	(wObjekt, Name)	Objekt registrieren
Get	(Name) $\rightarrow$ wObjekt	Eintrag suchen
	(RegEx) $\rightarrow$ [Name]	Namensraum durchsuchen
setGroups	([Gruppenobjekt])	setzt Standardgruppen
getGroups	() $\rightarrow$ [Gruppenobjekt]	liefert die gesetzten Standardgruppen
setObject	(ID, [byte]) $\rightarrow$ [ID]	serialisierte Daten integrieren
getObject	(wObjekt) $\rightarrow$ {ID, [byte]}	Objekt serialisieren

Tabelle 3.2: Liste der Operationen

Objektmodelle findet sich in [CC91]. Ein bekannter Ansatz, der ein objektorientiertes Modell ebenfalls um Verteilung erweitert, ist Emerald [Jul92]. Eines der Hauptmerkmale von Emerald sind die ortsabhängigen Objekte. Dabei wird der Ort, im Sinne von Knoten, an dem ein Objekt residiert, auf Sprachebene definiert und es findet im Gegensatz zu DSM-Systemen

keine automatische Replikation statt. Ein Objekt kann auch dynamisch zur Laufzeit von einem Knoten zu einem anderen verschoben werden, um beispielsweise Lastverteilungsmechanismen zu implementieren. Im Gegensatz zu TGOS wird bei Emerald die Verteilung auf Sprachebene definiert, wodurch eine spezielle Programmiersprache mit einem passenden Compiler verwendet werden muss.

Ein weiterer Ansatz für ein verteiltes Objektmodell stellen die *fragmentierten Objekte* [MGpLNS91] dar. Ein fragmentiertes Objekt ist aus Anwendungssicht von jedem Knoten aus mit den gleichen Funktionen aufrufbar und verhält sich wie ein lokales Objekt. Die verteilte Funktionalität wird intern von Fragmenten erbracht, in die das Objekt aufgeteilt ist und die jeweils auf anderen Knoten residieren können. Die Kommunikation zwischen den Fragmenten erfolgt dabei über nachrichtenorientierte Mechanismen. Fragmentierte Objekte versuchen, ihren verteilten Charakter vor einem Nutzer zu verbergen, im Gegensatz zum TGOS-Modell, das die Verteilungsmechanismen sichtbar und steuerbar macht. Jedoch wäre es ohne weiteres möglich, die nachrichtenorientierte Kommunikation der Fragmente durch TGOS zu ersetzen.

Die oft als *Middleware* bezeichneten verteilten Objektmodelle, wie beispielsweise Corba [Vin93], JAVA EE oder .Net, bieten deutlich mehr Funktionalität und Komplexität, als die bereits Erwähnten. Bei all diesen Middleware Systemen steht, wie schon zuvor, nicht der Ansatz eines verteilten Objektraumes im Sinne des verteilten Speichers im Vordergrund, sondern die Ausführung von Methoden an entfernten Objekten (RPC/RMI) und das dafür benötigte Marshalling der Objekte. Die Kommunikation zwischen entfernten Diensten und/oder Prozessen erfolgt dabei, falls nicht über entfernte Methodenaufrufe, mit Hilfe nachrichtenorientierter Mechanismen. Zusätzlich gibt es eine Reihe weiterer Dienste, die beispielsweise die Persistenzierung in eine Datenbank oder die Nutzung in heterogenen Umgebungen ermöglichen.

Neben den verteilten Objektmodellen, bedient sich TGOS, wie anfangs erwähnt, auch einiger DSM-Konzepte. Dabei hebt TGOS die transparente Verteilung des gemeinsamen Speichers auf und ersetzt es durch das Konzept der Objektsichten und expliziten Operationen. Eine weitere Untergruppe der DSM-Systeme stellen dabei *Distributed Transactional Memory* Systeme dar. Bei diesen Systemen steht insbesondere die transaktionale Konsistenz, als Mittel der einfachen Synchronisierung und Programmierung, im Vordergrund. Die Konsistenz wird hier im Gegensatz zu TGOS gewöhnlich pro Speicherbereich definiert, Seitengranularität ist oftmals üblich. Ebenso wie Tuple Spaces bieten diese Systeme keine ereignisorientierte Programmierbarkeit. Vertreter dieses Ansatzes sind zum Beispiel der Object Sharing Service [MMSS09] der Universität Düsseldorf, sowie das verteilte transaktionale Betriebssystem Rainbow [SSK+10] der Universität Ulm.

Im Gegensatz zu den bisher beschriebenen Systemen, welche die Re-

plikationsschicht transparent in das Objektmodell integrieren und vor dem Nutzer verstecken, integriert das TGOS-Modell diese explizit und sichtbar als eigenständige Komponenten. Der Ansatz, die Replikationsschicht separat zum verteilten Objektmodell zu betrachten, wurde in [DRH07] verfolgt, jedoch wiederum völlig transparent für den Anwender. Im TGOS-Modell wird dabei nicht direkt Bezug auf den Aufbau und die Ausgestaltung der Replikationsschicht genommen, sondern diese nur als abstrakte Schicht wahrgenommen.

3.4 Zusammenfassung

Im TGOS-Programmiermodell werden viele bereits existierende und teils gegensätzliche Konzepte in neuer Form miteinander kombiniert und in ein in sich geschlossenes Gesamtmodell integriert. Der datenzentrierte und transparente Ansatz klassischer DSM-Systeme, wie IVY, Muni oder auch Rainbow, wird aufgegriffen und dahingehend verändert, dass die Transparenzeigenschaft durch den expliziten Charakter nachrichtenorientierter Systeme ersetzt wird. Dadurch kann der Zeitpunkt einer Netzwerkkommunikation sowie deren Granulariät genau bestimmt werden, wie in Abschnitt 3.1 gefordert. Das TGOS-Modell bedient sich wie Middleware-Systeme eines strikten Typensystems und objektorientierter Paradigmen, verzichtet aber völlig auf entfernte Methodenaufrufe und damit verbundene Konzepte.

Die vorgestellte Abstraktion von Replikationsschicht und Objektsichten ermöglicht eine einfache Austauschbarkeit der Replikationsschicht und das Wechseln zwischen unterschiedlichen Netzarchitekturen, ohne die, auf Basis der Objektsicht implementierten, Algorithmen anpassen zu müssen. Die Integration der Replikationsschicht in sichtbarer und abstrakter Form stellt ebenfalls eine Neuerung dar; ist sie doch in den gängigen Middleware-Systemen, wie Java, und DSM-Systemen, wie Rainbow, ein integraler und vor allem transparenter Bestandteil. Dadurch ist es beispielsweise möglich, dass die Knoten völlig frei Objekte in ihren Objektsichten verändern können und neu beitretende Knoten dennoch ein konsistentes Abbild in der Replikationsschicht vorfinden.

Eine weitere Besonderheit stellt der Ansatz zur Umsetzung von Konsistenzmodellen dar, die im Gegensatz zu den gängigen DSM-Systemen nicht direkt im Modell integriert sind, sondern innerhalb des Modells mit Hilfe der bereitgestellten Basisoperationen verwirklicht werden. Des Weiteren wird Konsistenz im TGOS-Modell nicht pro Speicherbereich definiert, sondern innerhalb des Programmflusses, was ebenfalls im Gegensatz zu DSM-Systemen steht. Dadurch können neue Konsistenzmodelle im laufenden Betrieb integriert und bestimmte Programmabläufe konsistenziert werden.

Im Gegensatz zu Middleware-Systemen, wie Jave EE oder Corba, definiert TGOS eine sehr schlanke Schnittstelle mit nur wenigen Basisopera-

tionen sowie einer geringen Anzahl an Randbedingungen, die zu beachtet sind. Diese Reduktion der Komplexität begünstigt nicht nur den Lern- und Lehraufwand, sondern sorgt auch dafür, dass allfällige Implementierungen sehr schlank gehalten werden können.

Kapitel 4

Architektur einer verteilten Welt mit TGOS

Im Folgenden sollen die im vorherigen Kapitel vorgestellten theoretischen Konzepte des TGOS-Modells für die Architektur einer virtuellen und verteilten Welt verwendet werden.

Im ersten Absatz werden die Besonderheiten beim Einsatz eines datenzentrierten Entwurfsmusters beschrieben. Darauf aufbauend, werden die Komponenten einer mit TGOS verteilten Welt beschrieben, die für die Visualisierung und Berechnung der Physik und Logik verantwortlich sind. Wie in Kapitel 3 beschrieben, bietet das TGOS-Modell die Möglichkeit, neue Konsistenzmodelle innerhalb des Modells zu definieren. In Abschnitt 4.3 werden daher mögliche Realisierungen einiger ausgewählter Konsistenzmodelle vorgestellt und erläutert. Besonderes Augenmerk liegt dabei auf der transaktionalen Konsistenz und deren Eignung für verteilte virtuelle Welten. Anhand eines konkreten Beispiels wird der Einsatz von TGOS in Abschnitt 4.4 beschrieben, sowie die Umsetzung eines szenenspezifischen Konsistenzmodells demonstriert. Absatz 4.5 widmet sich der Umsetzung der in virtuellen Welten verwendeten Lastverteilungsalgorithmen, wie beispielsweise Area-of-Interest-Management, sowie der Beschreibung des Aufbaus einer für eine verteilte Welt mit TGOS geeigneten Replikationsschicht.

Neben den Erläuterungen zum Entwurf einer virtuellen Welt mit TGOS, wird in Absatz 4.6 auch ein Vergleich mit einem klassischen Client/Server-Ansatz gezogen.

4.1 Datenzentriertes Entwurfsmuster

Eine der Neuerungen bei der Gestaltung von verteilten virtuellen Welten mit TGOS ist der Einsatz eines datenzentrierten Entwurfsmusters. In gängigen Ansätzen für verteilte Welten oder Spiele, wie beispielsweise Quake oder SecondLife, steht zumeist die Struktur der zugrunde liegenden Netzwerkar-

chitektur sowie der Entwurf der benötigten Nachrichtentypen und damit verbundenen Automaten im Vordergrund. Im Gegensatz dazu wird beim TGOS-Modell der Schwerpunkt auf den Entwurf der Datenstrukturen gelegt.

4.1.1 Verteilte Datenstruktur

Das Wesentliche an den eingangs beschriebene Datenstrukturen ist, dass es sich um **verteilte** Datenstrukturen handelt, die von allen teilnehmenden Knoten gleichberechtigt geändert oder gelesen werden können. Das Konzept der Objektsichten (siehe Abbildung 3.3 in Abschnitt 3.2.1), mit der Trennung in einen lokalen Objektraum und eine globale Replikationsschicht, bietet die Möglichkeit, dass jeder Knoten lokal eine eigene Version der verteilten Datenstruktur haben kann und es dennoch ein global konsistentes Abbild gibt, welches automatisch durch die Replikationsschicht bereitgestellt wird.

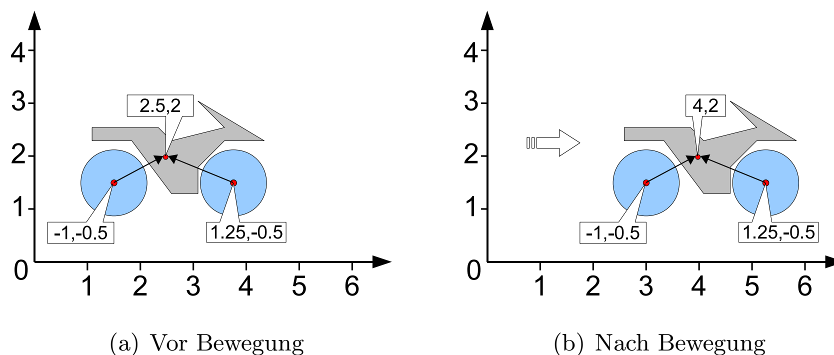
Die Vielzahl unterschiedlicher Inhalte (zum Beispiel Autorennen, fliegende Teppiche, etc.) einer Welt adäquat in eine einzige Datenstruktur zu überführen, stellt sich, aufgrund der großen Heterogenität der Inhalte, als schwierig, je nach Zielkonflikt, sogar als unlösbar heraus. Daher ist es sinnvoller, die Welt in einzelne Bereiche zu unterteilen, welche unabhängig von einander strukturiert sein können. Diese Bereiche, oder besser Szenen, sind nicht nur im Sinne der Komplexitätsreduktion sinnvoll, sondern auch um etwaige Lastverteilungs- oder Area-of-Interest-Algorithmen auf die entsprechenden Inhalte abzustimmen (siehe Abschnitt 4.5). Obwohl die Inhalte der Szenen sehr unterschiedlich sein können, sollten alle Szenen auf eine gemeinsame Basis zurück greifen, um Funktionen, die in der gesamten virtuellen Welt zur Verfügung stehen sollten, automatisch bereitzustellen. Dazu zählt beispielsweise der Beitritt eines Avatars oder auch die Art, wie die Position oder Ausrichtung eines Objektes definiert wird.

Aus den oben genannten Gründen ist es sinnvoll, einen gemeinsamen, verteilten Weltgraphen zu definieren. Dieser stellt eine Erweiterung der Definition 4 aus Kapitel 2 dar, welche in Definition 6 erweitert und präzisiert wird.

Definition 6: Verteilter, gemeinsamer Weltgraph
Ein verteilter, gemeinsamer Weltgraph ist eine Struktur, welche einen Bauplan für eine virtuelle Welt darstellt. Dieser Bauplan beschreibt neben der grafischen und physikalischen Ausgestaltung auch Interaktionsmöglichkeiten und cineastische Abläufe. Teilgraphen, die unabhängige Inhalte oder Szenen beschreiben, werden verteilte, gemeinsame Szenengraphen genannt.

Für den Prototypen einer mit TGOS verteilten virtuellen Welt wird diese in disjunkte Szenen unterteilt, welche durch einen azyklischen, gerichteten Graphen gemäß der Definition 6 definiert werden. Weiterhin wird festge-

legt, dass ein Nutzer sich immer nur in genau einer Szene befinden kann, beziehungsweise genau einer Szene zugeordnet ist. Jeder Knoten im Graph ist ein *TGOS-Wurzelobjekt* (siehe Abschnitt 3.2.4), wodurch alle Referenzen zwischen den Knoten global eindeutig sind.



(a) Vor Bewegung

(b) Nach Bewegung

Abbildung 4.1: Transformationshierarchie

Der Szenengraph ist hierarchisch aufgebaut und seine Struktur definiert eine Transformationshierarchie, bei der die Transformation eines Kindknotens immer relativ zum Vaterknoten definiert ist; das Koordinatensystem des Kindknotens hat seinen Ursprung im Mittelpunkt des Vaterknotens. Als Transformation bezeichnet man dabei die Komposition von Position, Rotation und Skalierung eines Objektes. Abbildung 4.1 verdeutlicht das Konzept der Transformationshierarchie. Die Räder des Motorrades in Abbildung 4.1(a) sind Kindknoten der Motorradkarosserie und werden relativ zu dessen Schwerpunkt (roter Punkt) positioniert. Wird nun die Karosserie wie in Abbildung 4.1(b) bewegt, so bewegen sich auch die Räder mit. Die hier vorgestellte Konzeption eines Szenengraphen ist nur eine mögliche Variante und erhebt nicht den Anspruch einer allgemein gültigen Lösung.

Der Szenengraph besteht aus verschiedenen Objekttypen, die durch Spezialisierung des allgemeinen Graphknotens definiert werden. Hierzu gibt es mehrere Möglichkeiten; neben dem Einsatz einer Vererbungshierarchie können die Knoten auch via Schnittstellendefinitionen und Delegation spezialisiert werden. Der hier vorgestellte Ansatz nutzt der Einfachheit halber eine Vererbungshierarchie. Abbildung 4.2 zeigt die UML-Notation der Vererbungshierarchie, welche auch im Wissenheim Worlds Prototypen (siehe Kapitel 5) Verwendung findet. Im Diagramm wird zwischen strukturierenden Knoten auf der linken Seite und Objektknoten auf der rechten Seite unterschieden.

Die *Strukturknoten* definieren für alle nachfolgenden Kindknoten spezielle logische Zugehörigkeiten, unabhängig von der Transformationshierarchie. Zum Beispiel gehören alle Kindknoten eines *Verbundobjektes* logisch gesehen zu diesem Verbundobjekt, wie in Abbildung 4.3 beim Avatar und Gegen-

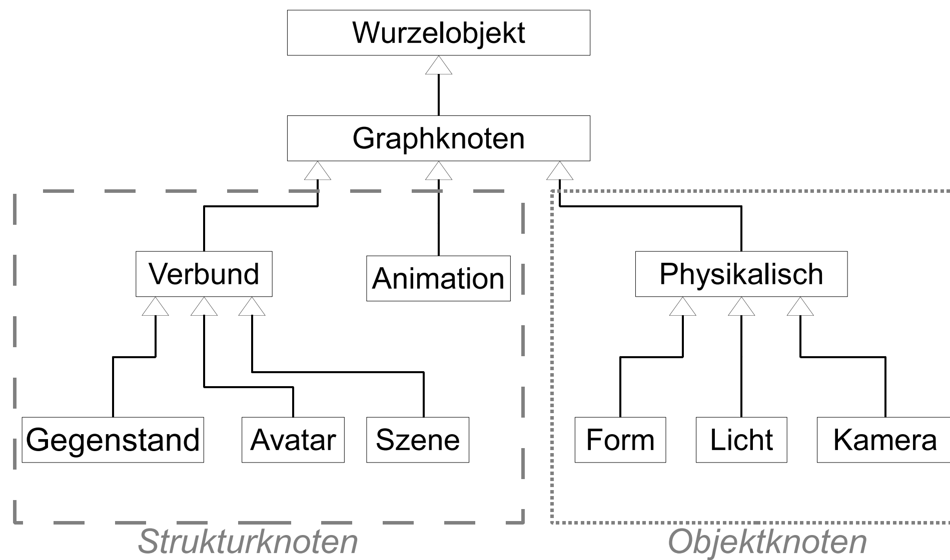


Abbildung 4.2: Vererbungshierarchie des Szenengraphens

stand zu sehen ist. Auch eine Schachtelung von strukturierenden Objekten ist möglich. Verbundobjekte besitzen zusätzlich eine Transformation, die es ermöglicht, den logischen Verbund als Ganzes zu transformieren. Ein spezielles Verbundobjekt stellt das Szenenobjekt dar, welches als Einstiegspunkt in eine Szene fungiert und welches zusätzlich *szenenspezifische Ereignisroutinen* bereitstellt (siehe Abschnitt 4.1.2). Das *Animationsobjekt*, welches den Zustand (Start, Stop, Fortschritt, etc.) einer cinematographischen Animation definiert (näheres zu Animationen findet sich in Abschnitt 4.2), wirkt auf alle Kindknoten, wie in Abbildung 4.3 durch den roten Rahmen angedeutet. Die *Objektknoten* definieren Spezialisierungen für Objekte, welche in der virtuellen Welt vorkommen, beziehungsweise diese beeinflussen. Ein *physikalisches Objekt* definiert die physikalischen Eigenschaften sowie die Transformation eines Objektes in der virtuellen Welt. Es wird durch ein *Formobjekt* weiter spezialisiert, indem es ihm eine 3-dimensionale Form verleiht und genau ein Material für dessen Oberfläche definiert. Formobjekte werden durch den Visualisierungsprozess dargestellt und bilden die sichtbaren Bausteine der virtuellen Welt. Um komplexe Objekte, welche aus vielen Materialien bestehen, darstellen zu können, kombiniert man mehrere Formobjekte miteinander. Im Falle des Körpers eines Avatars, geschieht dies unter Ausnutzung der Transformationshierarchie, wie in Abbildung 4.3 bei ❶ zu sehen ist. Dort ist der Fuß am Unterschenkel, der Unterschenkel am Oberschenkel und der Oberschenkel am Unterkörper angebracht. Das Lichtobjekt definiert hingegen eine Lichtquelle in der virtuellen Welt, während eine Ka-

mera eine bestimmte Sicht in der virtuellen Welt definiert.

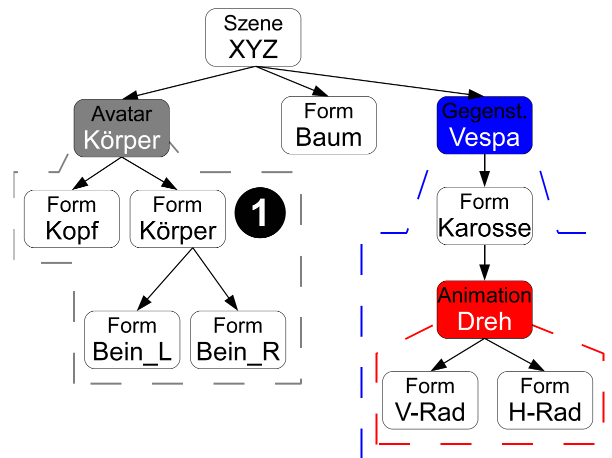


Abbildung 4.3: Struktur eines Szenengraphens

Im Gegensatz zum nachrichtenbasierten Ansatz erfolgt die Kommunikation der teilnehmenden Spieler direkt über den Szenengraphen. Beispielsweise aktualisieren Spieler die Position ihres Avatars, indem sie dessen Transformationsobjekt mit Hilfe der TGOS-Pushoperation aktualisieren. Bei allen anderen Teilnehmern wird dadurch automatisch der Inhalt des betroffenen Transformationsobjektes aktualisiert. Durch das im TGOS-Modell definierte Sichtenkonzept (siehe Abschnitt 3.3) ergibt sich ferner die Unterscheidung in einen lokalen und globalen Szenengraphen. Als lokaler Szenengraph wird die aktuelle Version des Graphen in der Objektsicht des Knotens bezeichnet und als global der Szenengraph, wie er sich aus Sicht der Replikationsschicht darstellt.

4.1.2 Interaktion und lokaler Kontext

Wie anfangs erwähnt, soll der Szenengraph auch Informationen zur Interaktionssteuerung beinhalten, wozu insbesondere auch die Nutzerinteraktionen mit den, durch den Szenengraph beschriebenen, Objekten in der virtuellen Welt gehören. Ein gängiges Schema ist das Beobachter-Muster (engl. observer pattern [EGJ95]), bei dem das Objekt, mit dem interagiert werden kann, die Möglichkeit anbietet, ein Beobachterobjekt zu registrieren. Dieses Objekt besitzt wohldefinierte Methoden, die im Falle einer Interaktion durch das beobachtete Objekt gerufen werden. Zu diesem Zweck bietet jedes Formobjekt eine Beobachterschnittstelle an, die ein Szenendesigner verwenden kann, um auf Ereignisse am Objekt zu reagieren. Abbildung 4.4 zeigt eine vereinfachte Form der Beobachterschnittstelle in einer Java-kompatiblen Definition.

Dabei ist zu beachten, dass die Behandlung des Ereignisses und der Auf-

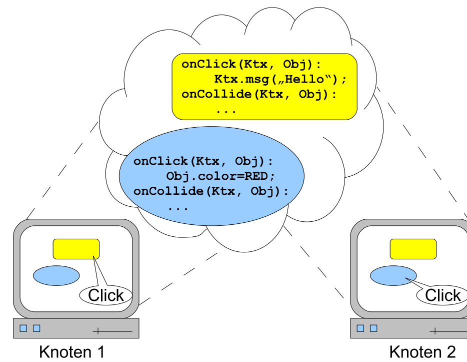
```
public interface FormEvents {
  void onClick    (Objekt clicked,
                   Context localContext);
  void onCollide  (Objekt collidedWith,
                   Object collidedBy,
                   Context localContext);
}
```

Abbildung 4.4: Beobachterschnittstelle eines Formobjektes

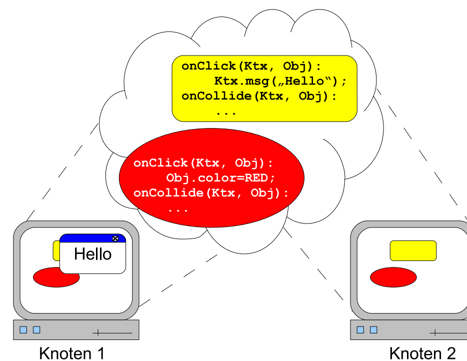
ruf einer eventuell vorhandenen Behandlungsmethode immer auf dem Knoten vonstatten geht, der das Ereignis ausgelöst hat. Die Behandlungsroutine wird, wie die Szene auch, global für alle teilnehmenden Knoten definiert, wodurch der Routine kein direkter Zugriff auf etwaige knotenlokale Komponenten oder Objekte möglich ist. Zu diesem *lokalen Kontext* zählen alle Objekte und Methoden, die spezifisch für einen Knoten sind, wie beispielsweise dessen Fenstersystem, Eingabegeräte oder auch die Identifikation des Avatars des steuernden Benutzers. Daher müssen beim Aufruf einer registrierten Behandlungsroutine Informationen über den lokalen Kontext übergeben werden, wie in Abbildung 4.4 mit dem Parameter *localContext* angedeutet. Bei der Behandlung dieser objektspezifischen Ereignisse tritt eine weitere Besonderheit im Aufbau des Szenengraphens auf. Tritt ein Ereignis an einem Formobjekt auf, welches keine Behandlungsroutine besitzt, so wird ermittelt, ob es ein zugehöriges Verbundobjekt gibt und das Ereignis an dessen Ereignisbehandlungsroutine propagiert, falls diese vorhanden ist.

Abbildung 4.5 verdeutlicht diesen Ansatz anhand eines einfachen Beispiels. Der Szenengraph beinhaltet zwei Objekte, die beide auf ein Klick-Ereignis reagieren können; das rechteckige Objekt zeigt bei einem Klick ein Meldungsfenster an, während das elliptische Objekt seine Farbe ändert. Wie in 4.5(b) zu sehen, wird das Meldungsfenster, durch das Klick-Ereignis von 1, nur auf diesem Knoten dargestellt, da der Aufruf des Meldungsfensters kontextspezifisch war. Knoten 2 hingegen änderte den Farbwert des Objektes im Szenengraphen, was Auswirkung auf alle teilnehmenden Knoten hat.

Es gibt neben den objektspezifischen Ereignissen auch noch solche, die die gesamte Szene betreffen. In Abbildung 4.6 ist eine Beobachterschnittstelle für szenenspezifische Ereignisse mit zwei Ereignissen definiert. Das Ereignis *onHIDEvent* wird gerufen, falls eine Benutzereingabe (Tastendruck, Mausbewegung, etc...) vorliegt und *onNextStep*, wenn ein neuer Zeitpunkt t+1 erreicht wurde. Weiterführende Erklärungen zu den Ereignissen finden sich in Abschnitt 4.2. Die Referenz auf ein Beobachterobjekt ist im Szenenobjekt angesiedelt.



(a) Vor den Ereignissen



(b) Nach den Ereignissen

Abbildung 4.5: Ereignisbehandlung

4.1.3 Datenklassifikation

Wie in Abschnitt 2.2.1 bereits beschrieben, können die Daten einer verteilten Welt in konstante und variable Anteile unterschieden werden. Variable Daten werden dabei direkt in den Graphen integriert, da diese häufig geändert und damit idealerweise auch über die Verteilungsmechanismen des Graphen verteilt und/oder aktualisiert werden. Statische Daten, wie beispielsweise Texturen, 3D-Modelle oder Audiodaten, erfahren jedoch nur äußert selten Änderungen und sind in ihrem Umfang auch deutlich größer. Diese Daten liegen gewöhnlich in Form von Dateien vor, was ein Zwischenspeichern auf den teilnehmenden Knoten sehr einfach möglich macht. Die Daten bleiben dabei oftmals sowohl über das Programmend,e als auch über einen Neustart des Knoten hinaus bestehen. Diese Persistenz ermöglicht es, den Datenverkehr zwischen teilnehmenden Knoten und der virtuellen Welt enorm zu reduziert, da diese Dateien nur einmal geladen werden müssen. Da statische Daten zwar logisch im verteilten Szenengraphen vorhanden sind, physikalisch aber

```
public interface SceneEvents {
  void onHIDEvent (HIDEvent event,
                   Context localContext);
  void onNextStep (Time time,
                   Context localContext);
}
```

Abbildung 4.6: Beobachterschnittstelle eines Szenenobjektes

lokal zwischengespeichert werden, bedient man sich eines abstrakten Ressourcenindentifikators, um die Daten zu referenzieren. Der Zugriff auf die statischen Daten erfolgt dann über den Identifikator, unter Zuhilfenahme des lokalen Kontextes.

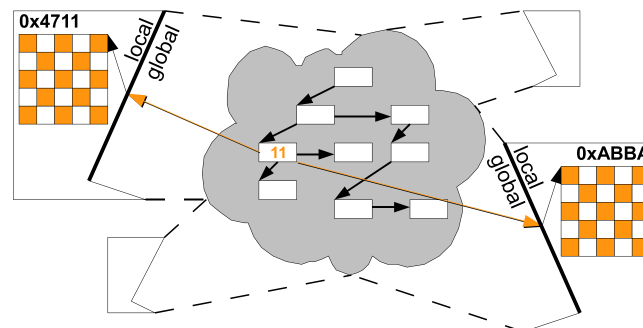


Abbildung 4.7: Lokale und globale Daten

Abbildung 4.7 verdeutlicht das Konzept dieses Ansatzes. Beide Knoten im Schaubild laden die gleiche logische Ressource und übersetzen den Identifikator in eine lokale Textur. Dabei kann die Textur auf beiden Knoten an ganz unterschiedlichen Adressen liegen. Durch diese Trennung ist es möglich, die dynamischen Daten des Szenengraphens von den statischen Daten zu trennen. Es ist jedoch trotzdem möglich, Mediendaten direkt in den Szenengraphen zu integrieren, was bei dynamischen und personalisierten Daten von Vorteil sein kann.

4.2 Komponenten einer verteilten Welt

Zusätzlich zum verteilten Welt- oder Szenengraphen benötigt eine virtuelle Welt weitere Komponenten. Diese dienen sowohl der Visualisierung und Manipulation des Graphen, als auch der Bearbeitung von Nutzereingaben. Abbildung 4.8 gibt einen Überblick über die für eine virtuelle Welt mit TGOS benötigten zusätzlichen Komponenten.

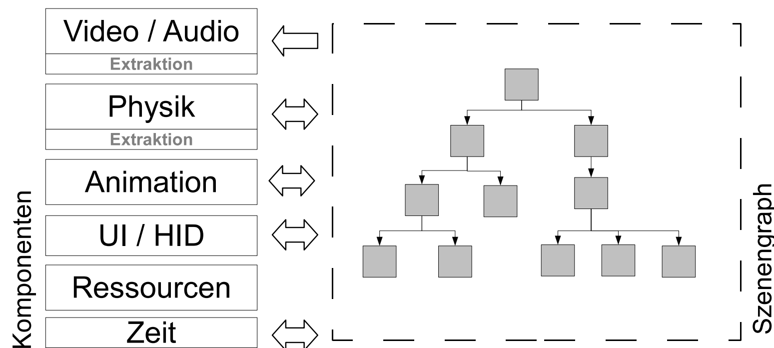


Abbildung 4.8: Komponenten einer virtuellen Welt mit TGOS

Die Auswahl der Komponenten basiert auf Erfahrungen mit dem Wissenheim Worlds Prototypen und stellen, wie schon beim Entwurf des Szenengraphen, nur eine mögliche Auswahl dar. Je nach Konzeption und Ausrichtung der virtuellen Welt können zusätzliche Komponenten benötigt werden. Im Folgenden soll nun näher auf die genauen Aufgaben der Komponenten sowie deren Funktionsweisen und Zugriffe auf den Szenengraphen eingegangen werden.

Video / Audio

Die Video/Audio-Komponente ist für das Darstellen der Objekte eines Szenengraphen - sowohl in visueller wie auch auraler Hinsicht - verantwortlich. Dabei ist die Art der Ausgabe stark von der Konzeption der verteilten Welt abhängig und kann von der Anzeige einer HTML-Seite bis hin zu einer fotorealistischen, 3-dimensionalen Darstellung mit Dolby Surround reichen.

Ein wichtiger Teilaspekt stellt die Extraktion der benötigten Informationen aus dem Szenengraphen dar. Betrachtet man beispielsweise eine 3-dimensionale Grafik, die mit Hilfe von Direct3D oder OpenGL dargestellt werden soll, so ist dort die Reihenfolge, in der Objekte gezeichnet werden, von großer Bedeutung. Transparente Objekte müssen beispielsweise sortiert werden, um die korrekten Überblendungseffekte zu erreichen. Des Weiteren wird für die Visualisierung und/oder die Akustik die absolute Transformation der Objekte benötigt, welche erst aus der Transformationshierarchie berechnet werden muss. Zusätzlich sollte eine Sicht- beziehungsweise Hörweitenüberprüfung durchgeführt werden, um die Anzahl der zu bearbeitenden Objekte zu minimieren. Während der Extraktion wird daher der Szenengraph traversiert und alle benötigten Informationen und Daten in einer Zwischenstruktur abgelegt, die für die jeweilige Ausgabe optimiert ist. Diese Datenstruk-

tur ist rein knotenlokal, kann aber Verweise auf den Szenengraphen enthalten, im Falle der Visualisierung zum Beispiel eine Referenz auf Materialeigenschaften.

Des Weiteren gilt, dass auf den verteilten Szenengraphen durch diese Komponente nur lesend zugegriffen wird.

Physik

Die Physik-Komponente dient der physikalisch korrekten Bewegung und Interaktion von Objekten. Eine wichtige Funktion ist sowohl die Kollisionserkennung und Behandlung zwischen Objekten, als auch das Anwenden von physikalischen Kräften auf Objekte.

Analog zur Video/Audio-Komponente findet auch innerhalb der Physik eine Extraktion statt, die sowohl die absolute Position der Objekte berechnet, als auch eine Kategorisierung der physikalischen Objekte vornimmt oder spezielle Indizes anlegt. Eine mögliche Kategorisierung unterscheidet zwischen aktiven, sich bewegenden Objekten und passiven, stillstehenden Objekten; Kollisionen können nur Objekte auslösen, die sich bewegen. Indexstrukturen werden häufig in der Kollisionserkennung verwendet, um mit möglichst geringem Rechenaufwand potentielle Kollisionspaare zu finden; eine sehr bekannte Indexvariante ist beispielsweise der *Octree* [WVG92].

Im Gegensatz zur Video/Audio-Komponente schreibt diese auch Daten in den Szenengraph. Dazu zählt insbesondere die Transformation, falls ein Objekt bewegt, skaliert oder rotiert wurde.

Animation

Neben Objekten, für die eine physikalisch korrekte Berechnung ihrer Bewegung wichtig ist, gibt es eine Vielzahl von Objekten, bei denen nur die Bewegung als solche von Interesse ist. Zum Beispiel ist es für die Drehung eines Windrades oder das Öffnen einer automatischen Schiebetür meistens unerheblich, ob es sich mit der physikalisch korrekten Geschwindigkeit bewegt. Eine gängige Methode der Animation stellt hierbei die Schlüsselbildanimation [Cat72] (engl. keyframe animation) dar, welche von allen gängigen 3D-Modellierungsprogrammen bereitgestellt wird.

Für die Steuerung solcher Animationen wird das in Abschnitt 4.1.1 vorgestellte Animationsobjekt verwendet. Dazu traversiert die Komponente den Szenengraphen und berechnet die Transformation für alle Objekte anhand der Schlüsselbildinformationen, welche an den zu animierenden Objekten angesiedelt sind. Trifft die Animationslogik bei der Traversierung des Graphen auf ein Animationsobjekt, so wird dieses hinsichtlich seines Zustandes (Start/Stop, letztes berechnetes Bild) ausgewertet. Der Zustand dieses Animationsobjekts gilt dabei für alle

nachfolgenden Kindknoten und endet erst beim Erreichen eines weiteren Animationsobjektes oder wenn keine weiteren animierbaren Objekte mehr vorhanden sind.

Die Animationskomponente aktualisiert die Transformation der animierten Objekte im Szenengraph.

UI / HID

Da für die graphische Darstellung der virtuellen Welt auf spezielle Grafikschnittstellen wie Direct3D oder OpenGL zurückgegriffen wird, ist es meistens nicht möglich, das durch das Betriebssystem oder die Laufzeitumgebung bereitgestellte Fenstersystem zu nutzen. Aus diesem Grund wird eine Benutzerschnittstellen-Komponente (engl. user interface, kurz UI) benötigt, welche sowohl die Logik eines Fenstersystems als auch dessen graphische Ausgestaltung bereitstellt.

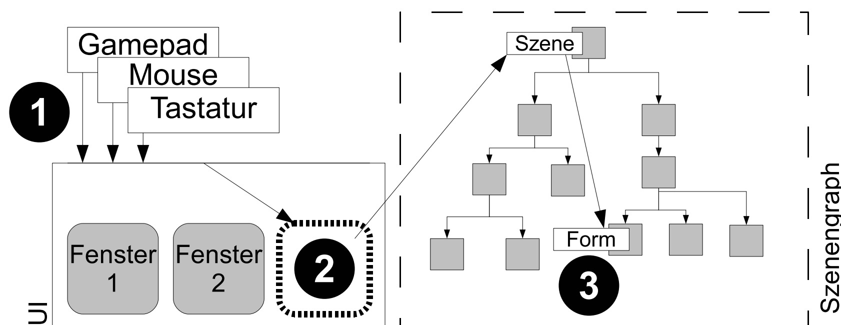


Abbildung 4.9: Ereignisverarbeitung

Zusätzlich muss die Benutzerschnittstelle die verschiedenen Eingabegeräte (engl. Human Input Devices, kurz HID), wie beispielsweise Maus, Tatstatur, Gamepad oder Joystick verwalten, sowie deren Ereignisse auswerten und verteilen. Abbildung 4.9 verdeutlicht den Ablauf einer Nutzereingabe sowie deren Verarbeitung. Meldet nun ein Eingabegerät ein Ereignis, wie zum Beispiel einen Tastendruck, an das UI, so ermittelt dieses, wie bei ❶ gezeigt, welche interne Komponente im Moment den Eingabefokus hat und leitet das Ereignis an diese Komponente weiter. Die in Abschnitt 4.1.2 vorgestellte Schnittstelle für szenenspezifische Ereignisse stellt eine solche Komponente des UI dar und ruft, wie bei ❷ zu sehen, die Behandlungsroutine am Szenenobjekt auf. Die Routine der Szene übernimmt die weitere Auswertung des Ereignisses und leitet es gegebenenfalls an die Ereignisroutine eines Objekts im Szenengraph weiter, in Abbildung 4.9 bei ❸ zu sehen.

Ressourcen

Wie in Abschnitt 4.1.3 bereits beschrieben, werden Mediendaten im Szenengraphen durch Ressourcenidentifikatoren referenziert. Die Ressourcen-Komponente sorgt nun für eine Auflösung von abstrakten Identifikatoren zu realen Daten, welche sowohl von der Video/Audio- als auch der Physikkomponente benötigt werden. Zusätzlich zur Bereitstellung ist die Komponente auch für die Konvertierung der Daten zuständig. Beispielsweise kann aus dem textuellen Format eines 3-dimensionalen Objektes, in Form einer Datei, eine Objektstruktur erzeugt, die sehr einfach in den Szenengraphen integriert werden kann.

Die Ressourcenkomponente lädt, ähnlich wie beispielsweise Second-Life, alle benötigten Mediendaten aus einem zentralen Repositorium, welches in verschiedenen Formen vorliegen kann. Im Falle des Wissenheim Worlds Prototypen (siehe Kapitel 5) beispielsweise, findet ein Webserver als Repository Verwendung. Zusätzlich muss die Komponente einen Cachingmechanismus für Mediendateien bereitstellen, der die geladenen Daten auf dem Knoten des Teilnehmers zwischenspeichert, um bei mehrfachen Zugriffen nicht auf das globale Repositorium zurückgreifen zu müssen.

Zeit

Ein zeitsynchronisierter Ablauf ist für alle Spiele und virtuelle Welten wichtig, um allen Benutzern eine einheitliche Sicht auf die Welt zu suggerieren. Üblicherweise lässt die Zeit in virtuellen Welten aber nur bedingt Rückschluss auf kausale Abhängigkeiten zu, da es sich nicht um eine logische Zeit im Lampertschen Sinne handelt. Es kann daher vorkommen, dass ein Knoten ein Ereignis zu einem Zeitpunkt t in der Welt auslöst, obwohl ein anderer Knoten zum selben physikalischem Zeitpunkt bereits bei Zeitpunkt t+1 angelangt ist. Dennoch kann die Zeit verwendet werden, um physikalische Berechnungen, Dead-Reckoning oder Animationen auszuführen.

Die Zeit in einer virtuellen Welt kann entweder diskret oder kontinuierlich definiert werden. Der Vorteil einer diskreten Zeit liegt in der einfachen Vergleichbarkeit zweier Zeitpunkte, bei einer kontinuierlichen Zeit ist ein direkter Vergleich auf Grund von Ungenauigkeiten und Rundungsfehlern schwierig. Des Weiteren ist eine diskrete Zeit numerisch stabiler als ihr kontinuierliches Pendant. Ein anderer Punkt betrifft die Rate, mit der die visuelle Ausgabe aktualisiert wird und die in der Regel durch die Bildwiederholfrequenz des Monitors auf 60 Hertz begrenzt ist. Um einen möglichst flüssigen Eindruck zu simulieren, sollten auch die Physik- und Animationskomponenten mit einem Vielfachen dieser Bildwiederholrate arbeiten, da es ansonsten zu wahrnehmbarer Varianz kommt.

Aus diesen Gründen ist es sinnvoll, eine diskrete Zeit zu nutzen, die,

```
public class Time {
  long timeStep;
  float stepRate;
}
```

Abbildung 4.10: Aufbau eines Zeitobjektes

wie in Abbildung 4.10 zu sehen, aus zwei Komponenten besteht. Die Erste ist die diskrete Zeit in Form von Zeitschritten (engl. time steps), welche streng monoton steigend ist, und als Zweites die Schrittrate, welche angibt, wie viele Zeitschritte innerhalb einer Sekunde vorkommen.

Für den verteilten Fall bedarf es einer Möglichkeit, die Zeit auf den teilnehmenden Knoten zu synchronisieren. Ein Möglichkeit ist, in jeder Szene ein Zeitobjekt zu verankern, das die globale Zeit repräsentiert. Dieses wird in regelmäßigen Zeitabständen mit der gerade gültigen globalen Zeit aktualisiert. Die Aktualisierungsintervalle sind dabei deutlich größer als die Intervalle der Zeitschritte auf dem Knoten. Bei jeder Aktualisierung kann dann die Zeitkomponente ihre lokale Zeit mit der globalen vergleichen und gegebenenfalls anpassen. Die Anpassung erfolgt dabei durch eine Verlangsamung, beziehungsweise Beschleunigung, der Uhr. Je nach gewähltem Replikationsmodell obliegt es einer der Zeitkomponenten der teilnehmenden Knoten, für eine Aktualisierung der globalen Zeit zu sorgen.

Das in Abschnitt 4.1.2 vorgestellte Szenenereignis *onNextStep* wird ebenfalls durch die Zeitkomponente ausgelöst und zwar bei jedem neuen Zeitschritt genau einmal. Da alle teilnehmenden Knoten die Komponente ausführen, tritt das Ereignis auch auf allen Knoten parallel auf.

Alle vorgestellten Komponenten nutzen den Szenengraphen als zentrale Datenstruktur und greifen dabei sowohl lesend als auch schreibend zu. Tabelle 4.1 gibt einen Überblick über das Zugriffsverhalten der einzelnen Komponenten.

Neben der Art des Zugriffs ist auch vermerkt, ob die Änderungen nur lokal, für den Knoten, der die Komponente ausführt, oder auch global, für alle Knoten, vorgenommen werden. Globale Aktualisierungen treten zum Beispiel in der Physik auf, wenn sich die Bewegungsrichtung des knotenlokalen Avatars ändert und diese Änderung verteilt werden soll. Ebenso verändert die UI/HID-Komponente den Szenengraphen nicht nur lokal, sondern verändert ihn in Form von Nutzeraktionen auch global. In der Zeitkomponente können Schreiboperationen entweder durch die Aktualisierung

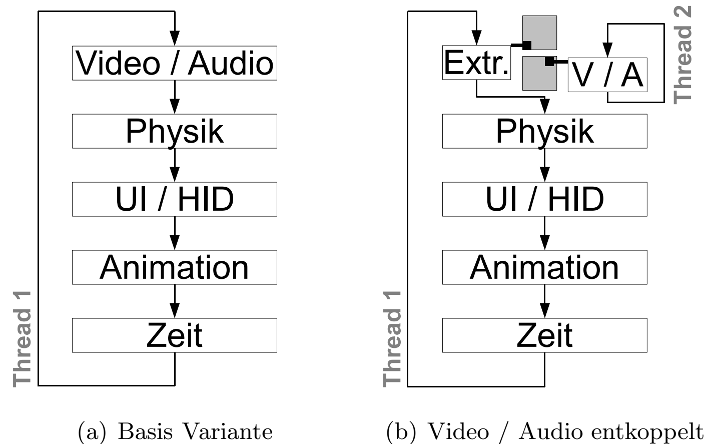
Komponente	Zugriffsart
Video/Audio	lesend
Physik	lesend & schreibend (lokal, global)
Animation	lesend & schreibend (lokal)
UI / HID	lesend & schreibend (lokal, global)
Ressourcen	lesend
Zeit	lesend & schreibend (lokal, global)

Tabelle 4.1: Zugriffsart der Komponenten auf den Szenengraphen

des globalen Zeitgebers auftreten oder auch durch den Aufruf des *onNext-Step*-Szenenereignisses.

4.2.1 Programmablauf

Nachdem die einzelnen Komponenten, die für die Darstellung und Interaktion mit einer virtuellen Welt benötigt werden, vorgestellt wurden, müssen diese im Rahmen eines Programmablaufes miteinander verknüpft werden. Die einfachste Ausführungsvariante stellt dabei die in Abbildung 4.11(a) dargestellte klassische Hauptschleife dar, in der alle Komponenten durch einen einzigen Thread repetitiv ausgeführt werden (engl. single threaded loop).



(a) Basis Variante (b) Video / Audio entkoppelt

Abbildung 4.11: Hauptschleifen

Durch die zunehmende Verbreitung von Prozessoren mit mehreren Kernen ist es sinnvoll, möglichst viele Teile der Hauptschleife parallel auszuführen. Alle Komponenten in parallele Threads auszulagern, ist ohne An-

passungen nur schwer möglich, da viele Komponenten sowohl lesend als auch schreibend auf den Szenengraphen zugreifen und dadurch eine entsprechende Synchronisierung nötig wird. Die einfachste Form der Parallelisierung bietet sich für die Video/Audiokomponente an, da die gesamte Komponente nur lesend auf den Szenengraphen zugreift und die eigentliche Visualisierung beziehungsweise Akustik auf separate Datenstrukturen, die aus dem Szenengraphen extrahiert werden, zurückgreift. Zum anderen bietet sich die Visualisierungskomponente an, da sie - je nach Grad des Fotorealismus - viel Rechenzeit benötigt. Der Ansatz ist in Abbildung 4.11(b) zu sehen; die Extraktion verbleibt in der klassischen Hauptschleife, lediglich die Synthetisierung der Visualisierung und/oder Akustik wird durch einen oder mehrere zusätzliche Threads bewerkstelligt. Da nun die Extraktion die separate Datenstruktur schreibt, während die Synthetisierung diese Daten liest, könnte es wieder zu Problemen mit Inkonsistenzen kommen. Aus diesem Grund ist es nötig, die separate Datenstruktur mindestens in doppelter Ausführung anzulegen, damit, während die eine Struktur geschrieben wird, die andere konsistent gelesen werden kann (im Bild durch die grauen Kästchen angedeutet). Dieses Verfahren wird in der Literatur häufig auch als *Doppelpufferung* (engl. double buffering) bezeichnet.

4.2.2 Aktualisierungsformen

In ihrer einfachsten Form arbeiten alle Komponenten *zustandslos*. Dies bedeutet, dass beispielsweise weder die Video/Audio- noch die Physikkomponente Informationen über Änderungen an Objekten im Szenengraphen erhält, beziehungsweise generiert. Vergleichbar ist dies mit dem klassischen Unterschied zwischen Polling und Unterbrechungen. Sämtliche Komponenten müssen dadurch den Szenengraphen kontinuierlich beobachten, um durch Vergleich mit dem letzten Durchlauf Änderungen zu ermitteln oder komplett zustandslos zu arbeiten. Die Extraktionskomponente der Video/Audio- oder Physikkomponente würde in diesem Fall ihre Zwischenstrukturen bei jedem Aufruf komplett neu aufbauen, obwohl nur marginale oder gar keine Änderungen aufgetreten sind. Dies hat einen deutlich erhöhten Rechenaufwand zur Folge, vereinfacht aber die Programmierung deutlich und ist für Systeme ohne dedizierte Ereignismechanismen die einzig mögliche Vorgehensweise.

Bei der *inkrementellen* Aktualisierung werden Änderungen an Objekten im Szenengraphen protokolliert und die Komponenten können anhand des Protokolls die geringstmöglichen Aktualisierungen durchführen. Änderungen durch entfernte Knoten am verteilten Szenengraphen zu erkennen, ist Dank des TGOS-Ereignismechanismus problemlos möglich. Aufwendiger hingegen ist es, Änderungen durch die lokalen Komponenten zu erfassen. Dazu ist es notwendig, dass alle Objekte ihre Daten kapseln und alle Methoden, welche diese internen Daten ändern, die Änderung protokollieren. Insbesondere

die Protokollierung ist fehleranfällig und muss sehr sorgfältig durchgeführt werden.

4.3 Konsistenzmanagement

Das TGOS-Modell selbst definiert kein explizites Konsistenzmodell, sondern bietet durch die Basisoperationen die Möglichkeit, eigene Konsistenzmodelle innerhalb des TGOS-Modells zu entwerfen und umzusetzen. Neben den klassischen Konsistenzmodellen, wie strikte und schwache Konsistenz, soll im Folgenden insbesondere auf die transaktionale Konsistenz eingegangen werden. Diese bietet sich beispielsweise an, um die Struktur des Szenengraphen bei Änderungen streng konsistent zu halten. Zusätzlich zu den klassischen Konistenzmodellen wird das Konzept der szenenspezifischen Konsistenz erläutert, die es ermöglicht, pro Szene eigene, speziell angepasste Konsistenzmodelle zu entwerfen.

Im Folgenden soll die praktische Realisierbarkeit der unterschiedlichen Konsistenzmodelle mit TGOS näher beleuchtet, sowie gegebenenfalls deren Eignung für eine verteilte virtuelle Welt untersucht werden.

4.3.1 Strikte und schwache Konsistenz

Für die *strikte Konsistenz* im TGOS-Umfeld findet die folgende Definition 7 Verwendung.

Definition 7: Strikte Konsistenz
Jede Leseoperation auf ein Objekt X liefert eine Version des Objektes zurück, welche identisch ist mit der letzten Schreiboperation auf das Objekt X. Es existiert zusätzlich eine totale Ordnung aller Schreib- und Leseoperationen.

Um diese Definition umzusetzen, muss sichergestellt werden, dass bei einem Lesezugriff auf ein Objekt immer die neuste Version zurückgeliefert wird und alle Zugriffe total geordnet werden. Die einfachste Lösung ist in Abbildung 4.1 verdeutlicht, in der jeder Lese- beziehungsweise Schreibzugriff durch eine Sperre gesichert wird. Dadurch ist sichergestellt, dass ein Leser immer die aktuellste Version des Objektes liest.

Definition 8: Sequentielle Konsistenz
"Die Resultate jeder beliebigen Ausführung sind dieselben wie in dem Fall, dass die von allen Prozessen an der Replikationsschicht vorgenommenen (Lese- und Schreib-)Operationen in einer bestimmten sequentiellen Anordnung erfolgt sind und die Operation jedes einzelnen Prozesses in dieser Sequenz in der von seinem Programm vorgegebenen Reihenfolge."

Tanenbaum & van Steen [TvS08]

Listing 4.1: Umsetzung der strikten Konsistenz mit TGOS

```
...
Sync(X);
  X.z = 10;
Push(X);
...
Sync(X);
  y = x.z;
Push();
...
```

Die *sequentielle Konsistenz* [AG96], welche in Definition 8 beschrieben wird, stellt ein weiteres wichtiges Konsistenzmodell dar. Betrachtet man die Definition des TGOS-Modells - und dort insbesondere die Definition der Randbedingungen (siehe 3.2.6), welche eine Replikationsschicht erfüllen muss - so fällt besonders die geforderte partielle Ordnung der Operationen auf.

Durch diese vorgegebene Ordnung der Basisoperationen wird automatisch eine sequentielle Konsistenz definiert, die bestimmt, dass die Operationen eines Knotens immer in der Reihenfolge seiner initialen Ausführung geschehen müssen.

Definition 9: Schwache Konsistenz
Als schwache Konsistenz wird jede Konsistenz definiert, die schwächer als sequentielle Konsistenz ist.

Während für die strikte Konsistenz im Wesentlichen nur eine Definition existiert, gibt es für den Begriff der *schwache Konsistenz* verschiedene Definitionsansätze. Für die Betrachtung der Konsistenzmodelle mit TGOS soll die Definition in 9 verwendet werden.

Listing 4.2: Umsetzung der schwachen Konsistenz mit TGOS

```
...
X.z = 10;
Push(X, keine Ordnung);
X.z = 15;
Push(X, keine Ordnung);
X.z = 20;
Push(X, keine Ordnung);
...
```

Ein Umsetzung eines schwachen Konsistenzmodells ist in Abbildung 4.2 dargestellt, indem auf die standardmäßige partielle Ordnung der Ereignisse verzichtet wird. Das Beispiel wäre auch mit der Invalidierungs-Operation gültig.

4.3.2 Transaktionale Konsistenz

Im Rahmen der Arbeit wurde auch die Eignung der *transaktionalen Konsistenz* für den Einsatz innerhalb einer verteilten virtuellen Welt untersucht. Insbesondere für die Konsistenzierung kritischer Operationen bietet sie eine Alternative zu klassischen strikten Konsistenzmodellen, wie beispielsweise Sperren oder Semaphoren.

Die transaktionale Konsistenz kann entweder in *pessimistischer* oder *optimistischer* Form realisiert werden. Beim pessimistischen Ansatz wird eine hohe Konfliktwahrscheinlichkeit angenommen und die Zugriffe werden daher mit Hilfe von Sperren synchronisiert. Das Verfahren wird sehr häufig in transaktionalen Datenbanksystemen eingesetzt, hat aber den Nachteil, dass es aufgrund der Sperren zu Verklemmungen kommen kann. Im Gegensatz zur pessimistischen Variante, die jeden Zugriff auf ein Datum mit einer Schreib- und/oder Lesesperre schützt, wird im optimistischen Modell der Zugriff lediglich protokolliert und in einer Lese- respektive Schreibmenge vermerkt. Zusätzlich wird vor einem schreibendem Zugriff das betreffende Datum kopiert, um im Falle eines Transaktionsabbruches die Änderungen rückgängig machen zu können. Diese Sicherheitskopien werden in der Literatur auch *Schattenkopien* (engl. shadow copies) genannt. Erst am Ende einer Transaktion wird in einer Validierungsphase geprüft, ob die Transaktion mit anderen Transaktionen in Konflikt steht und gegebenenfalls abgebrochen werden muss. Dabei wird zwischen einer Vorwärts- beziehungsweise Rückwärtsvalidierung unterschieden. Bei ersterer vergleichen alle laufenden Transaktionen die Schreibmenge der validierenden Transaktion mit ihren eigenen Lese- beziehungsweise Schreibmengen und brechen sich im Falle von Überschneidungen ab und starten erneut. Bei der Rückwärtsvalidierung prüft die validierende Transaktion, ob seit ihrem Start weitere Transaktionen erfolgreich committet wurden und ob deren Schreibmenge mit ihrer eigenen Lesemenge in Konflikt steht. Ist dies der Fall, so führt die Transaktion einen Rollback aus und startet erneut. Ein weiteres Unterscheidungskriterium ist die Art und Weise, wie die allfälligen Änderungen an den Daten kommuniziert werden. Dabei kann zwischen einer invalidierenden und einer aktualisierenden Form unterschieden werden. Bei der Ersten werden die durch die Schreibmenge einer committeten Transaktion spezifizierten Daten als ungültig markiert und müssen beim ersten Zugriff aktualisiert werden. Werden neben der Schreibmenge auch gleich die aktualisierten Daten verbreitet, so spricht man von einem aktualisierenden Verfahren. Eine der ersten Arbeit, die sich mit optimistischen Transaktionen beschäftigt, wurde 1981 von

Kung und Robinson[KR81] veröffentlicht.

4.3.2.1 Transaktion bei hoher Latenz

Besonders interessant ist das Verhalten der Transaktionen bei hohen Latenzen. Im Falle einer verteilten Welt treten diese sowohl durch die Nutzung von Weitverkehrsnetzen (engl. wide area networks, WAN), als auch durch Zugriff mit mobilen Geräten, die beispielsweise Funknetze (IEEE 802.11 b/g/n oder UMTS, Edge, etc.) einsetzen, auf. Die höheren Latenzen schränken dabei die Nutzung von verteilten Transaktionen durch die daraus resultierende verringerte Validierungs- beziehungsweise Sperrrate erheblich ein. Bei einer hypothetischen mittleren Paketumlaufzeit (engl. round trip time) von 10ms können im besten Fall 100 globale Validierungen oder Sperranforderungen pro Sekunde durchgeführt werden. Im Gegensatz zu pessimistischen Protokollen können optimistische Transaktionen ohne Netzwerkverkehr parallel arbeiten; Kommunikation muss erst in der Validierungsphase erfolgen. Zwar müssen Transaktionen bei einem Konflikt erneut ausgeführt werden, wofür zusätzliche Rechenzeit benötigt wird, jedoch fällt dies, im Vergleich mit der Wartezeit für eine Sperre, insbesondere bei kurzen Transaktionen nicht so sehr ins Gewicht.

Bei optimistischen Transaktionsprotokollen können Transaktionen, welche nur lesen oder nur auf knotenexklusiven Daten arbeiten, ohne eine globale Validierung auskommen. Daraus folgt, dass sich Transaktionen gut zur Konsistenzierung von Daten eignen, welche nur selten geschrieben, aber häufig gelesen werden. Dies trifft in verteilten Welten besonders auf den Szenengraphen zu, da dessen Struktur zwar häufig gelesen, aber nur selten verändert wird.

Je nachdem, welcher Mechanismus zur Synchronisierung der Valdierungsphase gewählt wird, lassen sich weitere Optimierungen vornehmen. Ein Beispiel hierfür ist ein tokenbasierter Mechanismus [IEE89], bei dem ein abstraktes Token zwischen den teilnehmenden Rechnern zirkuliert und nur der Rechner im Besitz desselben einen Commit durchführen kann. Dabei stellt gerade die gemeinsame Nutzung des Tokens den größten Flaschenhals für die verteilten Transaktionen dar. Geht man nun von einer auf Client/Server-Architektur basierenden Replikationsschicht aus, so muss sowohl die Latenz zum Server, als auch die Latenz vom Tokenbesitzer zum Server in Betracht gezogen werden. Im schlimmsten Fall benötigt ein validierender Knoten für die Inbesitznahme des Tokens dadurch die doppelte Paketumlaufzeit. Eine Verbesserungsmöglichkeit bieten spezielle server-gestützte Commit-Protokolle, wie sie beispielsweise auch bei OSS [MMS10] Verwendung finden. Nachteilig ist jedoch anzumerken, dass durch die Vermischung von Replikationsschicht und Algorithmus ein Austausch der Replikationsschicht erschwert wird. Es ist jedoch möglich, beide Ansätze bereitzustellen und dynamisch, je nach Gegebenheit, zwischen beiden hin und her zu wech-

seln.

4.3.2.2 Transaktionale Konsistenz mit TOGS

Im Rahmen der Arbeit wurde eine optimistische Variante der transaktionalen Konsistenz entworfen und prototypisch implementiert, welche ausschließlich die von TGOS bereitgestellten Basisoperationen und Ereignisse nutzt. Die Entscheidung für ein optimistisches Verfahren wurde aufgrund der im vorherigen Abschnitt beschriebenen Vorteile in einer Umgebung mit hoher Latenz, was insbesondere auf virtuelle Welten zutrifft, getroffen.

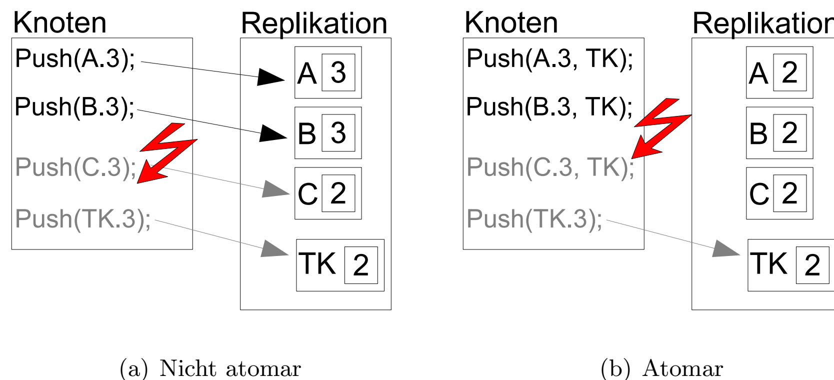
Im Gegensatz zu den gängigen Systemen verwenden die mit TGOS realisierten Transaktionen, im Weiteren auch *TGOS-Transaktionen* genannt, kein Invalidierungsprotokoll, sondern stattdessen Aktualisierungen. Dadurch sollen die zusätzlichen Kosten, die bei größeren Latenzen entstehen, vermindert werden, da Knoten, welche die Aktualisierung benötigen, sich eine Anfrage sparen. Auch würde es ein Invalidierungsprotokoll nötig machen, beim ersten Zugriff auf ein Objekt innerhalb einer Transaktion dessen Gültigkeit zu prüfen und gegebenenfalls eine neue Version anzufordern. Jedoch ist das Abfangen von Objektzugriffen in vielen stark getypten und objektorientierten Sprachen, wie beispielsweise Java, eine komplexe und nichttriviale Angelegenheit; weitere Informationen zu diesem Thema finden sich in Abschnitt 4.3.2.3. Zusätzlich muss eine Transaktion beim Empfang einer Commit-Nachricht prüfen, ob es einen Konflikt gibt und gegebenenfalls die Ausführung abbrechen, was wiederum in vielen Sprachen nicht möglich ist, da beispielsweise ein direkter Zugriff auf den Instruktionszeiger nicht möglich ist. Die Verwendung eines Invalidierungsprotokolls könnte daher nur durch Erweiterung des Compilers (abfangen der Objektzugriffe, Zugriff auf den Kellerspeicher) oder durch Anpassung der Laufzeitumgebung realisiert werden.

Listing 4.3: Definition des TGOS-Tokens

```java
public class Token extends Wurzelobjekt {
  int taNum;
  int writeSets[][] = new int[128][];
}
```

Um die Validierungsphase der Transaktionen zu synchronisieren, nutzen die TGOS-Transaktionen einen Tokemechanismus; nur der Knoten, der im Besitz des Tokens ist, ist berechtig, ein Commit durchzuführen. Das in Abbildung 4.3 dargestellte Token ist dabei als gewöhnliches TGOS-Wurzelobjekt realisiert. Ein Knoten ist im Besitz des Tokens, falls er die durch die TGOS *Sync*-Operation realisierte Sperre hält. Die Sync-Operation sperrt nicht nur das Objekt, sondern sorgt auch dafür, dass das Objekt in der Objektsicht der

aktuellsten Version der Replikationsschicht entspricht. Das Token enthält neben der aktuellen Transaktionsnummer *taNum* eine variable Anzahl an vorangegangenen Schreibmengen. Die aktuelle Schreibmenge wird durch eine Modulo-Operation mit der maximalen Anzahl an speicherbaren Schreibmengen ermittelt (=writeSets[taNum % 128]). Eine solche Schreibmenge besteht aus den global eindeutigen TGOS-Objektidentifikatoren, welche im Beispiel als Integer realisiert sind, aller im Verlauf der Transaktion geschriebenen Objekte. Bei einem Commit erhöht die validierende Transaktion den Commitzähler und trägt seine Schreibmenge, unter Verwendung der Modulo-Operation, in das Token ein. Der Tokenbesitzer gibt das Token durch die *Push*-Operation des TGOS-Modells wieder frei und aktualisiert damit auch die Replikationsschicht.



(a) Nicht atomar

(b) Atomar

Abbildung 4.12: Atomare Ausführung des Commits

Zusätzlich zum Token müssen auch alle innerhalb der Transaktion geschriebenen Objekte durch Push-Operationen aktualisiert werden. Sowohl das Aktualisieren des Tokens, als auch das Aktualisieren der Objekte muss dabei atomar erfolgen. Andernfalls könnte, wie in Abbildung 4.12(a) dargestellt, ein Absturz oder Verbindungsabbruch des validierenden Knotens dazu führen, dass nur ein Teil der Objekte aktualisiert wird, was Inkonsistenzen zur Folge hätte. Um dies zu vermeiden, werden, wie in Abbildung 4.12(b) zu sehen, die Aktualisierungen der geänderten Objekte mit der Aktualisierung des Tokens verkettet. Da die Ausführung einer verketteten Operation im TGOS-Modell atomar definiert ist, ist somit auch die Validierung atomar.

Wie eingangs erwähnt, enthält das Token eine beschränkte Anzahl an vorherigen Schreibmengen, um die Größe des Tokens zu beschränken. Dies birgt jedoch die Gefahr, dass Knoten, welche mehr Schreibmengen als im Token gespeichert sind, verpasst haben, sich in einem inkonsistenten Zustand befinden, von dem sie sich nicht mehr erholen können. Die einzige Lösungsmöglichkeit für den betroffenen Knoten ist, alle transaktional gesicherten Daten zu verwerfen und diese neu anzufordern.

Die TGOS-Transaktionen definieren fünf Methoden, die verwendet werden, um Änderungen an Objekten in Transaktionen einzubetten. Das Erstellen der Lese- und Schreibmengen muss dabei durch den Benutzer von Hand vorgenommen werden. Möglichkeiten zur automatischen Erstellung werden im nächsten Abschnitt besprochen.

add2WriteSet(wObject) fügt ein Objekt der Schreibmenge der Transaktion hinzu und erstellt gleichzeitig eine Schattenkopie des Objektes. Die Methode darf nur innerhalb einer Transaktion ausgeführt werden.

add2ReadSet (wObject) fügt ein Objekt der Lesemenge der Transaktion hinzu, es wird keine Schattenkopie erstellt. Diese Methode darf ebenfalls nur innerhalb einer Transaktion ausgeführt werden.

Begin() startet eine Transaktion und integriert alle empfangenen Aktualisierungen, um einen neuen konsistenten Zustand zu generieren.

Abort() bricht eine Transaktion ab und führt einen Rollback durch, in dem alle durch die Transaktion vorgenommenen Änderungen mit Hilfe der Schattenkopien rückgängig gemacht werden.

Commit() → (Boolean) versucht, eine Transaktion zu commiten. Im Erfolgsfall liefert die Methode Wahr zurück, andernfalls wird automatisch ein Rollback durchgeführt und Falsch zurück gegeben.

Das einfache Beispiel 4.4 demonstriert den Einsatz der zuvor beschriebenen Methoden der TGOS-Transaktion. Der Programmausschnitt zeigt eine Methode, die ein Element aus der übergebenen, einfach verketteten Liste entfernt. Die gesamte Löschoperation wird dabei durch eine Transaktion abgesichert, um einen konsistenten parallelen Zugriff durch mehrere Knoten zu ermöglichen. Jedes Element der Liste besitzt einen *next*-Zeiger auf das nächste Element. Die Routine bekommt das erste Element der Liste als Parameter *list* sowie das zu löschende Element *toDelete* übergeben.

Ein wesentliches Merkmal dieser Transaktionen stellt die Verwendung des *do..while*-Konstruktes dar. Dies ist nötig, um die Transaktion zu wiederholen, falls ein Commit aufgrund einer Kollision fehlschlägt. Eine Möglichkeit, dieses Konstrukt automatisch und für den Benutzer unsichtbar zu implementieren, wird im folgenden Abschnitt 4.3.2.3 beschrieben.

4.3.2.3 Automatische Lese- & Schreibmengen-Aufzeichnung

Für die automatische Aufzeichnung der Lese- & Schreibmenge müssen sowohl die lesenden als auch die schreibenden Zugriffe auf Objekte, beziehungsweise deren Inhalte, protokolliert werden. Dies ist in objektorientierten Sprachen und deren Laufzeitumgebungen beispielsweise durch den durchgängigen Einsatz von *Abfrage-* und *Änderungsmethoden* (engl. Getter & Setter) und

Listing 4.4: Transaktionierung einer Listenoperation

```
void deleteElem(Element list, Element toDelete) {
  do {
    TA.begin()
    TA.add2ReadSet(list);
    Element prev = list;
    Element curr = list.next;
    // durchsuche List nach dem Objekt
    while (curr!=null) {
      TA.add2ReadSet(curr);
      if (curr==toDelete) break;
      prev = curr;
      curr = curr.next;
    }
    // nicht gefunden, dann abbrechen
    if (curr==null) {TA.abort();return;}
    // element löschen
    TA.add2WriteSet(prev);
    prev.next = curr.next;
  } while (!TA.commit());
}
```

einer darin gekapselten manuellen Protokollierung möglich. Eine andere Variante, im Falle der Sprache Java, stellt die Nutzung des *Debug-Interface* dar, mit dem Zugriffe auf einzelne Variablen einer Klasse abgefangen werden können; der Einsatz dieses Verfahrens benötigt aber unverhältnismäßig viel Rechenzeit, weswegen ein praktischer Einsatz ausgeschlossen werden kann.

Eine weitere Möglichkeit stellen die *aspektorientierten* Programmierparadigmen [EFB01] dar, welche orthogonal zum bestehenden Code spezielle Routinen direkt in den Zwischencode einer Laufzeitumgebung, wie beispielsweise Java-ByteCode oder .Net-CIL, einweben können. Bekanntester Vertreter im Java-Umfeld stellt AspectJ [KEHMKG01] dar, das eine Vielzahl sogenannter *Point-Cuts* bereitstellt, um sich in den bestehenden Programmcode einzuweben. Durch diese Technik ist es möglich, dem bestehendem Code eine Lese- & Schreibmengen-Aufzeichnung hinzuzufügen, die automatisch gerufen wird, falls Objektdaten gelesen, beziehungsweise geschrieben, werden. Auch lassen sich mit dieser Technik die im vorherigen Abschnitt beschriebenen do..while-Schleifen automatisch in den Code integrieren. Im Vergleich zur vorher beschriebenen, die Java-Debuggerschnittstelle nutzenden Variante, ist der aspektorientierte Ansatz deutlich performanter. Dennoch gibt es auch hier einige Einschränkungen, insbesondere der von TGOS definierte

automatische Hüllenbildungsmechanismus verursacht Probleme.

Angenommen ein Wurzelobjekt (siehe Abschnitt 3.2.4) besitzt eine Referenz auf ein weiteres, lokales Datenobjekt. Würde nun dieses geändert, so könnte der Zugriff darauf zwar protokolliert werden, jedoch muss zusätzlich noch das referenzierende Wurzelobjekt ermittelt werden. Eine Möglichkeit wäre, die Zuweisung einer Referenz auf ein lokale Objekt an eine Objektvariable des globalen Objektes abzufangen und somit die Rückreferenz automatisch zu ermittelt. Jedoch werden so nur Objekte erfasst, die direkt durch das globale Objekt referenziert werden. Die transitive Hülle für beliebige lokale Objekte zu finden ist - ohne Compiler oder Laufzeitunterstützung - nur sehr umständlich und zeitaufwendig möglich. Eine Lösungsmöglichkeit in Verbindung mit einem angepassten Compiler sind beispielsweise Rückreferenzen [GFSS03], bei denen jedes Objekt für jede Referenz auf sich immer eine korrespondierende Rückreferenz besitzt, die auf das referenzierende Objekte zeigt. Der Mechanismus ist aus Sicht des Anwenders dabei völlig transparent.

Eine vollständige automatische und generische Aufzeichnung der Lese- & Schreibmenge ist innerhalb des TGOS-Modells nur sehr schwer möglich. Insbesondere eine adäquate Ausführungsgeschwindigkeit für den allgemeinen Fall zu erzielen, gestaltet sich schwierig, falls eine unveränderte Laufzeitumgebung verwendet wird.

4.3.3 Konsistenzdomänen

Neben der Möglichkeit, unterschiedliche Konsistenzmodelle zu generieren, gibt es auch die Möglichkeit, ein bestehendes Konsistenzmodell in mehrere disjunkte Bereiche, die Konsistenzdomänen, aufzuteilen. Dieser Ansatz ist, inbesondere für Konsistenzmodelle mit hohem Synchronisierungsaufwand, eine Möglichkeit, eine bessere Skalierbarkeit zu erreichen.

Das Konzept lässt sich beispielsweise gut auf die im vorherigen Abschnitt beschriebene transaktionale Konsistenz mit TGOS anwenden. Würden alle Knoten in der verteilten Welt auf ein einzelnes Token zur Synchronisierung angewiesen sein, so wäre der Skalierbarkeit, wie leicht zu sehen ist, enge Grenzen gesetzt. Betrachtet man die virtuelle Welt, so existiert durch das Szenenkonzept bereits eine Partitionierung, die für die transaktionale Konsistenz genutzt werden kann. Zu diesem Zweck wird jeder Szene ein eigenes Tokenobjekt zugeordnet, welches für Transaktionen innerhalb einer Szene für die Validierung genutzt werden kann. Durch diese Aufteilung können Transaktionen, die sich auf disjunkte Szenen beziehen, parallel ausgeführt werden, ohne dass Reibungsverluste auftreten.

Müssen Änderungen durchgeführt werden, die mehrere Konsistenzdomänen betreffen und in allen zur selben Zeit konsistent durchgeführt werden müssen, so werden *domänenübergreifende Transaktionen* benötigt. Dies tritt auf, falls die Dömanen nicht auf disjunkten Daten arbeiten, es an den

Rändern der Domänen Überschneidungen oder es Referenzen zwischen den Domänen gibt (beispielsweise eine kontinuierliche Landschaft, die in verschiedene Domänen aufgeteilt ist). In einem solchen Fall müssen zur Validierung die Token aller betroffenen Domänen im Besitz des validierenden Knoten sein, um einen Commit durchzuführen. Die Aggregation der Token kann jedoch zu Verklemmungen führen, wenn beispielsweise zwei Transaktionen, welche auf denselben Domänen arbeiten, die Token in unterschiedlicher Reihenfolge akquirieren. Eine mögliche Lösung ist, die Token der Domänen total zu ordnen und eine Akquirierung nur in einer auf- oder absteigenden Reihenfolge zuzulassen, bei der es keine Überschneidungen zwischen verschiedenen Transaktionen geben darf.

4.3.4 Szenenspezifische Konsistenz mit TGOS

Die Konsistenz der Daten einer Szene - und insbesondere die Konsistenz der Animationen oder des dramaturgischen Ablaufs - obliegen dem jeweiligen Szenendesigner und der von ihm implementierten *szenenspezifischen Konsistenz.*

Im Gegensatz zu einem allgemeinen Konsistenzmodell, wie beispielsweise der transaktionalen oder der sequentiellen Konsistenz, wird die szenenspezifische Konsistenz speziell für einen bestimmten Verwendungszweck definiert und optimiert. Insbesondere die Einbeziehung bestimmter Eigenarten einer Szene (Bewegung nur in zwei Dimensionen möglich, eingeschränkte Physik, Werte müssen nicht immer global eindeutig sein, etc.) erlauben eine optimalere Konsistenzierung, als es mit einem unangepassten Standardmodell möglich wäre.

Weiterführende Informationen finden sich in Abschnitt 4.4, wo eine szenenspezifische Konsistenz anhand eines einfachen Beispiels umgesetzt wird.

4.4 Entwurf einer Szene

Am Beispiel eines einfachen Spiels soll der Entwurf und die Implementierung einer Szene innerhalb einer mit TGOS verteilten Welt demonstriert werden. Als Inhalt findet ein in Wissenheim Worlds implementiertes Volleyballspiel Verwendung, das in Abbildung 4.13 dargestellt ist. Um die Komplexität möglichst gering zu halten, sollen nur die wesentlichsten Teile der Spielmechanik konzeptionell umgesetzt und beschrieben werden.

4.4.1 Voraussetzungen

Abbildung 4.14 skizziert die wichtigsten Elemente des Spiels. Jeder Spieler besitzt ein eigenes Spielfeld, innerhalb dessen Grenzen er sich bewegen kann. Jede Spielfigur besitzt ein Tupel, welches aus einem Impuls $\vec{i}$ besteht, einem Zeitpunkt t an dem der Impuls generiert wurde und einer Position p zum

Abbildung 4.13: Volleyballspiel in Wissenheim Worlds

Zeitpunkt t. Auf alle Objekte wirkt eine Gravitation g und sie können eine maximal Geschwindigkeit $\vec{v}_{max}$ annehmen. Obwohl die Szene 3-dimensional dargestellt wird, können sich die Spieler nur in einer Ebene bewegen, um das Spiel möglichst einfach bedienen zu können. An den Seitenbegrenzern A und B prallt der Ball wieder ins Spielfeld, ohne im Aus zu sein. Die Spielfiguren können sich nur zwischen dem Netz und ihrem jeweiligem Seitenbegrenzer bewegen.

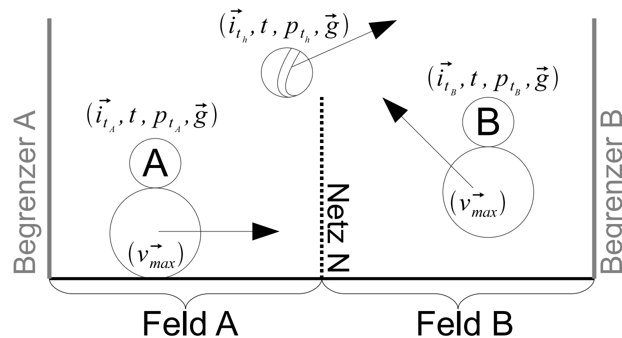


Abbildung 4.14: Basiselemente des Spiels

Ein Spieler soll seine Figur mit Hilfe der Tasten A und D nach links oder rechts bewegen und mit der Taste W springen können. Die Tastatureingaben der Spieler werden in einen Impuls umgewandelt und ihre Spielfigur damit aktualisiert, zusätzlich wird noch der Zeitpunkt und die Position vermerkt. Mit diesen Werten kann die Bewegung durch beide Spielteilnehmer extrapoliert werden. Der Ball hat einen Tupel analog zu dem der Spielfiguren, nur wird das Tupel lediglich dann aktualisiert, wenn ein Avatar mit dem Ball kollidiert und ihm dadurch einen neuen Impuls überträgt. Da die beiden Spielhälften strikt getrennt sind, kann der Ball maximal von einem Spieler

zu jedem Zeitpunkt berührt werden.

Die Entscheidung, ob der Ball zu Boden gefallen ist, kann ebenfalls von jedem Knoten vorgenommen werden. Der Knoten, auf dessen Feld der Ball liegt, bestimmt das Ergebnis.

Zusätzlich zur Steuerung der Spielfiguren und zur Berechnung der Bewegung des Balles muss auch der Spielzustand und die Zuordnung der Avatare verwaltet werden. Ein Spieler kann Spielfigur A oder B übernehmen, indem er auf sie klickt und falls diese Figur nicht bereits von einem anderen Teilnehmer kontrolliert wird. Sobald beide Spielfiguren vergeben sind, wird das Spiel gestartet, wobei Spieler A den Ball erhält.

4.4.2 Aufbau des Szenengraphens

In Abbildung 4.15 sind die für die Szene definierten Klassen dargestellt. Die *Spielfigurklasse* erbt dabei von der *Verbundklasse* des Szenengraphen und kann somit in die Struktur des Szenengraphen integriert werden. Gleiches gilt für die Klasse *Volleyball*, welche eine Szene spezialisiert. Zusätzlich implementiert die Volleyballklasse die Schnittsstelle für Szenenereignisse und die Spielfigurklasse die Objektereignisse, welche in Abschnitt 4.1.2 vorgestellt wurden. Um die Darstellung zu vereinfachen, wurden die Parameter der Operationen mit **...** abgekürzt.

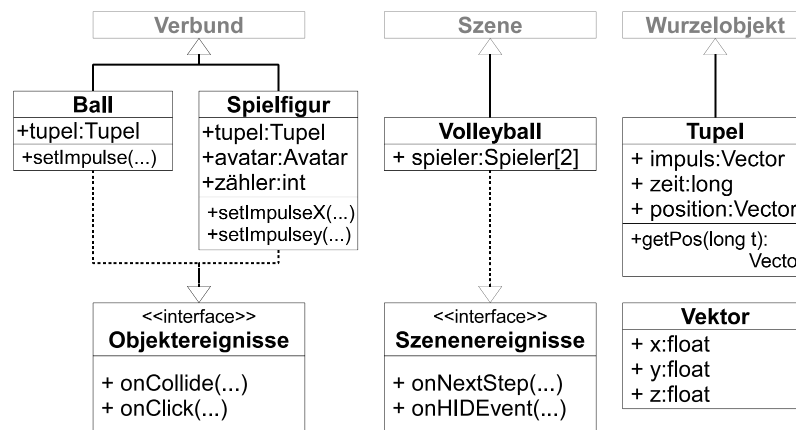


Abbildung 4.15: Klassen des Volleyballspiels

Das *Tupelobjekt* wird vom TGOS-Wurzelobjekt abgeleitet und kann somit durch die TGOS-Operationen explizit verteilt werden. Es repräsentiert das in den Voraussetzungen beschriebene Tupel aus Impuls, Zeitpunkt und Position. Die Routine *getPos* berechnet anhand der im Tupel gespeicherten Position, Impuls und Startzeitpunkt die Position zum übergebenen Zeitpunkt t und liefert diese als Rückgabe zurück.

Die Konstruktion des Szenengraphens wird im Konstruktor der Volleyballklasse definiert, welcher als Parameter eine Referenz auf den lokalen Kontext erhält. Über die damit zugreifbare Ressourcenkomponente können die für den Graphen benötigten Mediendaten geladen werden. Die Szene wird dann im Namensdienst registriert und kann so von allen anderen Knoten zugegriffen werden. Der Konstruktor, der die Szene erstellt, muss daher nur ein einziges Mal aufgerufen werden.

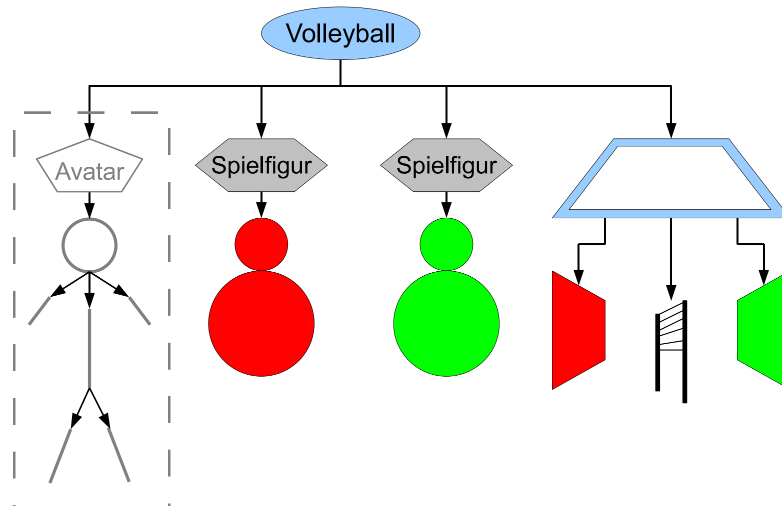


Abbildung 4.16: Szenengraph der Volleyball-Szene

In Abbildung 4.16 ist die Struktur und der Inhalt einer Instanz des Szenengraphens dargestellt. Die grauen Sechsecke repräsentieren Instanzen der Spielfigurklasse. Sowohl das Spielfeld, die Begrenzer, das Netz und die Spielfiguren sind Instanzen von Formobjekten. Im gestrichelten Rechteck ist ein Avatar dargestellt, der sich in der Szene befindet. Dabei muss es sich nicht um einen Spieler des Volleyballspiels handeln, es könnte sich auch um einen Zuschauer handeln.

4.4.3 Integration der Spiellogik

Nach der Definition der verwendeten Datenstrukturen und dem Aufbau des Szenengraphens soll nun die Spiellogik integriert werden. Diese gliedert sich in drei Bereiche: Auswahl einer Spielfigur, Behandlung von Tastaturereignissen und der Berechnung des Spielablaufes. Der Ablauf beinhaltet dabei sowohl die Bewegungen der Spielfiguren und des Balls, als auch die Berechnung des Spielstandes.

Auswahl der Spielfigur

Um die Logik für die Auswahl einer Spielfigur zu definieren, wird die

onClick()-Routine der Spielfigur genutzt. Diese wird gerufen, falls ein Teilnehmer mit der Maus auf die Form einer Spielfigur klickt. In Codebeispiel 4.5 ist die Routine dargestellt, welche von beiden Spielfiguren gemeinsam genutzt wird. In Zeile 2 wird anhand der Farbe des Formobjekts ermittelt, welche Spielfigur angeklickt wurde und der entsprechende Index ausgewählt. Da die Auswahl einer Spielfigur synchronisiert werden muss, wird die Auswahl in eine Transaktion eingebettet. In Zeile 5 wird überprüft, ob die Spielfigur schon vergeben ist, was der Fall ist, falls ein anderer Avatar eingetragen ist. Sollte dies der Fall sein, so wird die Transaktion abgebrochen und die Routine ohne Ergebnis verlassen. Andernfalls wird der Avatar, welcher vom Spieler des ereignisauslösenden Knotens gesteuert wird, als Besitzer der Spielfigur eingetragen und der Ergebniszähler auf Null zurückgesetzt.

Listing 4.5: Spielfigurauswahl beim onClick-Ereignis

```
1 public void onClick(Object o, Ctx ctx) {
2   int idx = o.color==GREEN ? 0 : 1;
3   do {
4     tgosTA.begin()
5     if (Spieler[idx].avatar!=null) {
6       tgosTA.abort();
7         break;
8       }
9       tgosTA.add2WriteSet(Spieler[idx]);
10    Spieler[idx].avatar=ctx.getAvatar();
11    Spieler[idx].zähler=0;
12  } while(!tgosTA.commit())
13 }
```

Behandlung der Tastatureingaben

Als nächstes wird die *onHIDEvent*-Routine der Klasse Volleyball definiert, welche durch das UI bei einem Tastendruck gerufen wird. In dieser Routine werden die Tastatureingaben ausgewertet und die Spielfigur entsprechend gesteuert, falls der Avatar des Knotens im Besitz einer Spielfigur ist. In Codebeispiel 4.6 ist der Aufbau der Routine skizziert.

Im ersten Abschnitt von Zeile 2 bis Zeile 7 wird geprüft, welche Spielfigur dem Avatar zugeordnet ist, oder ob es sich nur um einen Zuschauer handelt, der nicht am Spiel teilnimmt. Im Zeile 10 bis 15 wird für die Bewegung nach links und rechts bei einem Tastendruck der Impuls gesetzt oder gelöscht. Der Impuls für das Springen, ausgelöst durch Druck auf die Taste W, in Zeile 16 und 17 wird nur gesetzt, falls sich

Listing 4.6: Behandlung der Tastaturereignisse

```
1 public void onHIDEvent(HIDEvent evt, Ctx ctx) {
2  //-- Spielfigur finden
3   Spieler s = null;
4   for (int i=0;i<2;i++)
5     if (Spieler[i].avatar==ctx.getAvatar())
6       s = Spieler[i];
7   if (s==null) return;
8  //-- Impuls nach Taste setzen
9   switch(evt.getKey()) {
10  case 'A': s.setImpulseX(evt.isPressed() ? -1 : 0,
11                            ctx.getTimeStep());
12            break;
13  case 'D': s.setImpulseX(evt.isPressed() ? 1 : 0,
14                            ctx.getTimeStep());
15            break;
16  case 'W': if (evt.isPressed() &&
17                  s.isOnGround()) s.setImpulseY(1);
18            break;
19  }
20 }
```

der Avatar auf dem Boden befindet. Ein Zurücksetzen des Sprungimpulses ist logischerweise nicht möglich.

Die Routine *setImpulsX* und *setImpulsY* setzt jeweils das Bewegungstupel, getrennt für die X- und Y-Achse. Als Parameter erhalten sie die Stärke und Richtung des Impulses, sowie den aktuellen Zeitpunkt. Die für die Aktualisierung des Tupel benötigte gegenwärtige Position wird der Transformation der Spielfigur entnommen.

Berechnung des Spielablaufes

Nachdem die Spieler nun die Bewegungsinformationen ihrer Spielfiguren setzen können, müssen diese noch ausgewertet und berechnet werden. Dies wird in der Ereignisbehandlungsroutine *onNextStep* der Szene realisiert, die für jeden Zeitschritt und auf jedem Knoten gerufen wird. In Auflistung 4.7 ist der grobe Aufbau der Routine dargestellt. Aufgaben, die zwar ausgeführt werden, aber nicht in Form von Pseudocode dargestellt sind, werden durch Klammerung mit **...** dargestellt. Die Transformation eines Szenengraphenobjekts ist aus Platzgründen mit *trans* abgekürzt.

In Zeile 2 wird die neue Position des Balls anhand seines Bewegungstupels generiert und seine Transformation aktualisiert. Danach wird die

Listing 4.7: Berechnung des Spiels

```
1 public void onNextStep(long timeStep, Ctx ctx) {
2   ball.trans.setPos(ball.tupel.getPos(timeStep);
3   for (Spieler s : spieler) {
4     if (s.avatar==ctx.getAvatar()) {
5       s.trans.setPos(s.tupel.getPos(timeStep));
6       ... Position auf Spielfeld begrenzen ...
7       TGOS.push(s.trans);
8       if (ball.collidesWith(s)) {
9         ball.setImpuls(s);
10        TGOS.push(ball.tupel);
11      }
12    }
13  }
14  ... Berechnung ob Ball den Boden berührt hat ...
15   ...            Ergebnisberechnung             ...
16 }
```

Position der Spielfiguren analog zur Position des Balls neu berechnet. In Zeile 8 wird geprüft, ob die Spielfigur mit dem Ball kollidiert und falls ja, der Impuls des Balls aktualisiert. Im einfachsten Fall prallt der Ball mit einer konstanten Beschleunigung von der Spielfigur in entgegengesetzter Richtung ab. Je nach Güte des verwendeten physikalischen Modells können bei der Berechnung des Impulses weitere Kriterien miteinbezogen werden, wie beispielsweise die Oberfläche der Spielfigur. Zusätzlich findet in der Routine die Berechnung des Spielstandes statt. Da der Ball nur in genau einer Spielhälfte auf den Boden gelangen kann, wird diese Berechnung jeweils von dem Knoten ausgeführt, auf dessen Spielfeld der Ball liegt.

Innerhalb der Routine findet sich auch die Ausgestaltung der szenenspezifischen Konsistenz, da dort der Szenengraph mit Hilfe von TGOS-Operationen global aktualisiert wird. Im Beispiel tritt dies in Zeile 7 und Zeile 10 auf. Jeder Knoten berechnet die Position des Balls anhand des Impulses lokal. Erfährt der Ball eine Impulsänderung durch eine Berührung mit einer Spielfigur, so wird diese Änderung mit Hilfe der Push-Operation global aktualisiert. Da immer nur genau eine Spielfigur den Ball zu jedem Zeitpunkt berühren kann, kann es trotz der verwendeten sequentiellen Konsistenz nicht zu Inkonsistenzen kommen. Da bei jeder Aktualisierung des Ballimpulses auch die Startposition aktualisiert wird, fallen numerische Instabilitäten nicht ins Gewicht.

Im Gegensatz zum Ball, wird die Position der Spielfiguren immer nur

durch deren Besitzer berechnet und dann an alle anderen Knoten verteilt. Dies geschieht mit Hilfe der Push-Operation in Zeile 7. Dadurch wird für die Position eine Form der *Eventual Consistency* umgesetzt, da die neue Position zwar verzögert auf den teilnehmenden Knoten ankommt, aber ein konsistenter Zustand auf alle Fälle erreicht wird. Es wäre auch hier möglich, analog zum Ball nur die Bewegungsvektoren global zu aktualisieren oder eine beliebige andere Konsistenz zu verwenden.

Die Konsistenz des Balls stützt sich aufgrund der zeitbasierten Berechnung stark auf die Synchronität der Zeit t zwischen den teilnehmenden Knoten ab, benötigt für eine korrekte Ausführung aber keine kausale Abhängigkeit. Daher ist die durch die Zeitkomponente bereitgestellte globale Zeit auch für das Volleyballspiel als ausreichend zu betrachten (siehe Abschnitt 4.2).

4.4.4 Zusammenfassung

Das Beispiel verdeutlicht sehr anschaulich den netzarchitekturunabhängigen Ansatz von TGOS. Operationen werden immer in Bezug auf die Datenstruktur definiert und auch die globalen Operationen beziehen sich direkt auf Elemente im Szenengraphen.

4.5 Last- und Replikationsmanagement

Das TGOS-Modell als solches definiert nur sehr wenige Randbedingungen für eine geeignete Replikationsschicht. Dennoch kommt ihr für den Betrieb einer virtuellen verteilten Welt eine enorme Bedeutung zu. Insbesondere Skalierbarkeit, Ausfallsicherheit und Sicherheit stellen wichtige Merkmale dar.

Die hier vorgestellte Konzeption einer Replikationsschicht basiert auf den mit Wissenheim Worlds gewonnenen Erfahrungen. Wie auch schon für die Definition des Szenengraphen, ist das hier vorgestellte Konzept nur eine mögliche Variante.

4.5.1 Lastverteilung durch Area-of-Interest

Für verteilte virtuelle Welten, die mit einer großen Anzahl an gleichzeitig aktiven Benutzern zurecht kommen müssen, sind Area-of-Interest-Algorithmen zwingend nötig. Um die Eignung von TGOS auch für diesen Bereich zu demonstrieren, sollen im Folgenden verschiedene Konzepte für die Umsetzung einiger Area-of-Interest-Mechanismen dargelegt werden, die sich ausschließlich der von TGOS bereitgestellten Operation bedienen.

4.5.1.1 Statische AoI-Zuordnung

Als einfachste Form der Area-of-Interest-Verwaltung bietet sich eine statische Partitionierung in unterschiedliche Zonen an, bei der jede Zone eine in sich geschlossene Einheit bildet und bei der kein Informationsfluss über Zonengrenzen hinweg stattfindet.

Ein solcher Ansatz lässt sich mit TGOS mit Hilfe der Gruppenkommunikationsmechanismen aus Abschnitt 3.2.10 verwirklichen. Jeder Zone ist dabei genau eine Gruppe zugeordnet und alle Avatare, die sich in einer Zone befinden, haben standardmäßig diese Gruppe für alle TGOS-Operationen gesetzt. Somit werden Aktualisierungen nur an diejenigen Knoten verteilt, deren Avatare sich in der Zone befinden. Wechselt ein Avatar von einer Zone in die nächste, wechselt er zum einen die Standardgruppe und verlässt zum anderen die Empfangsgruppe. Kurzzeitig bietet es sich an, Ereignisse von beiden Gruppen zu erhalten, um beispielsweise einen fließenden Übergang zwischen beiden Zonen zu ermöglichen.

Eine noch striktere Trennung wäre auf Ebene der Replikationsschicht möglich. Diese könnte beispielsweise für jede einzelne Zone einen separaten Dienst anbieten, der sämtliche Kommunikation und Daten verwaltet. Wenn ein Nutzer eine Zone wechselt, so muss er auch den Replikationsdienst wechseln. Im striktesten Fall wäre im Gegensatz zu kommerziellen Welten, wie beispielsweise SecondLife oder World of Warcraft, kein fließender Übergang (ein Wechsel von einer Zone in die nächste, ohne Warte- oder Ladezeit) zwischen verschiedenen Zonen mehr möglich.

4.5.1.2 Dynamische AoI-Zuordnung

Die im vorherigen Abschnitt vorgestellten statischen Varianten sorgen für eine erste Partitionierung der Last. Jedoch funktionieren dieses Verfahren nur, wenn die Last auf die verschiedenen Zonen gleichmäßig verteilt ist. Befinden sich mehrere Nutzer gleichzeitig in einer einzigen Zone, so stößt der statische Ansatz an seine Grenzen. Dynamische Area-of-Interest-Mechanismen hingegen sind darauf ausgelegt, die Ausgestaltung und Granularität der Partitionierung dynamisch an die Last anzupassen.

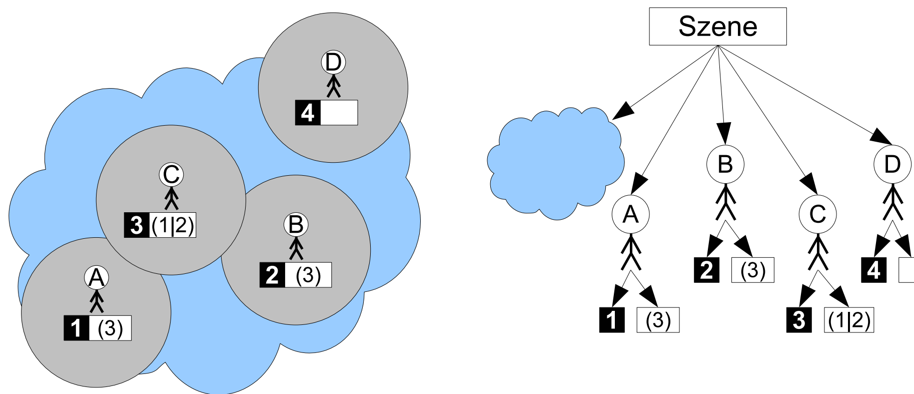
Die dynamischen Area-of-Interest-Verfahren ersetzen dabei nicht die statischen, sondern erweitern diese. So ist es sinnvoll, die Welt in statische disjunkte Zonen zu unterteilen, die sich auch in der Ausgestaltung der Replikationsschicht niederschlagen. Zusätzlich sorgen die dynamischen Varianten für eine Lastverteilung innerhalb der statischen Zonen.

Wie schon im statischen Fall, stützen sich auch die dynamischen AoI-Verfahren auf das von TGOS definierte Gruppenkommunikationsmodell. Zwei unterschiedliche Varianten für dynamische Area-of-Interest-Verwaltung sollen dabei im Rahmen der Arbeit vorgestellt werden, ein *koordinatorbasierendes* und ein *Peer-To-Peer-basierendes* Modell.

Koordinatormodelle

Koordinatormodelle sind insbesondere auch bei Peer-to-Peer-basierten Systemen häufig anzutreffen (vgl. [BKH04],...), obwohl es auf den ersten Blick diametral zum Peer-to-Peer-Gedanken erscheint. Dennoch findet es Verwendung, da die Abstimmung und Koordination, wer welchen Avatar sieht, ohne eine zentrale Komponente deutlich mehr Nachrichten benötigen würde, als bei einem vergleichbaren Koordinatormodell. Der Unterschied zum klassischen Client/Server-Modell besteht jedoch darin, dass jeder Knoten ein Koordinator werden kann und dass beim Ausfall eines Koordinators der nächste Knoten der Gruppe übernehmen kann. Der prinzipielle Ablauf wurde bereits in Abschnitt 2.3.1 beschrieben.

Der hier konzipierte Koordinator ist immer für eine Szene verantwortlich und erstellt für jeden Avatar eine Area-of-Interest variabler Größe, die in etwa dem maximalen Sichtbereich des Avatars entspricht. Der Koordinator regelt dabei nicht direkt den Nachrichtenverkehr, das heißt, er entscheidet nicht pro Ereignis, an welche Knoten dies weitergeleitet werden soll, sondern nutzt ausschließlich das von TGOS bereitgestellte Gruppenmodell zur Steuerung. Dabei kann zwischen zwei verschiedenen Varianten unterschieden werden; entweder der Bestimmung des Empfängerkreises beim Auslösen des Ereignisses, sprich welche Gruppen eine TGOS-Push-Operation als Parameter erhält, oder indem sich ein potentieller Empfänger einer für ihn relevanten Gruppe anschließt. Beim letzten Fall könnte ein Avatar seine Bewegungsänderungen immer an eine private Gruppe schicken, der sich andere Avatare in seinem Sichtbereich anschließen können.



(a) Sichtbereiche der Avatare (b) Daten im Szenengraph

Abbildung 4.17: AoI-Management mit Koordinator

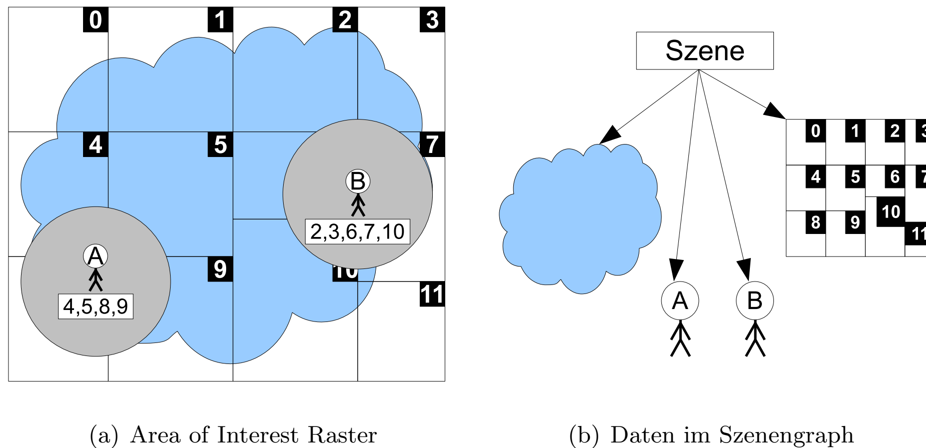
In Abbildung 4.17(a) ist das Konzept als eine einfache 2-dimensionale

Aufteilung dargestellt. Die Beschränkung auf zwei Dimensionen ist keine zwingende Vorgabe und dient hauptsächlich dem leichteren Verständnis des Sachverhaltes. Es bietet sich jedoch für Situationen an, bei denen Avatare sich auf einer nahezu planen Oberfläche bewegen und die dritte Dimension keine sinnvolle Partitionierung ermöglicht. Die Area-of-Interest eines Avatars ist als grauer, halbtransparenter Kreis (im Falle von drei Dimension wird der Kreis durch eine Kugel ersetzt) dargestellt, der Identifikator seiner eigenen privaten Gruppe als weiße Zahl auf schwarzem Grund. Avatare, deren Sichtbereiche sich mit denen anderer Avatare überschneiden, senden ihre Bewegungsaktualisierungen entweder an die private Gruppe der betreffenden Avatare oder treten diesen bei. Im Bild wird dies durch die Zahlen im weißen Kasten unterhalb der Avatare dargestellt, die den Gruppenvektor des betreffenden Avatars darstellen. In beiden Fällen nutzt der Koordinator den Szenengraphen, um die Gruppenzugehörigkeiten festzulegen, indem er den Gruppenvektor verändert. Abbildung 4.17(b) zeigt die im Szenengraph verankerten Informationen. Neben der privaten Gruppe des Avatars findet sich dort auch der Gruppenvektor, der die Gruppenidentifikatoren all jener Avatare enthält, mit denen Überschneidungen auftreten. Im ersten Fall nutzt der Knoten den Gruppenvektor, um bei einer Aktualisierung der Bewegungsinformationen diese direkt als Parameter für die Push-Operation zu nutzen und so den Empfängerkreis festzulegen. Bei der zweiten Variante würde der Knoten bei einer Aktualisierung seines Gruppenvektors durch den Koordinator ein TGOS-Aktualisierungsereignis erhalten und könnte dann seine Gruppenzugehörigkeit entsprechend dem Vektor anpassen.

Peer-to-Peer-Modelle

Bei einem reinen Peer-to-Peer-Modell existiert pro Szene kein dedizierter Koordinator, sondern die einzelnen Avatare organisieren die Gruppenkommunikation selbstständig. Eine Möglichkeit ist, die Szene in feingranulare Areas-of-Interest zu unterteilen, statt, wie beim vorherigen Ansatz, den Avataren einzelne AoIs zuzuordnen. Die Partitionierung kann dabei nach verschiedenen Methoden erfolgen, denkbar ist beispielsweise der Einsatz eines Voroni-Modells, wie in [HL04] beschrieben. Eine andere Variante wäre, eine Aufteilung nach räumlichen Gesichtspunkten (Raum durch einen Berg geteilt, innerhalb eines Gebäudes, etc.) durchzuführen. Im Gegensatz zu den statischen Verfahren, kann das Raster im Betrieb angepasst werden, indem beispielsweise stark frequentierte Bereiche weiter unterteilt oder wenig belastete Gebiete verschmolzen werden.

Abbildung 4.18(a) zeigt den Aufbau eines solchen Rasters samt der Sichtbereiche der Avatare und deren Gruppenzugehörigkeiten. Die Gruppenidentifikatoren und deren Bereichszuordnung werden dabei im Sze-



(a) Area of Interest Raster

(b) Daten im Szenengraph

Abbildung 4.18: AoI-Management ohne Koordinator

nengraphen verankert und können so von allen teilnehmenden Knoten gelesen werden. Im Gegensatz zum Koordinatoransatz, bestimmt hier der Knoten des Avatars die Größe des Sichtbereiches. Schneidet der Sichtbereich eines Avatars nun ein Feld, in welchem er noch nicht Gruppenmitglied ist, so lädt er das Gruppenobjekt und tritt damit automatisch der Gruppe bei. Verlässt er das Feld, löscht er die Referenz auf das Gruppenobjekt. Zu beachten ist, dass bei diesem Ansatz der Austritt aus einer Gruppe verzögert auftreten kann, da nach der Definition im TGOS-Modell dies erst beim Finalisieren des Objektes durch die Freispeichersammlung erfolgt. Alternativ besteht zusätzlich noch die Möglichkeit, dass der Knoten manuell aus einer Gruppe austritt, indem er die entsprechende TGOS-Funktion nutzt. Zusätzlich nutzt er die Menge der aktuellen angehörenden Gruppen als Parameter für Push-Operationen, die seine Bewegungsinformationen betreffen. Im Gegensatz zum Koordinatormodell liegen die Gruppeninformationen der Avatar, wie in Abbildung 4.18(b) zu sehen, nicht im Szenengraphen, sondern residieren nur lokal auf den einzelnen Knoten. Da das Area-of-Interest-Raster im Szenengraphen verankert ist, können die teilnehmenden Knoten dieses bei Bedarf anpassen, um beispielsweise weitere Verfeinerungen vorzunehmen. Die transaktionale Konsistenz bietet sich dann für die Konsistenzierung an.

4.5.2 Die Infrastruktur einer verteilten Welt

In diesem Abschnitt soll nun ein Konzept für den Aufbau einer Infrastruktur einer verteilten Welt vorgestellt werden. Der hier vorgestellte Aufbau der Infrastruktur für eine mit TOGS verteilte Welt basiert auf Erfahrungen, die beim Betrieb der prototypischen virtuellen Welt Wissenheim Worlds

an der Universität Ulm gesammelt wurden. Die Infrastruktur definiert alle Komponenten und Dienste, welche für den Betrieb und die Persistenz einer verteilten Welt nötig sind. Der im Folgenden vorgestellte Entwurf stellt nur eine mögliche Variante dar; für eine andere virtuelle Welt kann sich auch eine andere Strukturierung als sinnvoller erweisen.

Im ersten Abschnitt soll die Frage nach der Wahl einer geeigneten Netzarchitektur behandelt werden. Darauf aufbauend wird eine Replikationsschicht vorgestellt, sowie zusätzliche Erweiterungen definiert, die für den Betrieb nötig sind, aber im Rahmen des TGOS-Modells nicht behandelt wurden. Dazu soll zuerst eine TGOS-kompatible Replikationsschicht definiert werden, die den in vorherigen Abschnitten definierten Anforderungen gerecht wird.

4.5.2.1 Auswahl der Netzarchitektur

Eine sehr wichtige Entwurfsfrage für die Definition der Replikationsschicht und die Strukturierung der Infrastruktur ist die Frage der Netzarchitektur. Aktuelle Forschungen in diesem Bereich konzentrieren sich sehr stark auf Peer-to-Peer-Ansätze, da diese theoretisch eine bessere Skalierbarkeit und einfachere Erweiterbarkeit versprechen. Jeder zusätzliche Knoten erhöht nicht nur die Last auf das Gesamtsystem, sondern stellt gleichzeitig Ressourcen, wie Speicher und Rechenzeit, zur Verfügung.

In kommerziellen Systemen ist der Client/Server-Ansatz am weitesten verbreitet, da er die einfachste Synchronisierung und das beste Sicherheitspotential bietet. Mit der verstärkten Verbreitung allfälliger Cloud-Dienste und deren dynamischer Ressourcenverwaltung können nun auch Betreiber einer virtuellen Welt ihren Server-Cluster sehr einfach an schwankende Lastzustände anpassen. Ein weiterer Vorteil einer Client/Server-Architektur liegt in den geringeren Anforderungen an die Klienten. Die zunehmende Zahl an mobilen Klienten beispielsweise, die über eine Funkverbindungen angebunden sind und nur wenig Rechenleistung und Speicherplatz bieten, macht einen Client/Server-Ansatz attraktiv. Zusätzlich ermöglicht es Teilnehmern, die von Rechnern mit beschränkten Rechten aus operieren (zum Beispiel Rechnerpools der Universitäten), ohne Probleme an der Welt teilnehmen zu können. Bei einem reinen Peer-to-Peer-Ansatz ist es unabdingbar, dass sich andere Knoten auf den teilnehmenden Knoten verbinden können, was meistens durch Firewalls oder durch Router, die NAT [Gro01] einsetzen, erschwert oder verhindert wird.

4.5.2.2 Replikationsschicht

Die Replikationsschicht besteht aus drei Komponententypen, die sich in drei Schichten gliedern. Abbildung 4.19 zeigt einen Überblick über die Schichten und die darin verankerten Komponenten. Auf der untersten Ebene sind die

Komponenten der Peerschicht, oder *Peers*, definiert. Ein Peer implementiert dabei die durch das TGOS-Modell definierte Replikationsschnittstelle und dient als Zugangspunkt zu den Replikationsdiensten. Der Zugriff via TGOS auf die Replikationsschicht muss dabei zwingend über einen Peer erfolgen. Daten auf den Peers sind nur flüchtiger Natur und gehen beim Ausfall eines Peers verloren. Der *SuperPeer*, auch Replikationsknotenpunkt genannt, ist für die persistente Speicherung der Daten der verteilten Welt verantwortlich. Zwischen Peers und SuperPeers besteht, wie im Bild zu sehen, eine Client/Server-Beziehung. Die Kommunikation zwischen beiden kann beispielsweise über eine TCP-Verbindung abgewickelt werden. Der SuperPeer stellt auch eine Schnittstelle zur Zugriffskontrolle bereit, dargestellt als weiße Ausspa-rung, die im nachfolgenden Teilabschnitt spezifiziert wird.

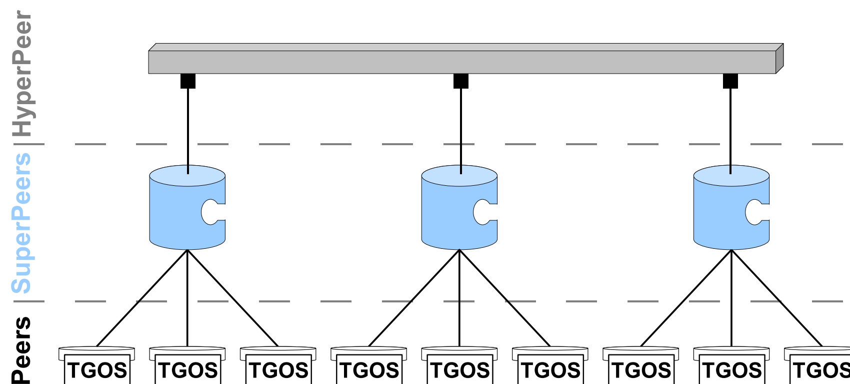


Abbildung 4.19: Replikationsschichten

Die *HyperPeer*-Schicht verbindet die einzelnen SuperPeers untereinander. Dabei ist die Implementierung des HyperPeers für die SuperPeers transparent und die Kommunikation kann wahlweise mit einem Peer-to-Peer- oder Client/Server-Ansatz realisiert werden. Der HyperPeer speichert per Definition keine Daten, sondern dient ausschließlich als Kommunikationskanal zwischen den SuperPeers.

In allen bisher betrachteten Szenarien hatten alle Knoten mit TGOS einen gleichberechtigten Zugriff auf die Replikationsschicht. Insbesondere konnten alle Knoten beliebig Daten in der Replikationsschicht aktualisieren und lesen. Im Rahmen einer virtuellen verteilten Welt kann es jedoch sinnvoll sein, Zugriffsbeschränkungen für das Ändern, beziehungsweise Auslesen, bestimmter Daten einzuführen. Zu diesem Zweck muss eine Replikationsschicht die Möglichkeit bieten, eine Kontrollinstanz zu registrieren, die über Ereignisse innerhalb der Replikationsschicht informiert wird und die es erlaubt, das Ausführen von Operationen zu verhindern oder zu modifizieren.

Beispielsweise machen es die in virtuellen Welten häufig verwendeten Zugangskontrollmechanismen, die nur autorisierten Nutzern den Zugriff auf

die Welt gestatten, nötig, schon beim Verbindungsaufbau eine Sicherheitsüberprüfung durchzuführen. In Abbildung 4.20 ist dieser Fall exemplarisch dargestellt.

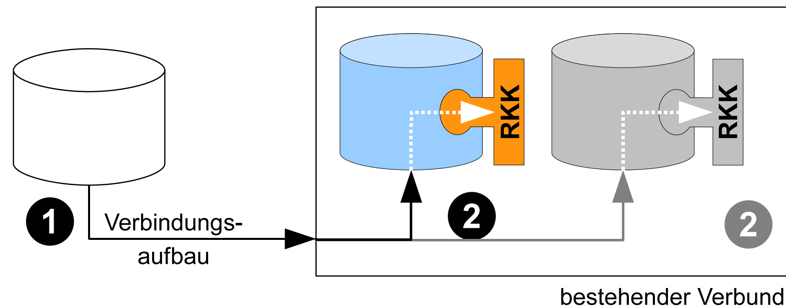


Abbildung 4.20: Replikationsereignisse am Beispiel des Verbindungsaufbaus

Knoten A möchte bei ❶ eine Verbindung zu einem bestehenden Replikationsverbund aufbauen, woraufhin die Replikationsschicht bei ❷ ein Verbindungsereignis auslöst. Dieses wird im Diagramm an die Replikationskontrollkomponente (kurz RKK) weitergereicht, welche nach einer Prüfung den Verbindungswunsch ablehnen oder bestätigen kann. Je nach Struktur der Replikationsschicht, kann das Ereignis dabei nur an einem Replikationspunkt auftreten, wie zum Beispiel bei Client/Server, oder an allen gleichzeitig, was beim Einsatz eines Peer-to-Peer-Ansatzes sinnvoll wäre.

Die Art der Ereignisse, welche durch eine Kontrollkomponente verarbeitet werden sollen, kann dabei je nach Replikationsschicht variieren. Nachfolgend wird eine kleine Anzahl an Ereignissen definiert, die für eine TGOS-kompatible Replikationsschicht sinnvoll erscheinen. Tabelle 4.2 gibt einen Überblick über die definierten Ereignisse, sowie deren Parameter, beziehungsweise Rückgabewerte. Der in der Tabelle als *Replik* bezeichnete Parameter definiert ein Tupel aus Identifikator und binärem Datenblock (siehe Abschnitt 3.2.12). Der Parameter *Quelle* beziehungsweise *Ziel* spezifiziert den Ursprung oder das Ziel der Anfrage in geeigneter Weise; denkbar wären beispielsweise die IP-Adresse und der Port des auslösenden Klienten.

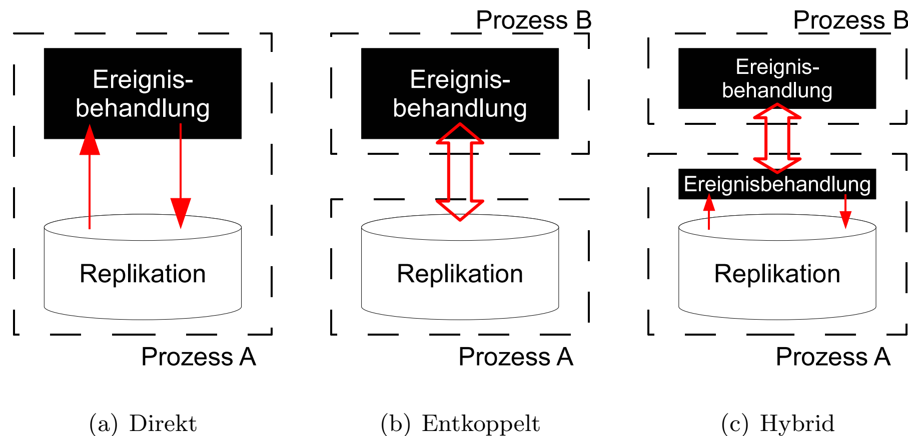
Die ersten beiden Ereignisse dienen der Steuerung der Verbindungen zur Replikationsschicht. Im Falle eines Verbindungsaufbaus soll das *Verbinden*-Ereignis ausgelöst werden, bevor die reguläre Kommunikation über die Verbindung statt findet. Der *Schlüssel*-Parameter kann einen binären Wert enthalten, welcher zur Authentifizierung und Autorisierung auf Replikationsebene verwendet werden kann. Wird Wahr zurückgeliefert, kann die Verbindung genutzt werden, bei Falsch wird sie geschlossen. Wird die Verbindung geschlossen, so tritt das *Verlassen*-Ereignis auf, welches als Parameter den Identifikator des entfernten Knotens bekommt. Die nächsten beiden Ereignisse werden genutzt, um auf Änderungen an der Replikationsschicht zu

Ereignis	Signatur
Verbinden	(Schlüssel, Adresse) → Boolean
Verlassen	(Adresse)
Schreiben	([Replik], Typ, Quelle)
Lesen	(Replik, Quelle) → Replik
Suchen	(Name, Id, Quelle) → Id
	(RegEx,[Name], Quelle) → [Name]
Registrieren	(Id, Name, Quelle) → Boolean
Aktualisieren	([Replik], Typ, Quelle, Ziel)
Invalidieren	([Replik], Typ, Quelle, Ziel)

Tabelle 4.2: Liste der Replikationsereignisse

reagieren. Das *Schreiben*-Ereignis wird ausgelöst, bevor die Repliken aktualisiert werden. Werden Repliken aus der übergebenen Liste entfernt, so werden diese nicht in der Replikationsschicht aktualisiert und lösen daher auch kein Ereignis aus. *Lesen* tritt auf, bevor eine Leseanfrage durchgeführt wird, als Parameter wird die auf die Anfrage gefundene Replik übergeben. Die zurückgegebene Replik wird dann als Antwort auf die Anfrage verwendet. Das *Suchen*-Ereignis wird vor dem Zurückliefern des Ergebnisses ausgeführt und analog zu Lesen wird der durch dieses Ereignis zurückgelieferte Wert als Antwort verwendet. *Registrieren* wird vor dem Schreiben des Verzeichniseintrages gerufen, welcher nur ausgeführt wird, falls Wahr zurückgeliefert wird. Sowohl das *Aktualisieren*- als auch das *Invalidieren*-Ereignis tritt auf, bevor die entsprechende Nachricht an den als Ziel spezifizierten Knoten versendet wird. Wie schon bei *Schreiben* werden Einträge, welche aus der Replikliste entfernt wurden, nicht versendet.

Die Kommunikation zwischen einer Replikationskontrollkomponente und der Replikationsschicht kann dabei sehr unterschiedlich ausgestaltet sein. Die einfachste Form ist der Einsatz von Rückruffunktionen (engl. callback functions) beziehungsweise einer Beobachterschnittstellen bei Objektorientierung. Dabei läuft sowohl die Kontrollkomponente, als auch die Replikationsschicht, wie in Abbildung 4.21(a) zu sehen, im gleichen Prozess ab. Die direkte Kopplung verringert den Kommunikationsaufwand zwischen den Komponenten, wodurch die Leistungsfähigkeit positiv beeinflusst wird, birgt aber auch verstärkt die Gefahr von Verklemmungen, die durch zyklische Wartesituationen auftreten können. Das andere Extrem ist in Abbildung 4.21(b) dargestellt, bei dem die Kontrollkomponente in einen separaten Prozess oder sogar Knoten ausgelagert ist. Die starke Entkopplung sorgt für ein hohes



(a) Direkt (b) Entkoppelt (c) Hybrid

Abbildung 4.21: Kommunikation zwischen Ereignisbehandlung und Replikationsschicht

Maß an Flexibilität, da beispielsweise eine Kontrollinstanz für mehrere Replikationsknoten zuständig sein kann. Jedoch sinkt bei einer hohen Ereignisfrequenz die Leistungsfähigkeit der Replikationsschicht, da die Kommunikation zwischen Kontrollkomponente und Replikationsschicht zusätzliche Latenz verursacht. Eine Kombination der beiden Ansätze zu einem hybriden Modell ist in 4.21(c) dargestellt. Die Kontrollkomponente besteht nun aus zwei disjunkten Teilen: einer lokal im Prozess der Replikationsschicht, der für Ereignisse mit hoher Frequenz zuständig ist und ein entfernter oder entkoppelter Teil, der Ereignisse mit niedriger Latenz verarbeitet, beziehungsweise den lokalen Teil konfiguriert.

Zusätzlich zur Behandlung von Ereignissen kann eine Kontrollkomponente auch Zugriff auf die interne Verwaltung der Daten und insbesondere Einfluss auf die Replikation nehmen. Beispielsweise kann die Kommunikation zweier SuperPeers via HyperPeer eingeschränkt werden, indem Aktualisierungen für Replikate nicht an alle SuperPeers verteilt werden oder Replikate exklusiv nur durch einen SuperPeer verwaltet werden dürfen.

4.5.2.3 Dienste der verteilten Welt

Abbildung 4.22 zeigt den konzeptionellen Aufbau der gesamten Infrastruktur. Dabei wird zwischen den Klienten der Nutzer und den Hintergrunddiensten unterschieden. Ein Nutzerdienst ist gleichzusetzen mit einem Teilnehmer an der virtuellen Welt und wird immer auf dem Rechner des Teilnehmers ausgeführt. Der Dienst kann sich immer nur mit genau einem Hintergrunddienst verbinden. Die Hintergrunddienste sind analog zur klassischen Definition aus Abschnitt 2.3.2 angelegt und stellen alle Funktionalitäten bereit, die für den persistenten Betrieb der Welt nötig sind. Jeder Dienst kann dabei einem dedizierten Rechner zugeordnet werden oder gemeinsam mit

anderen auf einem ausgeführt werden.

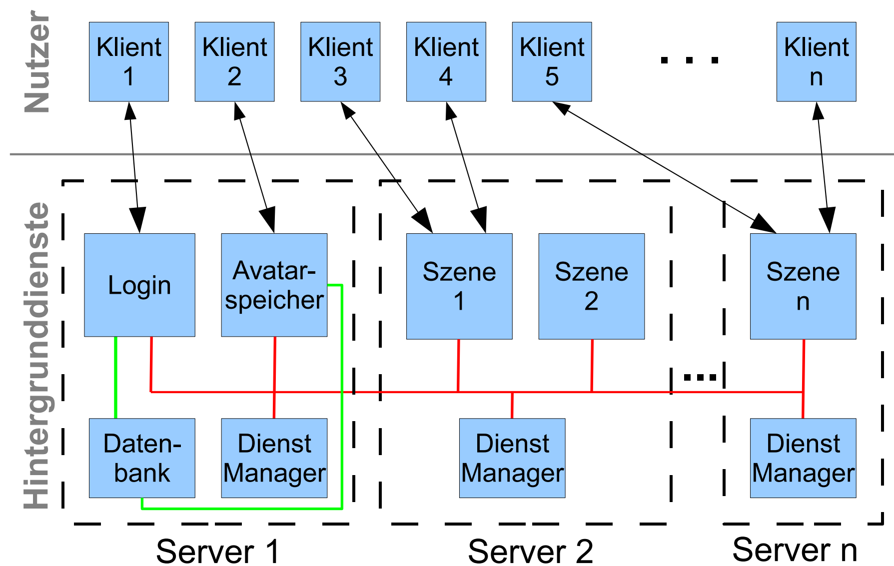


Abbildung 4.22: Dienste der verteilten Welt

DB

Der Datenbankdienst dient den Diensten als Basis für eine persistente Speicherung von Informationen, die auch nach einem totalen Ausfall der Hintergrunddienste erhalten bleiben. Neben den klassischen SQL-basierten Datenbanken können auch NoSQL-Varianten [Lea10] zum Einsatz kommen.

Login

Der Login-Dienst übernimmt die Authentifizierung und Autorisierung der Benutzer. Die Authentifizierung wird verwendet, um einem Teilnehmer seinen persistenten Avatar zuzuordnen und um sicherzustellen, dass ein Nutzer nur einmal in der Welt vorhanden ist. Die Authentifizierungsinformationen werden üblicherweise durch den Datenbankdienst bereitgestellt, jedoch können auch andere Varianten genutzt werden. Beispielsweise nutzt Wissenheim Worlds für Mitglieder der Universität Ulm einen LDAP-Server des Rechenzentrums, um Nutzer zu authentifizieren.

Der Logindienst stellt zusätzlich ein Ticketsystem bereit, mit dessen Hilfe die anderen Dienste Nutzer authentifizieren und eine Zugriffskontrolle durchführen können.

Avatarspeicher

Der als *Avatarspeicher* bezeichnete Dienst dient als Speicher- und Verwaltungsort für den virtuellen Avatar des Nutzers. Der Avatar eines Nutzers wird dabei in Form eines Avatargraphen gespeichert, der im Wesentlichen aus dem Avatarobjekt und seinen Kindknoten besteht. Dienste, welche die Struktur eines Avatars benötigen, können den Graphen vom Avatarspeicher laden.

Szenen

Die Szenendienste entsprechen den in Abschnitt 2.3.2 vorgestellten Zonenservern und übernehmen die Simulation einer Szene. Dies entspricht einer statischen Area-of-Interest auf logischer Ebene, wie sie in Abschnitt 4.5.1.1 beschrieben wird. Des Weiteren sorgt der Dienst dafür, dass Avatarzombies von Benutzern, die unerwartet die Szene verlassen und ihren Avatar nicht selber entfernen konnten, aus dem Szenengraph gelöscht werden.

Dienstmanager

Wie in der Abbildung angedeutet, sollen die Dienste auf mehrere Knoten verteilt werden können, um auf Hardwareebene eine Lastverteilung zu ermöglichen. Der Dienstmanager dient der Verwaltung der übrigen Dienste (mit Ausnahme der Datenbank). Wird ein neuer Rechner den Hintergrunddiensten zugeordnet, so wird dort zuerst der Dienstmanager gestartet. Alle weiteren Dienste werden dann automatisch durch diesen Dienst erzeugt, beziehungsweise administriert.

Die Kommunikation zwischen den Hintergrunddiensten erfolgt dabei via TGOS, im Diagramm durch die roten Verbindungslinien dargestellt. Auch die Nutzerdienste kommunizieren via TGOS, jedoch ist die Verbindung anders gestaltet, als zwischen den Hintergrunddiensten. Mehr Informationen hierzu finden sich im nächsten Abschnitt, der sich mit der Kopplung zwischen Replikationsschicht und Dienstestruktur beschäftigt. Der Datenbankdienst bildet bei der Kommunikation eine Ausnahme, da dieser nicht via TGOS angesprochen wird, sondern eine SQL- oder NoSQL-Verbindung, grün dargestellt, verwendet.

4.5.2.4 Verknüpfung von Diensten und Replikation

Die im vorherigen Abschnitt vorgestellten Dienste sollen nun mit der in Abschnitt 4.5.2.2 definierten Replikationsschicht verknüpft werden. Das Ergebnis ist in Abbildung 4.23 zu sehen. Das Bild orientiert sich in seiner Struktur an der Beschreibung der Dienste und nutzt die selben Symbole, die auch bei der Definition der Replikationsschicht Verwendung fanden.

Wie im Diagramm zu sehen, gliedert sich ein Hintergrunddienst in einen SuperPeer und einen Peer, durch den die Logik des Dienstes via TGOS mit

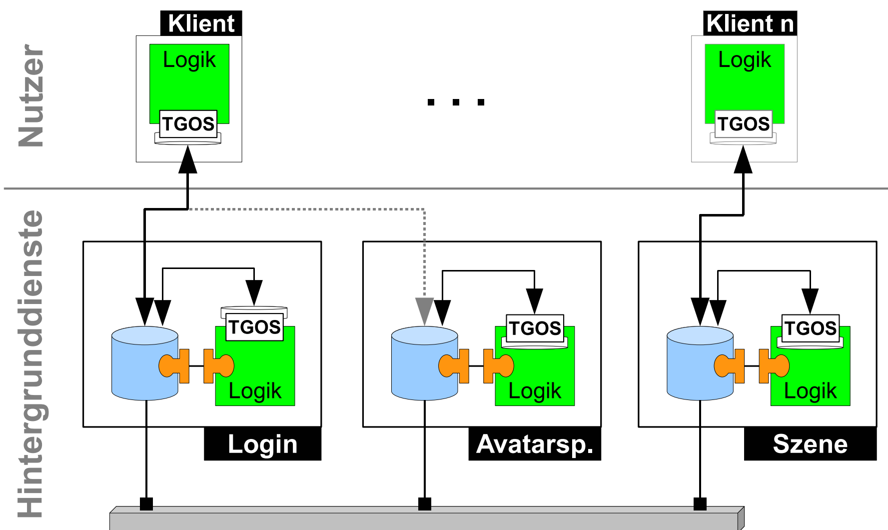


Abbildung 4.23: Wissenheim Worlds Replikationsschicht

anderen Peers kommunizieren kann. Zusätzlich nutzt die Logik die Replikationskontrollschnittstelle des SuperPeers, um diesen zu steuern und beispielsweise den Zugang zur virtuellen Welt oder das unbefugte Ändern oder Lesen von Daten zu unterbinden. Für die Kopplung von SuperPeer und Zugriffskontrolle können alle in Absatz 4.5.2.2 beschriebenen Varianten zum Einsatz kommen. Die Kommunikation der Hintergrunddienste, beziehungsweise ihrer Logikkomponenten untereinander, erfolgt ebenfalls über TGOS. Die SuperPeers bedienen sich des HyperPeers, um Replikate anderer Dienste zu lesen oder diese zu aktualisieren. Die Peers der Klienten verbinden sich mit den SuperPeers und ermöglichen so ebenfalls eine Kommunikation via TGOS-Operationen.

Die Logik der Dienste kann, mit Hilfe der Kontrollkomponente, die *Peers* der Klienten veranlassen, sich an den *SuperPeer* eines anderen Dienstes zu verbinden. Diese Verbindungsänderung geschieht dabei aus Sicht von TGOS transparent und dient hauptsächlich der Lastverteilung. Die Authentifizierung der Klienten kann dabei durch das Ticketsystem des Logindienstes erfolgen.

Da jeder Dienst über einen eigenen *SuperPeer* verfügt, bietet es sich an, die Daten der verteilten Welt unter den Diensten aufzuteilen. Die Logindaten sind beispielsweise nur im *SuperPeer* des Logindienstes gespeichert, die Daten der Avatare nur im Avatarspeicher. Benötigt ein Klient Daten, die in einem anderen Dienst residieren, so müssen die SuperPeers diese via HyperPeer beziehen. Das Konzept der strikten Trennung der Daten ist insbesondere für Szenendienste wichtig, da Aktualisierungsnachrichten dadurch auf einen SuperPeer beschränkt bleiben und die Kommunikation zwischen

den SuperPeers minimiert wird.

4.6 Ein nachrichtenbasierter Ansatz im Vergleich

Inwieweit sich die Konzeption einer verteilten Welt und insbesondere die Umsetzung der Inhalte mit TGOS von einem klassischen nachrichtenbasierten Ansatz unterscheidet, soll am Beispiel des RedDwarf-Frameworks untersucht werden. Der Vergleich nimmt dabei Bezug auf das in Abschnitt 4.4 vorgestellte Volleyballspiel, welches mit TGOS umgesetzt wurde.

4.6.1 Das RedDwarf-Framework

RedDwarf ist ein Open-Source-Framework für verteilte virtuelle Welten, welches aus dem Darkstar Projekt [Wal08], einem Sun Microsystems Laborprojekt, entwickelt wurde. Das Hauptziel des RedDwarf-Projektes ist es, einem Programmierer eine skalierbare, einfach zu nutzende Infrastruktur für verteilte virtuellen Welten oder Spiele zu bieten. Das Framework stützt sich dabei auf die Java-Laufzeitumgebung von Oracle® ab. Abbildung 4.24 skizziert den konzeptionellen Ansatz des Frameworks.

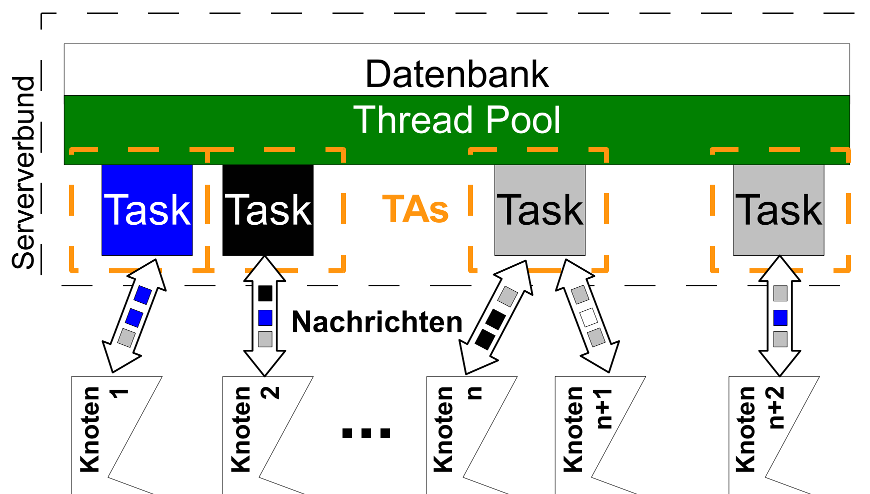


Abbildung 4.24: Konzeptioneller Aufbau des RedDwarf-Frameworks

Das Framework unterscheidet zwischen Klienten, den Knoten der Spielteilnehmer, und dem Serververbund, der die Spielwelt verwaltet. Die Klienten sind mit einer einfachen, nachrichtenbasierten Schnittstelle im Client/ Server-Stil angebunden.

Das zentrale Element des Serververbundes stellt die Datenbank dar, die für eine persistente und konsistente Speicherung der persistenten Da-

ten sorgt. Jede Nachricht eines Klienten erzeugt einen Task, welcher die Klientennachricht interpretiert und die Logik der verteilten Welt ausführt. Nimmt ein Task Änderungen an persistenten Daten vor, so erfolgen diese mit Unterstützung der Datenbank automatisch transaktional. Da Transaktionen möglichst kurz sein sollten, damit langlaufende Tasks kürzere nicht unnötig blockieren, sollte auch die Laufzeit der Task entsprechend kurz sein. Die Transaktionalität bezieht sich dabei nur auf die im Serververbund ausgeführten Transaktionen, die, im Gegensatz zu den TGOS-Transaktionen, nicht auf den Klienten ausgeführt werden. Jedem Task wird für die Ausführung ein Thread aus einem Thread-Pool zugeordnet.

Für die Kommunikation zwischen den Klienten untereinander und mit dem Serververbund nutzt RedDwarf ein Kanal-Konzept mit einer einfachen, nachrichtenbasierten Schnittstelle. Im Codebeispiel 4.8 ist die Definition des *ClientChannel* dargestellt. Der Kanal bietet über die *send*-Routine die

Listing 4.8: RedDwarf ClientChannel

```
 1 public interface ClientChannel {
 2   public String getName();
 3   public void send(ByteBuffer msg);
 4 }
 5
 6 public interface ClientChannelListener {
 7   public leftChannel(ClientChannel channel);
 8   public void receivedMsg(ClientChannel channel,
 9                           ByteBuffer message)
10 }
```

Möglichkeit, eine Nachricht an alle mit dem Kanal verbundenen Knoten zu schicken. Mit Hilfe des *Listeners* erfolgt die Benachrichtigung des Klienten im Falle einer ankommenden Nachricht. Analog existiert noch ein Serverteil des Kanals, der neben den spiegelbildlichen Nachrichtenfunktionen noch Operationen und Ereignisse zur Verwaltung von Klientenverbindungen enthält.

4.6.2 Volleyball mit RedDwarf

Der wesentliche Unterschied zwischen dem TGOS-Ansatz und einer Umsetzung mit RedDwarf liegt in der Auftrennung der Logik und der Datenstrukturen in einen Klienten- und Serverteil. Für beide Teile müssen separate Zustandsautomaten und passende Nachrichtenbehandlungsroutinen definiert werden. Zusätzlich müssen bei der Entwicklung stets beide Teile aktualisiert und synchronisiert werden.

Ein weiteres Problem ergibt sich aus der Notwendigkeit, dass jedem neu

beitretenden Knoten initial ein konsistentes Abbild der Szene zur Verfügung gestellt werden muss. Üblicherweise (vgl. [Lin99]) fertigt der Server hierzu ein konsistentes Abbild der Szene, ähnlich einem Checkpoint, an, serialisiert dieses in ein geeignetes Format und überträgt es an den Klienten. Dieser deserialisiert die Daten und erzeugt daraus lokal die Szene. Währenddessen müssen alle Aktualisierungen des Serverzustandes gepuffert werden, bis der Klient diese verarbeiten kann. Die benötigten Serialisierungsfunktionen müssen dabei, je nach Framework, vom Programmier selbst erstellt werden. Dabei geht es in erster Linie nicht um die eigentliche Umwandlung in eine binäre Form, vielmehr muss eine Verknüpfung zwischen den Objekten auf Klient- und Serverseite stattfinden. Beispielsweise kann ein Avatar im Klienten durch eine andere Struktur dargestellt werden als im Serverteil, dennoch muss eine globale Adressierbarkeit gegeben sein.

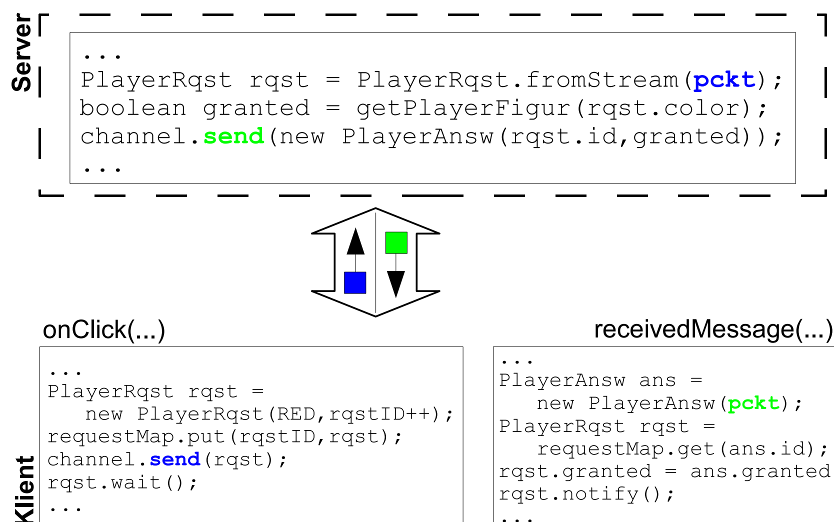


Abbildung 4.25: Asynchrone Auswahl der Spielfigur

Ein weiterer Punkt ist die asynchrone Ausführung, welche durch die Nachrichtenbasierung notwendig wird. Die Auswirkungen sind in Abbildung 4.25 dargestellt, welche in Pseudocode die wesentlichen Schritte zeigt, die für die Auswahl einer Spielfigur nötig sind. Der Vorgang startet in der *onClick*-Routine des Klienten, die beim Anklicken einer Spielfigur aufgerufen wird. Diese Routine erzeugt ein neues Anfragepaket (engl. request), welches beim Server eine Spielfigur reservieren soll. Das Paket enthält neben der Farbe der Spielfigur auch einen Anfrageidentifikator, mit dem bei einer Antwort des Servers die passende Anfrage gefunden werden kann. Zusätzlich wird die Anfrage in einer HashMap vermerkt und nach dem Senden wartet die Routine auf das Ergebnis. Erreicht das Paket den Server, so erzeugt

dieser aus den binären Daten erst ein Objekt und prüft dann, ob er die Spielfigur erfolgreich reservieren kann. Dann sendet er eine Antwort, welche den Anfrageidentifikator und das Ergebnis enthält, zurück. Die *receive-Message*-Routine des Klienten empfängt das Paket, sucht anhand des darin gespeicherten Anfrageidentifikators die richtige Anfrage und benachrichtigt die anfragende Routine.

Im Gegensatz zu TGOS werden nun drei Programmteile benötigt, um die Reservierung durchzuführen. Zusätzlich müssen Verwaltungsinformationen, in diesem Fall der Anfrageidentifikator und die HashMap, bereitgestellt werden, die nichts mit der eigentlichen Logik zu tun haben.

4.6.3 Bewertung

Eine Implementierung, basierend auf nachrichtenorientierten Mechanismen, gestaltet sich deutlich aufwendiger, als die entsprechende Umsetzung mit TGOS. Insbesondere die Trennung in Klienten- und Serverteil verdoppelt im ungünstigsten Fall den Implementierungsaufwand und erzeugt zusätzliche Komplexität und Fehlerquellen. Auch muss der Entwickler von Hand für eine globale Adressierbarkeit wichtiger Objekte der virtuellen Welt sorgen.

Obwohl im Beispiel ein Client/Server-Ansatz verwendet wird, lassen sich die meisten Problemstellungen auch auf eine Peer-to-Peer-Lösung übertragen. Statt einem Klienten und einem Server gibt es dort einen Empfänger und einen Senderteil und zusätzlich muss für viele Entscheidungen ein kompliziertes Abstimmungsverfahren verwendet werden.

4.7 Verwandte Arbeiten

Neben dem RedDwarf-Framework gibt es noch eine große Anzahl weiterer Arbeiten, die sich mit der Verteilung von virtuellen Welten als Ganzes oder mit bestimmten Aspekten befassen.

Das Projekt 027-in-Space der Universität Ulm und sein Nachfolger Wissenheim stellen beispielsweise einen der ersten Versuche dar, eine verteilte virtuelle Welt mit Hilfe eines rein datenzentrierten Ansatzes zu realisieren. Im Mittelpunkt stand dabei die Nutzung eines transaktionalen, verteilten und stark getypten Speichers, der durch das proprietäre Betriebssystem Plurix [STS98] bereitgestellt wurde. Im Gegensatz zu TGOS bot Plurix als Konsistenzmodell ausschließlich die transaktionale Konsistenz, wodurch die Skalierbarkeit und Effizienz eingeschränkt war. Ein Beschreibung der verteilten Welt und des transaktionalen Ansatzes findet sich in [FFSS06].

Im Rahmen des XtreemOS-Projektes, einem EU-geförderten Forschungsprojekt (FP6-033576) mit dem Schwerpunkt *Grid Computing*, erfolgte eine Portierung des auf Plurix basierenden Wissenheims nach Linux, unter Nutzung des Object Sharing Service (kurz OSS) der Universität Düsseldorf. OSS stellt einen verteilten, transaktionalen Speicher bereit, der sich nahtlos in das

Programmiermodell von Linux einfügt und im Gegensatz zu Plurix explizite Transaktionen anbietet. Jedoch bietet OSS im Gegensatz zu TGOS nicht die Möglichkeit, eigene Konsistenzmodelle zu implementieren und sowohl die Verteilung als auch die Granularität erfolgt für den Nutzer transparent.

Das Peers-at-Play-Projekt der Universitäten Mannheim, Duisburg-Essen und Hannover ist ein weiteres Forschungsprojekt, welches sich mit Verteilungstechniken virtueller Welten befasst. Der Schwerpunkt des Projektes liegt, im Gegensatz zu TGOS, jedoch auf der Peer-to-Peer-Kommunikation zwischen den Teilnehmern und den daraus resultierenden Herausforderungen. Im Rahmen des Projektes wurden mehrere Ansätze zur Konsistenzverwaltung vorgestellt, darunter ein Konzept [ITSB10], das Konsistenzdomänen auf Basis des Area-of-Interest-Bereichs eines Avatars erstellt und versucht, automatisch ein geeignetes Konsistenzmodell anhand der Interaktion der Avatare zu bestimmen. Des Weiteren existieren auch Arbeiten [KWSW08], die sich explizit mit dem Verbinden eines neuen Knotens zum Peer-to-Peer-Verbund und den damit verbundenen Herausforderungen befassen. Im Gegensatz zu TGOS liegen noch keine oder nur sehr wenige praktische Erfahrungen mit den beschriebenen Konzepten vor und das Hauptaugenmerk liegt auf dem Peer-to-Peer-Ansatz.

Eine weitere interessante Arbeit stellt *Croquet* [SKRR03] dar, welches analog zu Peers-at-Play eine Peer-to-Peer-basierte Architektur für eine verteilte Welt definiert. Die Besonderheit von Croquet ist der *TeaTime* genannte Ansatz zur Verteilung und Synchronisierung von Ereignissen. Ereignisse werden in Croquet mit einem globalen Zeitstempel versehen und können von jedem Objekt in der Welt empfangen und gesendet werden.

4.8 Zusammenfassung

Die Konzeption einer verteilten virtuellen Welt, aufbauend auf dem TGOS-Programmiermodell, rückte den Fokus von asynchronen Nachrichten auf den Zugriff verteilter Datenstrukturen. Wie an der Beispielszene in Abschnitt 4.4 zu sehen, gestaltet sich die Umsetzung des Volleyballspiels beinahe wie für ein unverteiltes Spiel. Lediglich an wenigen Punkten fand eine Synchronisierung mit Hilfe der TGOS-Basisoperationen statt oder wurde auf die transaktionale Konsistenz, in Form einer Komponente, zurückgegriffen.

Durch die im TGOS-Modell definierten Basisoperationen und Ereignisse lassen sich neue Konsistenzmodelle, wie in Abschnitt 4.3 beschrieben, sehr einfach netzarchitekturunabhängig und modular umsetzen. Auch komplexere Konsistenzmodelle lassen sich mit wenig Aufwand realisieren, wie am Beispiel der transaktionalen Konsistenz (siehe Abschnitt 4.3.2) gezeigt wurde. Dabei zeigte sich auch, dass sich die vorgestellte spezielle Ausprägung der transaktionalen Konsistenz sehr gut zur Sicherung kritischer Strukturen in verteilten Welten eignet, falls diese selten geschrieben, aber häufig

gelesen werden. Eine Verwendung im allgemeinen Fall, wie beispielsweise in [FFSS06], ist in virtuellen Welten nicht als sinnvoll zu erachten.

Auch die Umsetzung der für verteilte Welten so wichtigen Lastverteilungs- und Partitionierungsmechanismen ist innerhalb des Programmiermodells einfach möglich. Die in Absatz 4.5.1 vorgestellten Ansätze sind dabei unabhängig von der verwendeten Netzarchitektur und Replikationsschicht. Auch ist es leicht möglich, dynamisch zur Laufzeit zwischen verschiedenen Ansätzen zu wechseln, um so beispielsweise auf Änderungen im Nutzerverhalten zu reagieren.

Der Vergleich mit einem klassischen, nachrichtenorientierten Ansatz in Abschnitt 4.6 demonstriert, dass die Umsetzung mit Hilfe eines datenzentrierten Ansatzes sich deutlich einfacher gestaltet, da sowohl die asynchrone Programmierung als auch die Aufteilung der Logik in einen Client- und einen Serverteil entfallen kann.

Die verwandten Arbeiten im Bereich der Verteilung virtueller Welten beschäftigen sich zu großen Teilen mit der Betrachtung Peer-to-Peer-fähiger Algorithmen oder anderen nachrichtenorientierten Mechanismen. Der explizite, datenzentrierte und ereignisorientierte Ansatz, der im TGOS-Modell definiert wird, stellt ein neues Paradigma für die Verteilung virtueller Welten bereit. Auch das Konzept, Konsistenzmodelle innerhalb des Modells unter Verwendung der Basisoperationen umzusetzen, stellt eine Neuheit im Umfeld verteilter virtueller Welten dar.

Kapitel 5

Messungen

Die in Kapitel 4 vorgestellten Konzepte und Ansätze wurden in der virtuellen Welt *Wissenheim Worlds* umgesetzt und sowohl im Rahmen von Messungen, als auch im realen Einsatz evaluiert.

Die Implementierung von Wissenheim Worlds basiert auf einer Java-Laufzeitumgebung der Firma Oracle® , Version 1.6, die ohne Modifikationen verwendet wird. Der Zugang zur virtuellen Welt erfolgt über ein Java-Applet, welches in einem beliebigen, java-fähigen Webbrowser gestartet werden kann. Es ist keine Installation nötig und es werden keine Root-Rechte für die Ausführung benötigt. Für die Visualisierung und Akustik greift der Prototyp auf das LWJGL-Framework [LWJ10] zurück, welches für eine Anbindung an die hardwarebeschleunigte OpenGL- beziehungsweise OpenAL-Schnittstelle des Gastsystems sorgt. Im Rahmen von Wissenheim Worlds fand eine Implementierung der im TGOS-Modell definierten Funktionalität statt, die im Folgenden als *wwShare* bezeichnet wird. Zusätzlich wurden zwei TGOS-kompatible Replikationsschichten realisiert, die jeweils unterschiedliche Netzarchitekturen verwenden. Die Struktur von Wissenheim Worlds ist im Wesentlichen mit der in Abschnitt 4.5.2 vorgestellte und in Abbildung 4.22 dargestellten Architektur identisch.

Die Messungen gliedern sich in zwei Teile; die Evaluation der Basiskomponenten und die Evaluation einer verteilten virtuellen Welt.

5.1 Evaluation der Basiskomponenten

Die Evaluation der Basiskomponenten einer verteilten virtuellen Welt soll einen Einblick in die Leistungsfähigkeit der im TGOS-Modell definierten und in wwShare realisierten Funktionen geben. Auch sollen die Leistungsfähigkeit und die Kosten einer Konsistenzmodell-Implementierung mit wwShare anhand des transaktionalen Konsistenzmodells untersucht werden. Da die Replikationsschicht im TGOS-Modell austauschbar ist, müssen die Meßdaten immer in Relation zur momentan verwendeten gesehen werden. Aus diesem

Grund werden die beiden derzeit implementierten prototypischen Replikationsschichten in die Messungen miteinbezogen.

Für die Evaluation der Basiskomponenten wurde ein Cluster mit 16 Knoten verwendet, die mit Gigabit-Ethernet vernetzt und über eine Gigabit-Switch miteinander verbunden sind. Jeder Knoten dieses *Opteron-Clusters* besteht aus zwei AMD® Opteron-Prozessoren (Typ 244, 1.8GHz), denen zwei 2GB Arbeitsspeicher zur Verfügung stehen.

5.1.1 Serialisierungsaufwand

Jede Netzwerkkommunikation erfordert eine Serialisierung der Daten in ein Paketformat. Von Interesse für die Messung sind deshalb nicht die Serialisierungskosten als solche, sondern der Aufwand, der durch das automatische Serialisieren entsteht. Dabei ist sowohl die für die Serialisierung benötigte Zeit, als auch die Größe der serialisierten Daten von Interesse. Als Vergleichsbasis dient, neben den von Java standardmäßig bereitgestellten Serialisierungsmethoden, auch die manuelle Serialisierung. Bei dieser werden die in den Objekten enthaltenen Daten manuell durch den Programmierer extrahiert, beziehungsweise importiert, wie es beispielsweise bei der Nutzung einer Socket-basierten Kommunikation nötig wäre.

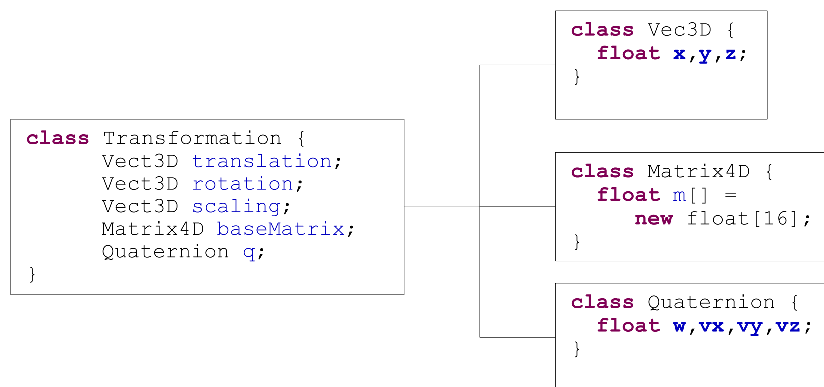
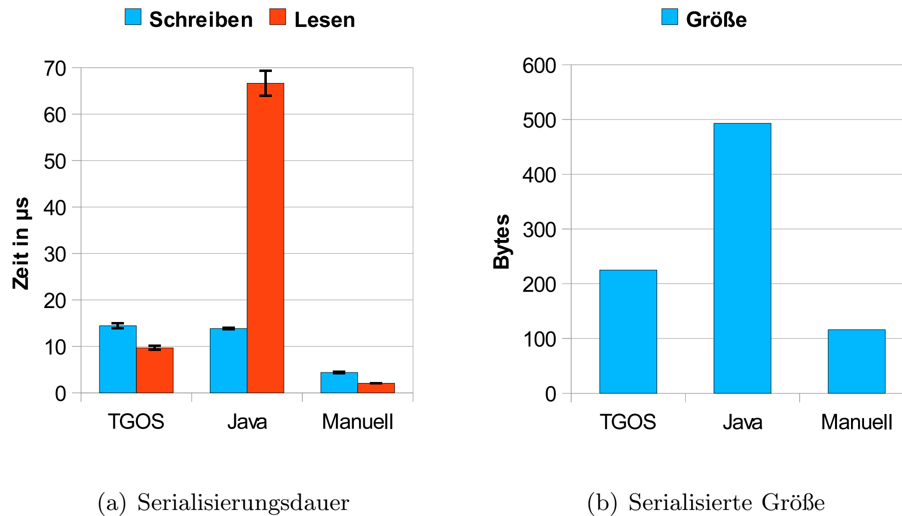


Abbildung 5.1: Das Transformationsobjekt und seine Komponenten

Als Basisobjekt für die Messungen wird das in Wissenheim Worlds gebräuchliche Transformationsobjekt eines Formobjektes benutzt. Dieses Objekt ist ein Verbund aus verschiedenen Objekttypen, welcher in Abbildung 5.1 detailiert dargestellt ist. Durch den Hüllenbildungsmechanismus des TGOS-Modells (siehe Abschnitt 3.2.4) wird immer der ganze Verbund serialisiert.

Abbildung 5.2(a) zeigt die für die Serialisierung benötigte Zeit, aufgeschlüsselt nach der verwendeten Methode. Abbildung 5.2(b) gibt Aufschluss

über die Größe der Daten in serialisierter Form. Für die Messungen wurde das Objekt 1000-mal serialisiert und die benötigte Zeit gemessen. Dies wurde 1000-mal wiederholt und der Mittelwert über die Daten gebildet; zusätzlich wurden die Messungen auf sechs Cluster-Maschinen parallel durchgeführt. Die Standardabweichung zwischen den Messreihen ist im Diagramm 5.2(a) als schwarzer Fehlerbalken dargestellt. Die Messwerte zeigen das Minimum an Serialisierungszeit, die sowohl positive Caching-Effekte, als auch JIT-Compiler Optimierungen enthält.



(a) Serialisierungsdauer

(b) Serialisierte Größe

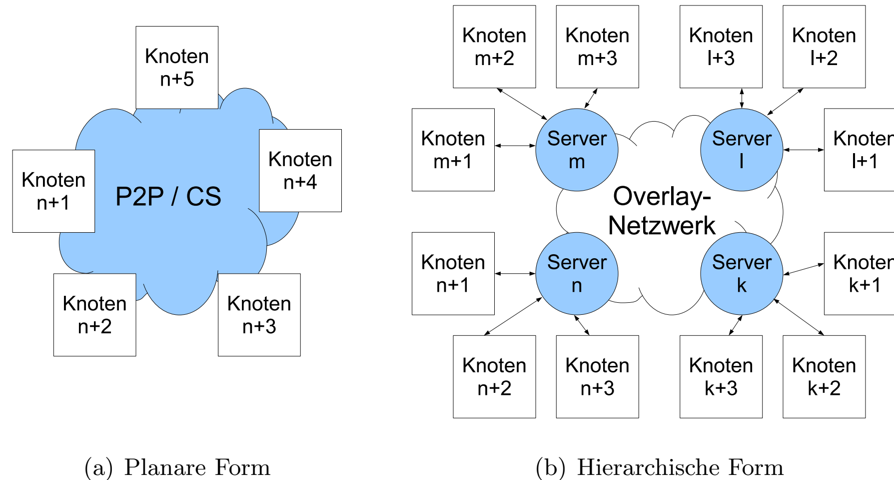
Abbildung 5.2: Marshalling des Transformationsobjekt

Obwohl die automatische Serialisierung von wwShare im Vergleich zu Java, insbesondere bei der Deserialisierung, deutlich schneller ist, benötigt sie im Vergleich zur manuellen Variante dennoch deutlich mehr Zeit. Sollten viele Serialisierungen pro Sekunde nötig sein, kann die automatische Variante zum Flaschenhals werden. Um diesem vorzubeugen, kann ein hybrider Ansatz gewählt werden, bei dem häufig aktualisierte Objekte über einen manuell programmierten Mechanismus serialisiert werden. Im Falle von Wissenheim Worlds bietet sich beispielsweise das hier vorgestellte Transformationsobjekt an, da es das mit Abstand am häufigsten serialisierte Objekt ist.

5.1.2 Replikationsschichten

Wissenheim Worlds stellt zwei TGOS-kompatible Replikationsschichten zur Verfügung: ein auf Client/Server-Architektur basierender Prototyp (*CS - Replikation*), welcher TCP-Verbindungen nutzt und eine auf P2P-Technik basierende Variante, welche auf dem JGroups-Gruppenkommunikationsframe-

work mit UDP-Protokoll aufsetzt (*P2P-Replikation*). Da die Replikationsschichten einen wesentlichen Einfluss auf die Leistungsfähigkeit des Gesamtsystems haben, sollen im Folgenden deren Leistungscharakteristiken beleuchtet werden. Ziel ist dabei nicht, die Güte einer bestimmten Replikationsmethode oder Implementierung zu bewerten, sondern eine Datenbasis zu schaffen, mit der die Kombination aus TGOS-Implementierung und Replikationsschicht bewertet werden kann.



(a) Planare Form

(b) Hierarchische Form

Abbildung 5.3: Replikationsarchitekturformen

Die Experimente werden zusätzlich durch zwei verschiedene Replikationsarchitekturen differenziert: die planare und die hierarchische Form. Bei der ersten Variante, zu sehen in Abbildung 5.3(a), kommunizieren alle teilnehmenden Knoten direkt miteinander; bei Client/Server über maximal einen Server als Verteiler, im Falle von P2P über Verbindungen der Knoten untereinander. Bei der hierarchischen Architekturform, dargestellt in Abbildung 5.3(b), findet eine Partitionierung des Replikationsraumes statt, bei der die einzelnen Zellen über ein Overlay-Netzwerk miteinander verbunden sind. Ein Overlay-Netzwerk [LCP+05] bezeichnet dabei ein logisches Netzwerk, das die einzelnen Zellen verbindet, auf einer darunterliegenden Netzinfrastruktur aufsetzt und dessen Struktur und Adressierung abstrahiert. Ziel des hierarchischen Ansatzes ist es, die Menge an Knoten so zu partitionieren, dass so wenig Datenverkehr wie möglich über das Overlay-Netzwerk abgewickelt werden muss.

5.1.2.1 Planare Replikation

Als erste Messgröße wird der maximale Durchsatz bei Schreiboperationen gemessen. Dabei ist zu beachten, dass jede Aktualisierung eines Replikats

durch einen Knoten eine Aktualisierungsnachricht an alle anderen Teilnehmer nach sich zieht. Für die Messungen werden acht Opteron-Knoten verwendet, mit einer Datengröße des Replikats von 512 Byte. Jeder Knoten aktualisiert jeweils ein eigenes Replikat. Es wurden vier Messreihen erstellt und deren Abweichungen als schwarze Fehlerbalken im Diagramm dargestellt. Gemessen wurde die Empfangsrate auf den teilnehmenden Knoten, welche theoretisch linear mit der Senderate skalieren sollte, bis eine Sättigung eintritt. Die Senderate gibt dabei an, wieviele Aktualisierungen pro Sekunde maximal durchgeführt werden können. Da jeder Knoten nur einen Thread für das Ausführen der Aktualisierungen verwendet, kann die Senderate auch deutlich unter der maximal erlaubten Rate liegen. Dies ist abhängig von der Sendeleistung der Replikationsschicht, deren Pufferungsmöglichkeiten und/oder Flusskontrollen.

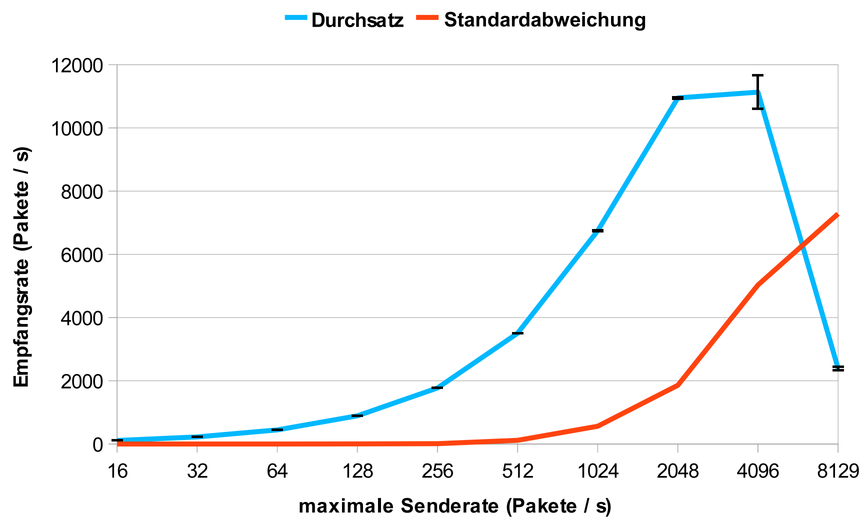


Abbildung 5.4: Maximaler Durchsatz mit P2P-Replikation

In Abbildung 5.5 sind die Ergebnisse für die Client/Server-Variante dargestellt; in Abbildung 5.4 die für die P2P-basierte. Auf der X-Achse ist die maximal mögliche Senderate dargestellt, die der Knoten zu erreichen versucht. Neben der Empfangsrate ist auch die Standardabweichung dargestellt. Die Client/Server-basierte Variante ist dabei der P2P-basierten, sowohl im maximalen Durchsatz als auch bei der Varianz, deutlich unterlegen und sättigt früher.

Neben dem Lastverhalten ist insbesondere die Reaktionszeit oder Latenz von besonderem Interesse. Um das Verhalten zu messen, wurde ein Multi-Ping-Ansatz implementiert, bei dem jeder Knoten die Latenz zu allen anderen Knoten mit Hilfe von Ping-Anfragen misst. Dabei ist zu beachten, dass

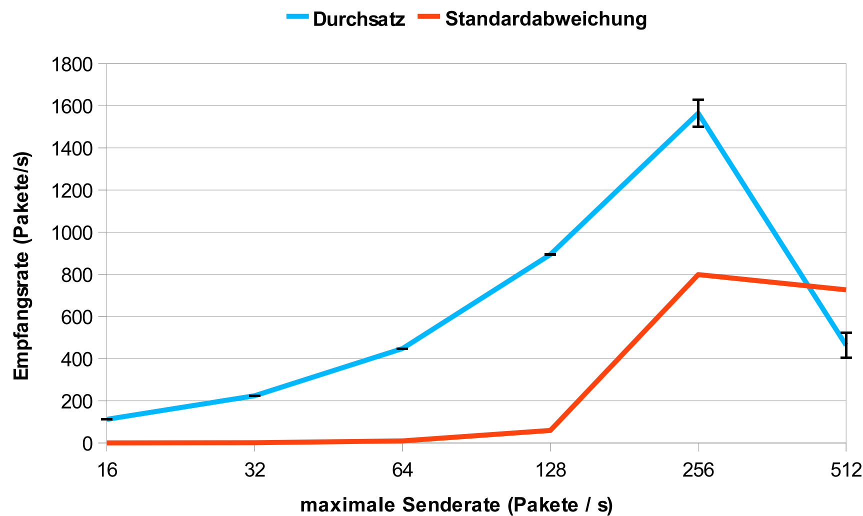


Abbildung 5.5: Maximaler Durchsatz mit CS-Replikation

wiederum auf die Aktualisierungsfunktion zurückgegriffen wurde, wodurch ein Knoten, welcher eine Ping-Anfrage in Form einer Replikataktualisierung erhält, diese beantwortet, indem er das gerade aktualisierte Replikat selbst noch einmal aktualisiert. Dies hat zur Folge, dass die Antwort nicht nur den anfragenden Knoten erreicht, sondern alle teilnehmenden Knoten. Diese ignorieren jedoch diese Replikatsaktualisierung. Durch die im TGOS-Modell für eine Replikationsschicht definierten Gruppenmechanismen, könnte auch eine Punkt-zu-Punkt-Kommunikation zwischen den Knoten realisiert werden. Da aber gerade auch das Latenzverhalten unter Last untersucht werden sollte, wurde auf diese Möglichkeit verzichtet.

Abbildung 5.7 zeigt die Ergebnisse der Messung für den Client/Server-Fall und Abbildung 5.6 jene für die P2P-basierte Variante. Das Latenzverhalten bei P2P-Replikation ist sehr gut und nimmt anfänglich mit steigender Datenrate noch etwas zu, was auf ein verbessertes Timing durch den konstanteren Datenverkehr zurückzuführen ist. Bei der CS-Replikation liegen die Latenzen deutlich über denen der P2P-Variante, auch steigen sie schneller an.

Die Messungen zeigen, wie unterschiedlich die Leistungsfähigkeit der beiden Replikationsschichten ist und wie wichtig die im TGOS-Programmiermodell definierte Austauschbarkeit ist. Die P2P-Variante ist dabei der CS-Replikation deutlich überlegen, was insbesondere auf das optimierte JGroups-Framework zurückzuführen ist und keine Rückschlüsse auf die Leistungsfähigkeit zwischen Client/Server- und P2P-Ansätzen zulässt.

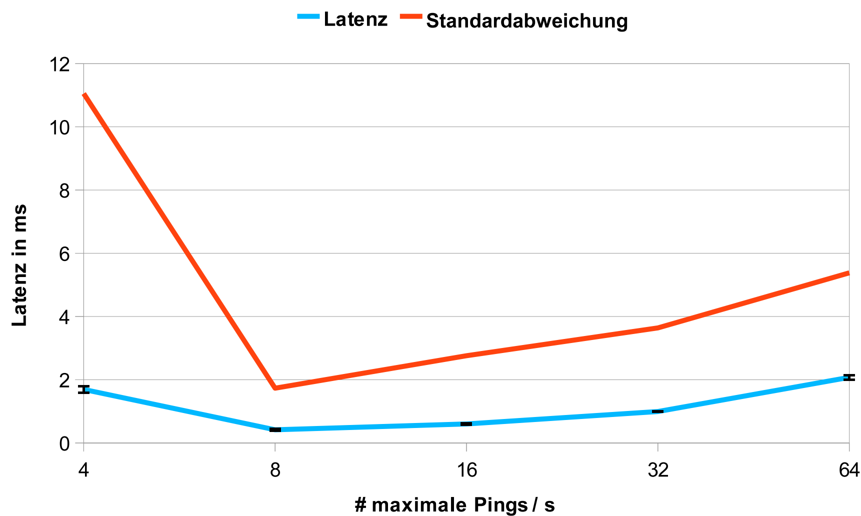


Abbildung 5.6: Latenz in Abhängigkeit der Last bei P2P-Replikation

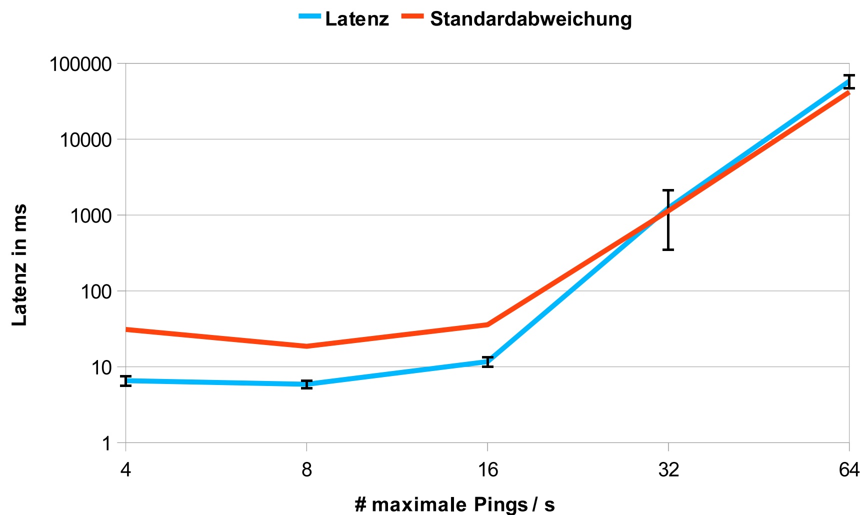


Abbildung 5.7: Latenz in Abhängigkeit der Last bei CS-Replikation

5.1.2.2 Hierarchische Replikation

Auf der ersten Ebene wird die im vorherigen Abschnitt beschriebene Client/Server-Variante eingesetzt, während es auf der zweiten Ebene, analog zur flachen Replikation, eine P2P-basierte und eine Client/Server-basierte Variante gibt. Bei der zweistufigen Replikation liegt das Hauptaugenmerk auf der Leistungsfähigkeit des Overlay-Verbundes, sowie dessen Auswirkungen

auf die Performanz. Von besonderem Interesse ist dabei nicht der maximale Durchsatz, da gerade durch die hierarchische Strukturierung der Datenverkehr zwischen den einzelnen Punkten minimiert werden soll, sondern die Latenz zwischen zwei Knoten, welche über den Overlay-Verbund miteinander kommunizieren.

Für die Messungen wurden deshalb die im vorherigen Abschnitt durchgeführten Latenzmessungen an diesen Aufbau adaptiert. Wieder schickt ein Knoten eine Aktualisierung an alle anderen Teilnehmer, welche jetzt aber alle an entfernten Replikationspunkten angesiedelt sind. Grafik 5.8 zeigt das Latenzverhalten beim Einsatz der Client/Server-Variante.

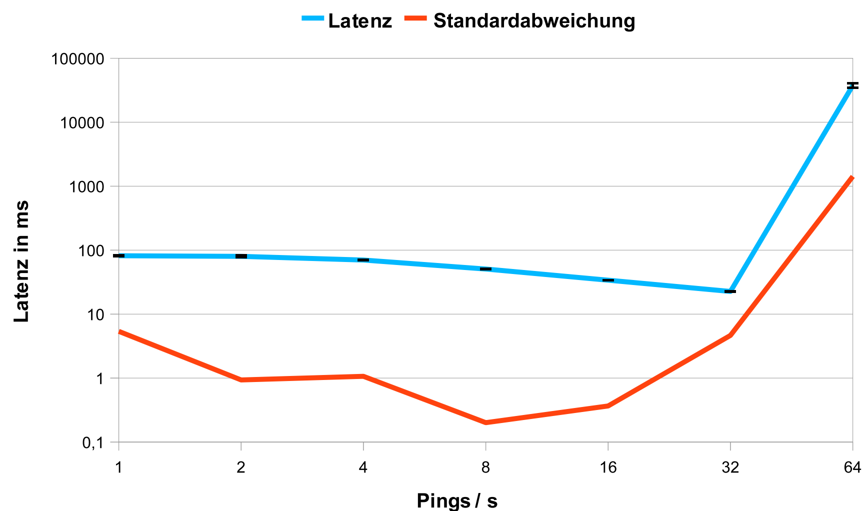


Abbildung 5.8: Latenz im hierarchischen Fall bei CS-Replikation

Erkennbar ist die deutlich gestiegene Latenz im Vergleich zur planaren Replikation. Auch lassen sich hier anfänglich (bis ca. 16 Pings pro Sekunde) deutliche Gewinne feststellen, was auf den konstanter werdenden Datenverkehr zurückzuführen ist.

5.1.3 TGOS-Operationen

Die erste Messung in diesem Abschnitt soll zeigen, wie viel Zeit die in wwShare implementierten TGOS-Operationen als Ganzes benötigen. Zu diesem Zweck wurde, analog zu den Serialisierungsmessungen, die Ausführungszeit der TGOS-Operationen ohne Replikationsschicht gemessen. Jede Operation wurde 1000-mal wiederholt und auf sechs Knoten parallel gestartet. Die Operationen verwendeten wiederum das Transformations-Objekt als Ziel, beziehungsweise als Quelle. Abbildung 5.9 zeigt die Kosten, nach Operatio-

nen gestaffelt, in Mikrosekunden. Die Fehlerbalken geben die Standardabweichung zwischen den Messreihen wieder.

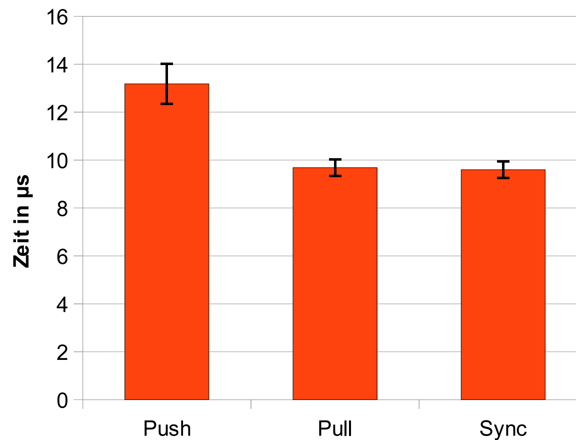


Abbildung 5.9: Ausführungszeit der TGOS-Operationen ohne Replikationsschicht

Wie aus den Messwerten ersichtlich, sind die Verwaltungskosten im Vergleich zu den Serialisierungskosten vernachlässigbar. Auch benötigt die serialisierende Push-Operation mehr Zeit, als die deserialisierende Pull-Operation, was sich ebenfalls mit den Ergebnissen der Serialisierungsoverhead-Messung deckt.

Die Messungen der TGOS-Operationen in Verbindung mit einer Implementierung der Replikationsschicht wurden analog zu den Replikationsmessungen in Abschnitt 5.1.2.1 durchgeführt. Als erste Messung wurde wieder die maximale Empfangsrate gemessen. Wichtig hierbei war, inwieweit die benötigten Serialisierungsoperationen die Empfangs- und Senderate beeinträchtigen würden. Gemessen wurde in diesem Fall mit der leistungsfähigeren JGroups-basierten Replikationsschicht. Der Messaufbau gestaltet sich analog zum Aufbau in 5.1.2.1.

Abbildung 5.10 zeigt die Ergebnisse der Messung. Die maximale Rate stagniert bei circa 10800 empfangenen Paketen pro Sekunde. Die von der TGOS-Implementierung wwShare verbrauchte Zeit fällt in diesem Fall nicht übermäßig ins Gewicht, da die Datenrate bei ausschließlicher Verwendung der Replikationsschicht nahezu identisch ist.

5.1.4 Transaktionale Konsistenz

Die Leistungsfähigkeit der mit Hilfe der TGOS-Basisoperationen implementierten transaktionalen Konsistenz soll anhand einiger synthetischer Tests ermittelt werden.

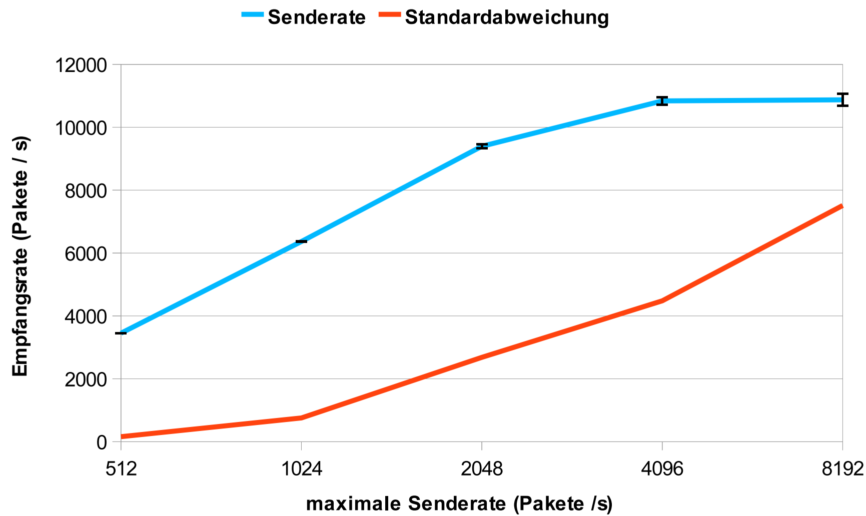


Abbildung 5.10: Maximale Empfangsrate für TGOS-Push-Operationen

Zuerst soll die theoretische Transaktionsrate in Abhängigkeit zur Knotenanzahl ermittelt werden. Dazu wurde zum Einen eine gemeinsame Variable von allen Knoten inkrementiert, um das Verhalten im Konfliktfall zu ermitteln. Zusätzlich wurde die Situation gemessen, dass alle Knoten eine disjunkte Variable inkrementieren. Hier treten zwar keine Konflikte auf, dennoch findet in der Validierungsphase der Transaktion eine Synchronisierung durch das Token statt. Die Messungen wurden unter Einsatz der CS-Replikation mit 16 Knoten des Opteron-Clusters durchgeführt. Jede Messung wurde mindestens viermal wiederholt, um Messfehler zu erkennen.

Abbildung 5.11 zeigt die Ergebnisse der Messung. Dabei wird auf der Y-Achse die durchschnittliche Transaktionsrate pro Knoten angegeben, auf der X-Achse die Anzahl an teilnehmenden Knoten. Wie nicht anders zu erwarten, sank die Transaktionsrate mit zunehmender Knotenanzahl, da jeder Knoten für ein erfolgreiches Commit das Token anfordern muss. Das gemeinsame Inkrementieren einer Variablen skalierte auf Grund der hohen Konfliktrate am Schlechtesten.

Neben der Transaktionsrate sind auch die laufenden Kosten für die Transaktion von Interesse. Der wesentlichste Kostenfaktor, neben dem Commit, ist hierbei das Erstellen der Lese- und Schreibmengen und das damit verbundene Erzeugen der Schattenkopien. Das Codefragment in Abbildung 5.1 zeigt die für die Messung verwendete Transaktion. Das Objekt, auf das für die Messungen innerhalb einer Transaktion zugegriffen wird, ist das aus Abschnitt 5.1.1 bekannt Transformations-Objekt, um die Ergebnisse in Relation zur Messung des Serialisierungsaufwands setzen zu können.

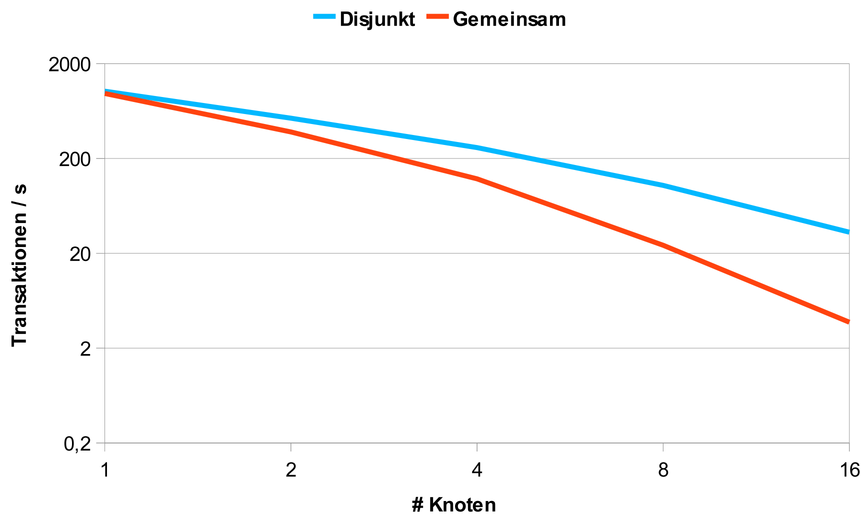


Abbildung 5.11: Transaktionsrate in Abhängigkeit zur Anzahl der Knoten

Listing 5.1: Codebeispiel für die transaktionale Aufwandsmessung

```
Transformation t = new Transformation();
do {
  TA.begin()
    TA.add2WriteSet(t);
    t.setTranslation(1,2,3);
} while (!TA.commit());
```

Die Transaktion wurde für die Messungen mit einer lokalen Replikationsschicht ohne Funktionalität betrieben und die Messungen wurden analog zu den Serialisierungsmessungen ausgeführt. Die *Begin*-Operation, welche eine Transaktion startet, integriert bei ihrem Aufruf alle Daten vorheriger Transaktionen, um einen aktuellen und konsistenten Zustand zu erstellen.

Um diesen Umstand für die Messungen zu simulieren, integrierte die Operation genau eine Änderung am Transformationsobjekt bei jedem Aufruf. Auf eine Messung der Erstellung der Lesemenge wurde verzichtet, da diese Operation lediglich einen Wert in eine Hash-Tabelle einträgt und somit keine signifikante Rechenzeit benötigt. Die *Commit*-Operation serialisiert zum Einen das Transformationsobjekt und synchronisiert zum Anderen das Token via Sync-Operation, was zusätzlich eine Deserialisierung und eine Serialisierung kostet. Ergänzend zu den direkten Operationen wurden auch die Kosten für das Speichern allfälliger Aktualisierungen gemessen, die asynchron auftreten können, aber nur bei einer Begin-Operation integriert

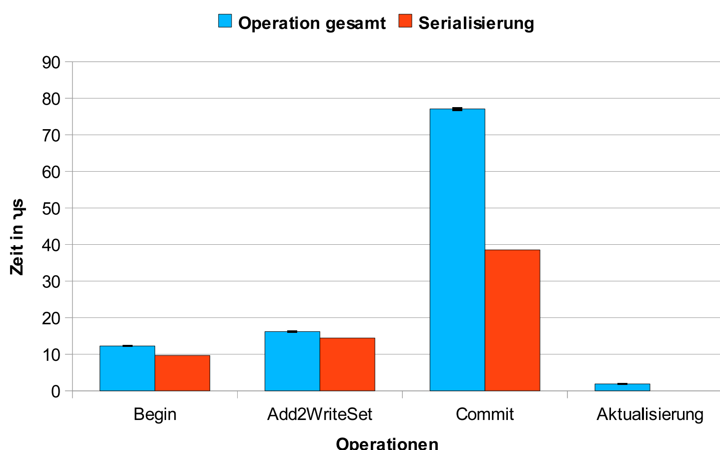


Abbildung 5.12: Kosten der transaktionalen Operationen

werden dürfen.

In Abbildung 5.12 sind die benötigten Zeiten, nach Operationen aufgeschlüsselt, dargestellt, sowie der Anteil der Serialisierungskosten an den Gesamtkosten. Bei allen Operationen machen die Serialisierungskosten den größten Anteil an den Gesamtkosten aus. Lediglich bei der Commit-Operation ist das Verhältnis ausgeglichen, da hier ein Großteil der transaktionalen Logik untergebracht ist.

Weitere Messungen der transaktionalen Konsistenz wurden in Verbindung mit Wissenheim Worlds durchgeführt und finden sich in Abschnitt 5.2.1.

5.2 Evaluation einer verteilten virtuellen Welt

Für die Evaluation einer verteilten virtuellen Welt wurde der anfangs beschriebene Wissenheim Worlds Prototyp verwendet. Der Prototyp ermöglicht dabei Messungen unter Einsatzbedingungen, weshalb diese im Gegensatz zu Abschnitt 5.1 nicht auf synthetischen Werten beruhen, sondern zum überwiegend größten Teil auf realen Daten. Im Rahmen eines universitären Projekts gestaltet es sich schwierig, Messungen mit hunderten von Benutzern durchzuführen, da eine solch große Anwenderzahl nur schwer zu erreichen ist. Aus diesem Grund wurden Aufzeichnungen von realem Benutzerverhalten in der verteilten Welt erstellt, welche in angepasster Form für computergestützte Benutzer verwendet wurden, um Tests mit mehr als 64 Benutzern durchführen zu können.

Wissenheim Worlds nutzt ausschließlich die Client/Server-basierte Replikationsschicht, kombiniert mit dem ebenfalls in Client/Server-Form realisierten Overlay-Verbund. Die Leistungscharakteristiken der beiden Repli-

kationssysteme wurden bereits in Abschnitt 5.1.2 beschrieben.

Für die Evaluation wurde neben dem in Abschnitt 5.1 bereits beschriebenen Opteron-Cluster auch das französische *Grid5000* [BCC+06] verwendet. Grid5000 ist ein Zusammenschluss mehrerer in Frankreich verteilter Cluster zu einem Gridverbund. Die Messungen wurden dabei jeweils innerhalb eines *Grid5000-Clusters* durchgeführt, bei dem die Knoten mit Gigabit-Ethernet über Netzwerkswitches untereinander verbunden sind. Die bei den Messungen beteiligten Knoten besitzen zwei AMD™ Opteron-Prozessoren (2.0 GHz) mit insgesamt 2GB Arbeitsspeicher. Da für die Messungen nur ein begrenztes Rechenzeitkontingent zur Verfügung stand, war es nicht möglich, mehrere Messreihen pro Experiment zu erstellen, weshalb auch keine Fehlerbalken in den Diagrammen angegeben werden können.

5.2.1 Area-of-Interest Messung

Wie in Kapitel 4 schon erwähnt, führt eine hohe Zahl an gleichzeitig aktiven Avataren in einer Szene, ohne zusätzliche Maßnahmen, schnell zu einer Sättigung des Netzes oder einer Überlastung des Replikationsknotenpunktes. Aus diesem Grund wurden die, ebenfalls in Kapitel 4 beschriebenen, Area-of-Interest-Konzepte prototypisch implementiert und für dieses Experiment verwendet. Im Rahmen der Messungen soll die Umsetzbarkeit der Area-of-Interest-Algorithmen im TGOS-Modell untersucht und belegt werden.

Die Messungen wurde auf Knoten des Grid5000-Cluster durchgeführt, wobei jeder Wissenheim Worlds Instanz eines Nutzer ein dedizierter Knoten zur Verfügung stand. Ebenso liefen der Login-Dienst, der Avatarspeicher und der Simulationsdienst der Szene auf disjunkten Knoten. Als Szene wurde dabei *RainbowIsland* verwendet, auf der es vier Zonen von besonderem Interesse gibt, welche untereinander außer Sichtweite sind. Die teilnehmenden Avatare wurden von einem zentralen Startpunkt aus gleichmäßig auf die vier Zonen verteilt. Für die Bewegung und Steuerung der Avatare wurden aufgezeichnete Bewegungsdaten realer Benutzer verwendet.

Abbildung 5.13 zeigt den aggregierten Bandbreitenbedarf der einzelnen Ansätze, während Abbildung 5.14 die aggregierte Aktualisierungsrate zeigt. Ohne AoI-Mechanismen war es nicht möglich, mehr als 64 Avatare gleichzeitig innerhalb einer Szene zu bewegen, da der Client/Server-basierte Replikationsknotenpunkt die aggregierte Bandbreite und/oder Aktualisierungsrate nicht bewältigen konnte. Die Aktualisierungsrate für den koordinatorbasierten Ansatz (im Diagramm als KOOR bezeichnet) liegt teils deutlich über der P2P-basierten Variante (P2P im Diagramm), was sich durch den verstärkten Koordinationsaufwand erklären lässt. Dieser tritt auf, wenn Avatare häufig den Sichtbereich eines anderen betreten und verlassen. Das Problem tritt insbesondere durch den Aufbau der RainbowIsland-Szene auf, da die Avatare auf vier relativ kleine Zonen verteilt wurden und nur einen vergleichs-

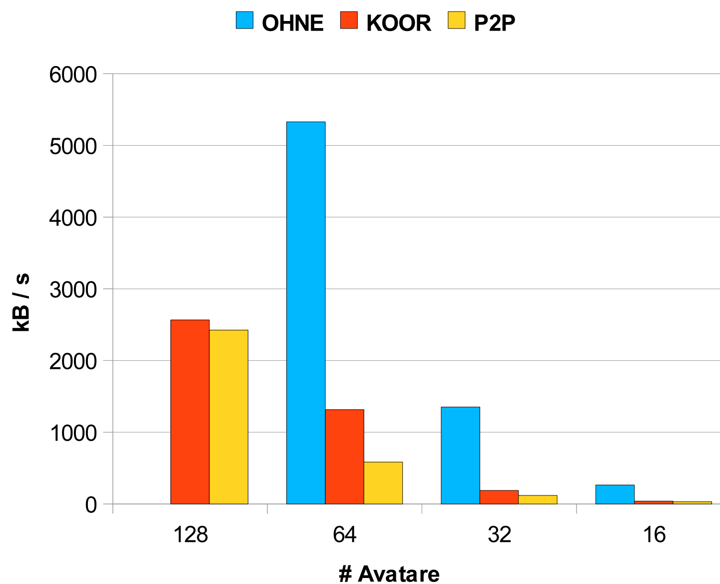


Abbildung 5.13: Aggregierter Bandbreitenbedarf mit AoI-Management

weise kleinen Sichtbereich besaßen. Um dieses Problem zu umgehen, wäre es denkbar, den Austritt aus dem Sichtbereich eines Avatars nicht sofort beim Verlassen durchzuführen, sondern erst nach einer Wartezeit Epsilon. Sollte der Avatar während dieser Zeit erneut in den Sichtbereich des anderen eintreten, müssten keine zusätzlichen Nachrichten verschickt werden.

Es wurden keine Szenenmessungen mit unterschiedlichen Latenzen durchgeführt, da Latenzen, wie sie beispielsweise durch Weitverkehrsverbindungen, DSL-Anschlüsse oder Drahtlosverbindungen entstehen, sich hauptsächlich auf die verwendeten Konsistenzmodelle und/oder Spielmechaniken auswirken. Deren Leistungsfähigkeit ist aber, mit Ausnahme der transaktionalen Konsistenz, nicht Gegenstand der Arbeit.

5.2.2 Messung der Lastverteilung

Wissenheim Worlds verwendet zur Lastverteilung die in Abschnitt 4.5.1.1 beschriebene statische Partitionierung der Welt und unterteilt diese in disjunkte Teilbereiche, die im Folgenden auch Szenen genannt werden. Jeder Teilbereich wird dabei von einem eigenständigen Dienst repräsentiert. Wechselt ein Nutzer von einer Szene zur nächsten, so wechselt er auch den Dienst und gegebenenfalls auch den Server.

Im folgenden Experiment wurde Wissenheim Worlds mit vier disjunkten Szenen gestartet. In der gesamten virtuellen Welt waren bis zu 128 Avatare unterwegs, die sich gleichmäßig auf alle Szenen verteilten. Die Avatare

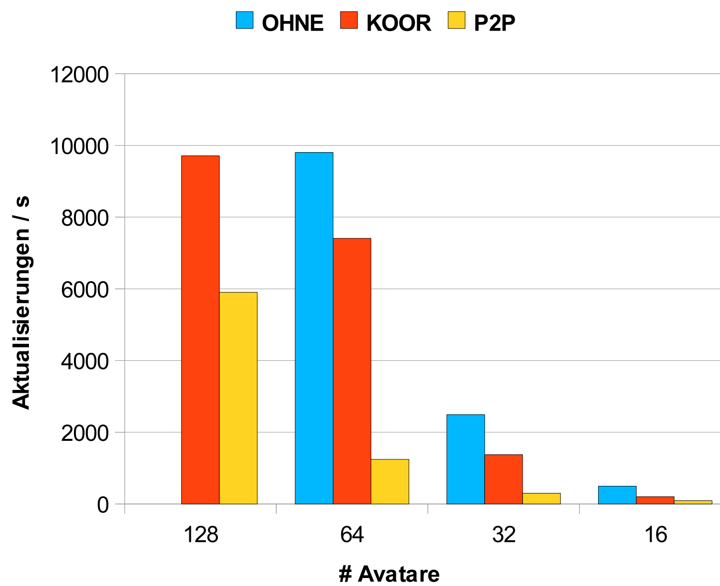


Abbildung 5.14: Aggregierte Aktualisierungsrate mit AoI-Management

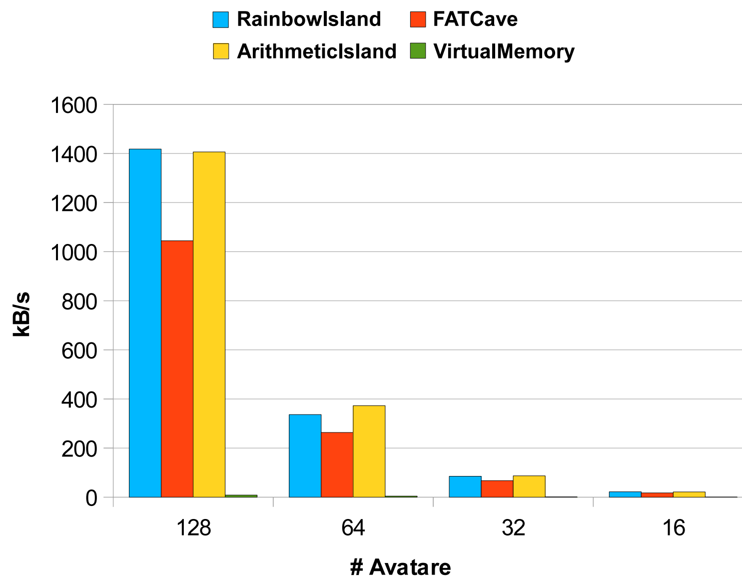


Abbildung 5.15: Aggregierter Bandbreitenbedarf pro Szene

verblieben während der Messung in ihrer Szene und wechselten nicht zu einer anderen. Jede Szene und jeder Teilnehmer hatte einen eigenen Knoten

zur Verfügung; der Login-Dienst, der Avatarspeicherdienst sowie der Client/Server-basierte Overlay-Relay teilten sich einen Knoten. Auf drei der vier Szenen bewegten sich die Avatare analog zu den Messungen in Abschnitt 5.2.1. Die vierte Szene, in diesem Fall *VirtualMemory* genannt, diente als Referenz für den Datenverkehr, falls keine Bewegungen stattfinden.

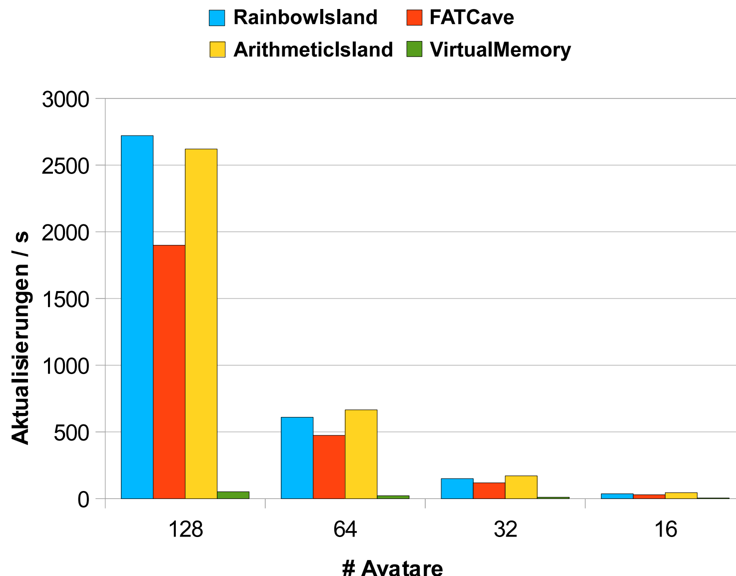


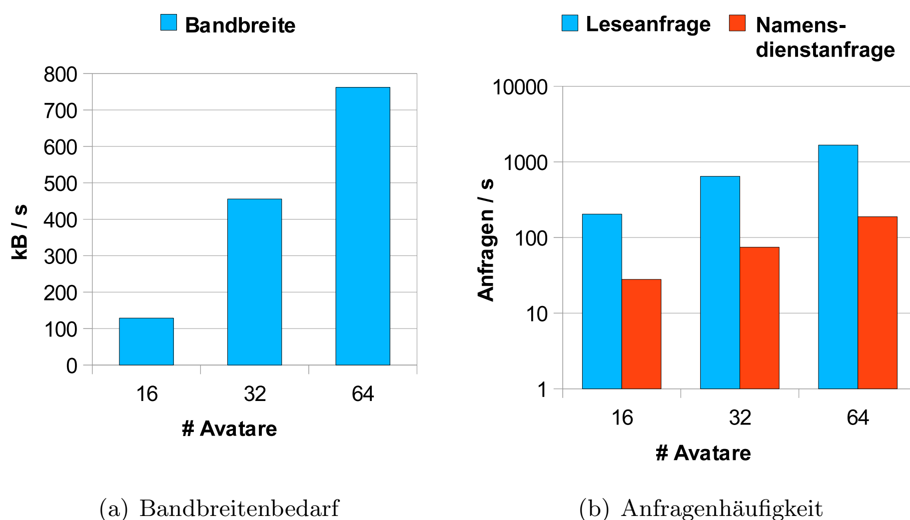
Abbildung 5.16: Aggregierte Aktualisierungsrate pro Szene

Abbildung 5.15 zeigt den aggregierten Bandbreitenbedarf pro Szene, in Abhängigkeit zur Gesamtzahl an teilnehmenden Knoten. Die Avatare sind dabei gleichverteilt, was bedeutet, dass zu jeder Zeit pro Szene ein Viertel aller Avatare vorhanden ist. Wie an der Referenzszene zu sehen ist, machen die Bewegungsdaten den überwiegenden Teil des Datenverkehrs aus. Die pro Szene unterschiedlichen Bewegungsmuster machen sich auch in der Datenrate bemerkbar. Abbildung 5.16 zeigt zusätzlich die aggregierte Aktualisierungsrate pro Szene bei steigender Teilnehmerzahl, deren Verhalten sich analog zu dem der Datenrate entwickelt.

Die Datenrate zwischen den Diensten war mit knapp *zwei Kilobytes pro Sekunde* wie erwartet vergleichsweise gering, da Datenverkehr nur durch den Zeitdienst und das Sammeln der Messwerte auftraten und es durch die statische Partitionierung zu keinem zusätzlichen Datenverkehr zwischen den Szenen kam.

Im aktuell laufenden Wissenheim Worlds verbleiben die Avatar für die Dauer ihres Aufenthaltes nicht in einer Szene, sondern wechseln diese häufiger. Jeder Wechsel erzeugt Datenverkehr im Overlay-Netzwerk, da beispielsweise

beim Verlassen einer Szene die Statusinformation eines Avatars sowie seine Autorisierungsinformationen aktualisiert werden müssen. Verbindet sich ein Avatar mit einer Szene, so werden wiederum diese Avatar- und Autorisierungsinformationen durch den betroffenen Dienst geladen. Zusätzlich fordern alle anderen Teilnehmer dieser Szene die aktuellsten Avatardaten vom Avatarspeicherdienst an. Da die Szenedienste die Avatardaten im Moment nicht zwischenspeichern, müssen die Daten bei jeder Anfrage erneut vom Avatarspeicher(Olymp) geladen werden.



(a) Bandbreitenbedarf (b) Anfragenhäufigkeit

Abbildung 5.17: Last bei Szenenwechsel

Die Messungen simulierten nun den Wechsel der Avatare zwischen den Szenen. Die durchschnittliche Verweildauer eines Avatars innerhalb einer Szene lag bei 30 Sekunden. Die Avatare wurden anfänglich gleichmäßig auf vier Szenen verteilt und wechselten dann reihum. Während der Messungen bewegten sich die Avatare nicht, so dass der gemessene Bandbreitenbedarf und die Anfragehäufigkeiten nur auf den Avatarwechsel zurückzuführen sind.

Der ermittelte Bandbreitenbedarf ist in Abbildung 5.17(a) dargestellt. Im Gegensatz zum vorherigen Experiment konnte diese Messung nur mit maximal 64 Knoten vorgenommen werden, da bei 128 wechselnden Knoten die Last auf den Avatarspeicherdienst (Olymp) zu groß wurde. Abbildung 5.17(b) zeigt die Anzahl an Objekt- und Namensdienstanfragen im Overlay, die durchschnittlich pro Sekunde gemessen wurden. Zu beachten ist, dass eine logarithmische Skala verwendet wird; die Anfrageanzahl steigt also quadratisch mit der Teilnehmerzahl.

Im Gegensatz zum vorherigen Experiment tritt hier enormer Verkehr im Overlaynetz auf und die Anfragen steigen sogar quadratisch. Obwohl eine so kurze Verweildauer eines Avatars im realen Betrieb nur selten zu

beobachten ist - die meisten Nutzer wollen die Inhalte einer Szene betrachten, beziehungsweise mit den Exponaten dort interagieren - können solche Situationen gerade in Durchgangsszenen vorkommen. Als Durchgangsszene bezeichnet man Gebiete, die wenig Inhalte bieten, aber von denen aus ein Spieler in viele andere Gebiete gelangen kann. Problematisch wird das häufige Wechseln hauptsächlich durch das fehlende Zwischenspeichern der Avatardaten im Szenendienst, wodurch jeder Avatar in der Szene die Daten redundant beim Avatarspeicherdienst anfordert.

5.2.3 Transaktionale Konsistenz

Zusätzlich zu den Messungen der transaktionalen Konsistenz im Rahmen der Evaluation der Basiskomponenten in Abschnitt 5.1.4, wurden auch Messungen im Rahmen von Wissenheim Worlds durchgeführt. Neben der Sicherung der Szenengraphstruktur werden Transaktionen insbesondere auch für den Textchat der Szene verwendet. Dabei wird jedes Hinzufügen einer Nachricht mit einer Transaktion gesichert. Eine wichtige Eigenschaft der hier vorgestellten transaktionalen Konsistenz ist, dass nur schreibende Zugriffe Synchronisierungskosten verursachen. Nur lesende Knoten sollten daher auch bei steigenden Latenzen und/oder steigender Konfliktrate keine Einbußen feststellen.

Um dieses Verhalten zu belegen, wurden Messungen mit dem bereits erwähnten Textchat der Szene durchgeführt, bei der die Auswirkungen des lesenden und schreibenden Zugriffs auf den Chat untersucht wurden. Zu diesem Zweck wurde die Bildwiederholrate der Klienten gemessen, welche vorher von allen nicht transaktionsspezifischen Effekten bereinigt und auf 60 Bilder pro Sekunde begrenzt wurde. Da die Ausführung der Transaktionen innerhalb des Klienten synchron erfolgt, verringert eine längere Ausführungszeit automatisch die Bildwiederholrate. Für die Messungen verbanden sich 32 Teilnehmer, welche von eins bis 32 durchnummeriert waren, mit einer Szene. Jeder Teilnehmer mit einer geraden Nummer schrieb eine Chatnachricht einmal pro Sekunde, alle ungeraden Teilnehmer griffen nur lesend auf den Chat zu. Gemessen wurde die gemittelte Bildwiederholrate aller Schreiber, im Vergleich zur der aller Leser bei steigender Latenz. Unter Latenz wird dabei immer die Umlaufzeit eines Paketes zwischen Klient und Replikationsknotenpunkt (SuperPeer) verstanden.

Abbildung 5.18 zeigt das Ergebnis der Messungen. Wie deutlich zu sehen ist, nimmt die Bildwiederholrate der schreibenden Knoten kontinuierlich mit der Latenz ab, während die der lesenden Knoten konstant bleibt. Abbildung 5.19 zeigt die Standardabweichung der gemessenen Bildwiederholraten, bei der deutlich zu erkennen ist, dass die Abweichungen nur bei den schreibenden Knoten zunehmen. Die Ergebnisse lassen sich direkt auf den für die Synchronisierung der Validierungsphase verwendeten Tokenmechanismus und dessen Eigenschaften (siehe Abschnitt 4.3.2.1) zurückführen.

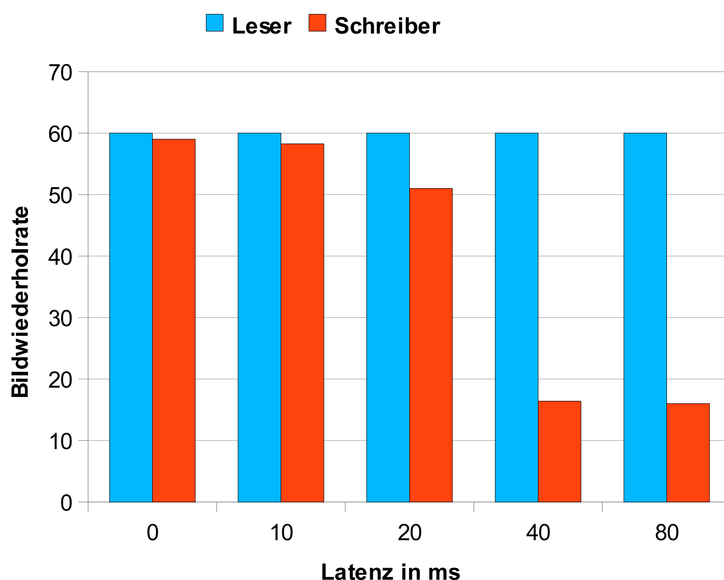


Abbildung 5.18: Bildwiederholrate des transaktionalen Chats bei steigender Latenz

Um die Auswirkungen der schreibenden Transaktionen zu begrenzen, wäre es möglich, diese in eigene Threads auszulagern, um so die Transaktion parallel im Hintergrund ablaufen zu lassen.

5.2.4 Mobile Klienten

Zusätzlich zu den Messungen auf Standard-PC-kompatiblen Geräten wurden auch Messungen im Hinblick auf mobile Geräte durchgeführt. Dazu wurde Wissenheim Worlds auf Android 2.1 portiert und die Messungen mit einem HTC Desire Smartphone durchgeführt (siehe Abbildung 5.20). Für die Messungen standen sowohl drahtlose Netzwerke nach 802.11b/g zur Verfügung, als auch ungedrosselte 3G-Verbindungen im ePlus-Netz.

Da die Rechenkapazität auf mobilen Geräten wie Smartphones oder Tablets im Vergleich zu Standard-PCs deutlich eingeschränkt ist, soll im ersten Experiment die durch TGOS verursachte Rechenlast untersucht werden. Zu diesem Zweck wurde wiederum die bereinigte Bildwiederholrate auf dem mobilen Gerät in Abhängigkeit zusätzlich an einer Szene teilnehmender Avatare gemessen. Bereinigt bedeutet in diesem Fall, dass alle weiteren negativen Effekte, wie beispielsweise zusätzliche Grafik- oder Physiklast, nicht in die Messergebnisse einfließen. Die Messungen wurden zum einen mit Wifi nach 802.11g als auch mit einer 3G-Verbindung über das ePlus-Netz durchgeführt. Während der Messungen war eine durchschnittliche Latenz von 80ms für

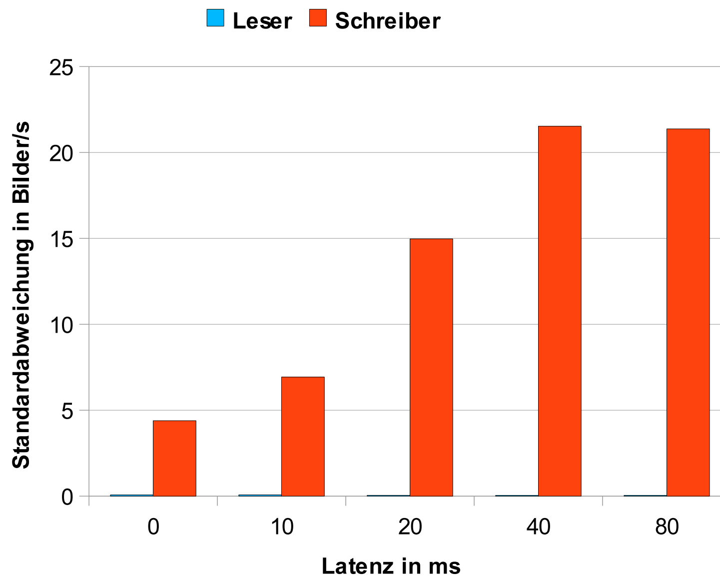


Abbildung 5.19: Standardabweichung des transaktionalen Chats

Wifi und 500ms für die 3G-Verbindung beobachtbar.

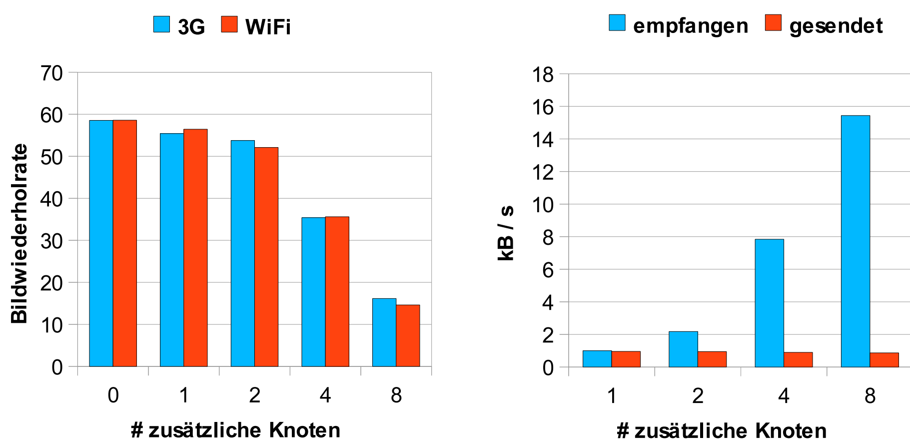
Abbildung 5.21(a) zeigt, wie die Bildwiederholrate mit zunehmender Anzahl an Avataren sinkt. Der größte Kostenfaktor war dabei die automatische Serialisierung, welche auf mobilen Geräten deutlich mehr Rechenzeit beansprucht, als auf Standard-PCs. Abbildung 5.21(b) zeigt die auf dem mobilen Gerät benötigte Bandbreite in Abhängigkeit zur Avataranzahl.

5.2.5 Feldtests

Im Rahmen der Vorlesung Verteilte Systeme an der Universität Düsseldorf wurde Wissenheim Worlds auch als Basis für virtuelle Übungen benutzt, bei denen die Studenten in der virtuellen Welt gemeinsam Übungsaufgaben lösten. Alle Teilnehmer der Übung nutzten das unter [Wor10] bereitgestellte Java-Applet, um der virtuellen Welt von Wissenheim Worlds beizutreten. Dabei waren alle Teilnehmer im Großraum Düsseldorf präsent, die Wissenheim Worlds Hintergrunddienste liefen auf Servern an der Universität Ulm.

Das Hauptproblem bei Messungen von Feldtests besteht in der schwierigen Korrelation und Vergleichbarkeit der gewonnenen Daten, da in einem realen Umfeld eine sehr große Heterogenität vorherrscht. Dazu zählen beispielsweise Auswirkungen durch unterschiedliche Ausrüstungen der beteiligten Computer, wie Grafikhardware, Netzanbindung oder Betriebssysteme. Auch die Geschicklichkeit eines Benutzers im Umgang mit seinem virtuellen Avatar oder sein Laufverhalten haben starken Einfluss auf die gewonnen

Abbildung 5.20: Wissenheim Worlds auf HTC Desire mit Android 2.1.



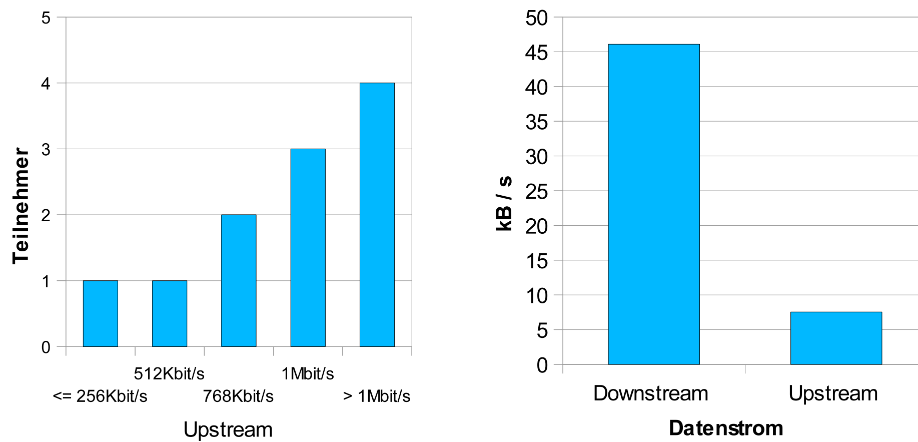
(a) Bereinigte Bildwiederholrate (b) Durchschnittlich benötigte Bandbreite

Abbildung 5.21: Wissenheim Worlds auf mobilen Geräten

Messwerte.

Interessante Messgrößen waren neben der Art der Anbindung der Nutzer, die in Abbildung 5.22(a) dargestellt ist, auch die im realen Einsatz benötigte durchschnittliche Bandbreite, welche in Abbildung 5.22(b) zu sehen ist. Der benötigte Upstream ist, mit durchschnittlich 7kB/s, vergleichbar mit Messwerten bestehender kommerzieller Welten [KCC+05].

Abbildung 5.23 gibt einen Überblick über das Latenzempfinden der Nutzer. Während des Testlaufes waren die Dead-Reckoning-Mechanismen deaktiviert, wodurch die empfundene Latenz und Varianz direkt auf das Netzwerk beziehungsweise auf die Replikationsschicht zurückzuführen war. Dennoch bewerteten fast die Hälfte aller Teilnehmer die Bewegungen anderer Avatare als zufriedenstellend.



(a) Anbindungen der Nutzer (b) Durchschnittliche benötigte Bandbreite

Abbildung 5.22: Ergebnisse des Feldtests

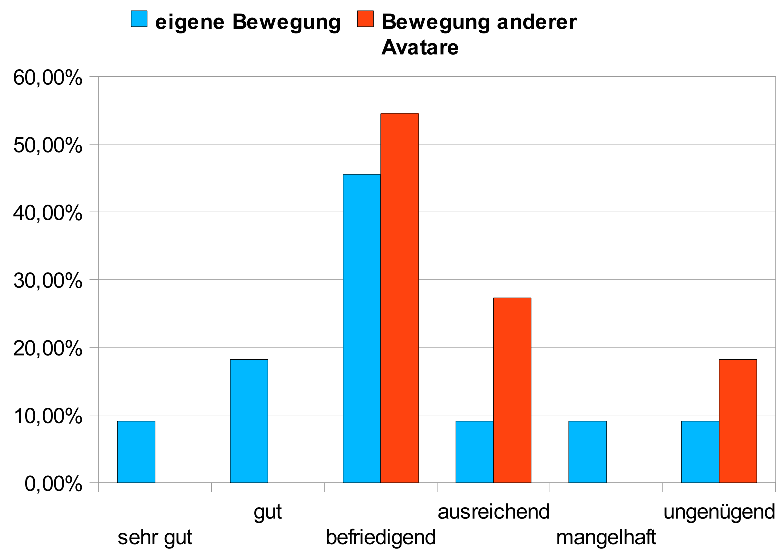


Abbildung 5.23: Latenzempfinden der Benutzer

5.3 Bewertung

Bei der Evaluation der Basiskomponenten zeigte sich, dass die größten Kosten in der wwShare-Implementierung des TGOS-Modells durch die automatischen Serialisierungs- beziehungsweise Deserialisierungsfunktionen verur-

sacht werden. Diese sind zudem sehr implementationsabhängig und können durch die in Abschnitt 5.21 angeregte manuelle Serialisierungsunterstützung deutlich verringert, beziehungsweise auf die manuelle Variante reduziert, werden. Im Vergleich zu Standard-Java gewinnt die implementierte Serialisierung sowohl bei benötigter Rechenzeit, als auch bei der Größe der serialisierten Form. Auch zeigen die Messungen, dass neben der Serialisierung der Auswahl der Replikationsschicht eine wichtige Bedeutung zukommt, da sie die maximal erreichbare Performanz begrenzt. Der zusätzliche Aufwand, der durch die internen Verwaltungsstrukturen der wwShare-Implementierung entsteht, ist im Vergleich zu den Kosten der De-/Serialisierung und der Replikationsschicht vernachlässigbar.

Im Rahmen der Evaluation einer verteilten virtuellen Welt, in diesem Fall Wissenheim Worlds, konnte die Eignung des TGOS-Modells für die Umsetzung der für virtuellen Welten so wichtigen Area-of-Interest- und Lastverteilungsmechanismen demonstriert werden. Auch hier war bei den Messungen die Güte der Replikationsschicht der ausschlaggebende Faktor. Auch zeigt sich am Beispiel der Avatare, wie wichtig eine Partitionierung der Welt und ein geeignetes Replikat-Caching ist, um die Last im Overlay-Netzwerk so gering wie möglich zu halten.

Bei den synthetischen Messungen der transaktionalen Konsistenz zeigte sich, dass auch hier die Implementation der automatischen Serialisierungsfunktionalität den größten Anteil am gemessenen Aufwand verursachte. Die Logik des Konsistenzmodells verursacht nur einen relativ geringen Anteil an den Gesamtkosten. Die Messungen in Zusammenhang mit Wissenheim Worlds belegen auch die in Abschnitt 4.3.2.1 beschriebenen Vorzüge eines optimistischen Protokolls mit Rückwärtsvalidierung und Aktualisierungen. Insbesondere die Behauptung, dass Knoten, die nur lesend zugreifen, keinen signifikanten Mehraufwand für die Synchronisierung benötigen, konnte verifiziert werden.

Die Messergebnisse der mobilen Klienten zeigen, dass das TGOS-Modell auch für diese Kategorie von Geräten grundsätzlich geeignet ist, auch wenn im Detail noch Optimierungsbedarf bei der implementierten Umsetzung besteht.

Die im Rahmen des Feldtests gewonnenen Ergebnisse lassen zwar keine direkten Rückschlüsse auf die Leistungsfähigkeit des TGOS-Modells zu, jedoch zeigt sich, dass der Ansatz auch in einem realen Einsatz zuverlässig funktioniert und die benötigte Datenrate der Implementierung mit kommerziellen Systemen vergleichbar ist.

Kapitel 6

Zusammenfassung

6.1 Resultat

Im Rahmen der Arbeit wurde ein neuartiges Programmiermodell, das TGOS-Modell, entwickelt, welches speziell an die Bedürfnisse und Anforderungen einer virtuellen verteilten Welt mit einer Vielzahl von Nutzern angepasst ist. Das Modell kombiniert dabei das aus DSM-Systemen bekannte Konzept der datenzentrierten Programmierung mit dem Ereignischarakter und der Explizität nachrichtenorientierter Ansätze.

Das datenzentrierte Programmiermodell ermöglicht dabei eine deutlich einfachere Programmierung einer virtuellen Welt, da statt asynchroner Aufrufe ein synchroner Programmablauf verwendet werden kann. Zusätzlich entfällt die aus Client/Server-Systemen bekannte Aufsplittung des Programmes, wodurch die Komplexität der Anwendung deutlich gesenkt wird. Durch die Integration des expliziten Charakters kann der, durch die Verteilung entstehende, Netzwerkverkehr zu jeder Zeit durch die Anwendung kontrolliert und die verursachte, beziehungsweise benötigte, Bandbreite bestimmt werden, was insbesondere für den Einsatz in virtuellen Welten eine notwendige Bedingung darstellt.

Eine weitere Neuheit des TGOS-Modells ist die Rolle der Replikationsschicht, die im Gegensatz zu anderen Ansätzen für den Nutzer nicht transparent ist, sondern ein wesentlicher und vor allem sichtbarer Teil des Programmiermodells ist. Durch die Abstraktion der Replikationsschicht kann diese einfach ausgetauscht werden, ohne dabei die mit Hilfe der TGOS-Basisoperationen erstellten Algorithmen anpassen zu müssen.

Ein weiterer wichtiger und neuartiger Ansatz ist die Integration und Definition von neuen Konsistenzmodellen innerhalb des TGOS-Programmiermodells. Die Konsistenzmodelle nutzen dabei ausschließlich die im TGOS-Modell definierten Operationen und Eigenschaften, wodurch auch eine einfache Modularisierung möglich wird. Auch der Ansatz, Konsistenz im Programmfluß festzulegen, anstatt, wie bei DSM-Systemen üblich, auf Basis der

Daten oder Speicherbereiche, stellt eine Neuerung dar. Durch diesen Ansatz ist es beispielsweise möglich, neue Konsistenzmodelle im laufenden Betrieb zu implementieren und anzuwenden.

Neben der allgemeinen Betrachtung der Umsetzbarkeit verschiedener Konsistenzmodelle mit TGOS, wurde insbesondere die Eignung der transaktionalen Konsistenz für den Einsatz innerhalb einer verteilten virtuellen Welt untersucht. Dabei konnte gezeigt werden, dass sich optimistische Transaktionen (mit Rückwärtsvalidierung und Aktualisierungen) sehr gut zur Sicherung kritischer Strukturen, welche nur selten geändert werden, eignen. Als besonders vorteilhaft erweist sich die Tatsache, dass für Leser die Synchronisierung keinen zusätzlichen Aufwand kostet.

Auch die in verteilten Welten wichtigen Lastverteilungs- und Area-of-Interest-Algorithmen lassen sich mit Hilfe des TGOS-Modells sehr einfach umsetzten. Analog zu den Konsistenzmodellen sind auch diese unabhängig von der jeweiligen Netzarchitektur und müssen bei eine Änderung derselben nicht angepasst werden. Auch kann analog zu den Konsistenzmodellen sehr leicht eine Modularisierung erfolgen.

Das in der Arbeit vorgestellte TGOS-Programmiermodell wurde theoretisch definiert und im Rahmen der prototypischen virtuellen Welt Wissenheim Worlds realisiert. Anhand des Prototyps konnte sowohl in synthetischen Messungen, als auch in Messungen unter Einsatzbedingungen, die Leistungsfähigkeit des Ansatzes bestätigt sowie die Umsetzbarkeit aller für eine virtuelle Welt wichtigen Mechanismen gezeigt werden.

Dabei zeigte die Evaluation der Basiskomponente der TGOS-Implementierung wwShare, dass der für wwShare benötigte Verwaltungsaufwand, im Vergleich zu den Kosten der automatischen Serialisierung und der Replikationsschicht, vernachlässigbar ist. Auch können die Kosten der automatischen Serialisierung durch geeignete Optimierungen verringert und gegebenenfalls auf das Niveau einer manuellen Serialisierung gesenkt werden. Durch den Einsatz mehrerer Replikationsschichten mit unterschiedlichen Netzarchitekturen wurde die theoretisch definierte Netzarchitekturunabhängigkeit des TGOS-Modells untersucht und bestätigt. Dabei zeigte sich, wie stark die Auswirkungen der Replikationsschicht auf die Leistungsfähigkeit des Gesamtsystems sind.

Im Rahmen der Evaluation der virtuellen Welt Wissenheim Worlds wurde die Realisierbarkeit und Effizienz, der für eine virtuelle Welt essentiellen Area-of-Interest- und Lastverteilungsmechanismen untersucht und anhand Messungen mit über hundert Knoten bestätigt. Zusätzlich stellte das Gesamtsystem in Feldtests seine praktische Verwendbarkeit unter Beweis.

Die Evaluation des, mit Hilfe der TGOS-Basisoperationen realisierten, transaktionalen Konsistenzmodells zeigte, dass sich auch komplexere Konsistenzmodelle einfach umsetzen lassen und diese nur geringe laufende Kosten verursachen.

Zusätzlich zu den gängigen PC-Plattformen wurde der TGOS-Ansatz

auch auf mobilen Geräten, am Beispiel eines Android-Smartphones, praktisch evaluiert und die Eignung bestätigt.

Der Prototyp ermöglichte es weiterhin, umfangreiche Erfahrungen mit dem Betrieb einer verteilten virtuellen Welt zu sammeln. Zusätzlich zu den Ergebnissen auf technischer Ebene dient Wissenheim Worlds auch der Forschung im Bereich eLearning [BSSW09, BSWS09] und wird für virtuelle Vorlesungen und zur Unterstützung vorlesungsbegleitender Übungen an den Universitäten Ulm und Düsseldorf verwendet.

6.2 Ausblick

Die im Rahmen der Arbeit entwickelten und prototypisch verwirklichten Konzepte stellen eine fundierte Basis dar, anhand der weiterführende Forschung im Bereich der datenzentrierten verteilten virtuellen Welten erfolgen kann. Die im Folgenden vorgestellten Punkte stellen mögliche Schwerpunkte für weiterführende Arbeiten dar.

Cloud Computing

Die Definition von Cloud Computing ist in der Literatur nicht eindeutig, eine bekannte und viel zitierte Zusammenfassung des Themas findet sich in [AFG+09]. Als kleinster gemeinsamer Nenner kann die Bereitstellung einer Rechen- und Persistenzierungsinfrastruktur angesehen werden, bei der Ressourcen dynamisch alloziert werden können. Die Allokation und Deallokation erfolgt dabei innerhalb weniger Sekunden. Das Bezahlmodell der meisten Cloud-Systeme verwendet dabei die verbrauchten Ressourcen als Berechnungsgrundlage, wodurch es sich besonders für den Ausgleich von Spitzenlasten anbieten.

Gerade die Hintergrunddienste einer virtuellen Welt sind für den Einsatz innerhalb einer Cloud prädestiniert. Die Eignung des TGOS-Modells für die Umsetzung von Lastverteilungs und Area-of-Interest-Algorithmen für die Cloud sowie das Zusammenspiel zwischen TGOS und Replikation bietet interessante Forschungsansätze.

Mobile Klienten

Im Rahmen der Arbeit wurde die Eignung des TGOS-Modells für mobile Geräte untersucht und durch Messungen auf realer Hardware bestätigt. Ausgehend von dieser Konzeptstudie ist die vollständige Integration mobiler Geräte in eine virtuelle Welt ein sehr umfangreiches Thema. Neben der schwierigen Netzanbindung stellen sowohl der eingeschränkt zur Verfügung stehende Hauptspeicher und die geringe Rechenleistung große Herausforderungen für eine verteilte virtuelle Welt dar. Insbesondere Algorithmen zur Detailsteuerung und Latenzverdeckung kommen besondere Bedeutung zu.

Dynamische Evolution

In zunehmendem Maße werden in kommerziellen virtuelle Welten Inhalte generiert, die dynamisch durch die Nutzer gestaltet, beziehungsweise beeinflusst werden. Eine Welt muss daher die Möglichkeit bieten, vom Benutzer erstellte Codefragmente und Mediendaten während des laufenden Betriebs in die Welt zu integrieren. Inwieweit diese Vorgaben innerhalb des TGOS-Modells umgesetzt werden können oder ob dieses gegebenenfalls erweitert werden muss, stellt interessante Fragestellungen dar.

Sicherheit

Die Gewährleistung von Sicherheit ist für verteilte virtuelle Welten von entscheidender Bedeutung. Im Rahmen der Arbeit wurden bei der Umsetzung des Wissenheim Worlds Prototyps zwar einfache Sicherheitsmechanismen für Authentisierung und Autorisierung umgesetzt, jedoch fehlen noch Konzepte und Mechanismen, um böswillige Angriffe zu erkennen und zu verhindern. Insbesondere der datenzentrierte Ansatz des TGOS-Modells stellt neue Herausforderungen an eine Sicherheitsarchitektur.

Literaturverzeichnis

[ACD+96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, Honghui Lu, R. Rajamony, Weimin Yu, and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29(2):18 –28, feb. 1996.

[ACRZ97] Cristiana Amza, Alan Cox, Karthick Rajamani, and Willy Zwaenepoel. Tradeoffs between false sharing and aggregation in software distributed shared memory. *SIGPLAN Not.*, 32:90–99, June 1997.

[Act] Blizzard Activision. www.worldofwarcraft.com.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, University of Berkeley, 2009.

[AG96] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66 –76, dec. 1996.

[Bar03] Richard Bartle. *Designing Virtual Worlds*. New Riders, 2003.

[BCC+06] Raphael Bolze, Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Irea Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. *SIGPLAN Not.*, 25:168–176, February 1990.

[BF93] Steve Benford and Lennart Fahlén. A spatial model of interaction in large virtual environments. In *ECSCW'93: Proceedings of the third conference on European Conference on Computer-Supported Cooperative Work*, pages 109–124, Norwell, MA, USA, 1993. Kluwer Academic Publishers.

[BKH04] W. Xu B. Knutsson, H. Lu and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, 2004.

[BKV06] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 6, New York, NY, USA, 2006. ACM.

[BO04] Don Burns and Robert Osfield. Open scene graph a: Introduction, b: Examples and applications. In *VR '04: Proceedings of the IEEE Virtual Reality 2004*, page 265, Washington, DC, USA, 2004. IEEE Computer Society.

[BSSW09] Tobias Bäuerle, Michael Sonnenfroh, Peter Schulthess, and Alexander Weggerle. Wissenheim - an interactiv 3d-world for leisure and learning. In *Proceedings of International Conference of Education, Research and Innovation (ICERI)*, 2009.

[BSWS09] Tobias Baeuerle, Michael Sonnenfroh, Alexander Weggerle, and Peter Schultess. Wissenheim worlds - a massively multi-user environment focused on e-action-learning. In *Proceedings of International Conference and Industry Symposium on Computer Games, Animation, Multimedia, IPTV, Edutainment and IT Security, Singapore*, 2009.

[BT01] Paul Bettner and Mark Terrano. 1500 archers on a 28.8: Network programming in age of empires and beyond. In *GDC*, 2001.

[Cat72] Edwin Catmull. A system for computer generated movies. In *ACM '72: Proceedings of the ACM annual conference*, pages 422–431, New York, NY, USA, 1972. ACM.

[CC91] Roger S. Chin and Samuel T. Chanson. Distributed, object-based programming systems. *ACM Comput. Surv.*, 23:91–124, March 1991.

[CCE+07] Edward Castronova, James J. Cummings, Will Emigh, Michael Fatten, Nathan Mishler, Travis Ross, and Will Ryan. What is a synthetic world? In Friedrich Borries, Steffen P. Walz, and Matthias Boettger, editors, *Space Time Play*, pages 174–177. Birkhaeuser Basel, 2007. 10.1007/978-3-7643-8415-9_63.

[CDW07] Sarah Coleman and Nick Dyer-Witheford. Playing on the digital commons: collectivities, capital and contestation in videogame culture. *Media, Culture & Society*, 29(6):934–953, 2007.

[CN93] Pavel Curtis and David A. Nichols. Muds grow up: Social virtual reality in the real world. In *In Proceedings of the Third International Conference on Cyberspace*, pages 193–200, 1993.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM COMPUTING SURVEYS*, 17(4):471–522, 1985.

[DD09] Alokika Dash and Brian Demsky. Software transactional distributed shared memory. *SIGPLAN Not.*, 44:297–298, February 2009.

[dJ77] Lieuwe Sytse de Jong. Towards a formal definition of numerical stability. *Numerische Mathematik*, 28(2):211–219, June 1977.

[DRH07] Jörg Domaschka, Hans P. Reiser, and Franz J. Hauck. Towards generic and middleware-independent support for replicated, distributed objects. In *Proceedings of the 1st workshop on Middleware-application interaction: in conjunction with Euro-Sys 2007*, MAI '07, pages 43–48, New York, NY, USA, 2007. ACM.

[DYNM07] Nicolas Ducheneaut, Nicholas Yee, Eric Nickell, and Robert J. Moore. The life and death of online gaming communities: a look at guilds in world of warcraft. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 839–848, New York, NY, USA, 2007. ACM.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[EGJ95] Richard Helm Erich Gamma and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, 1995.

[FFSS06] Markus Fakler, S. Frenz, M. Schottner, and P. Schulthess. A demand-driven approach for a distributed virtual environment. *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, pages 1538–1541, May 2006.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice.* Addison-Wesley, 1999.

[FRS05] Tobias Fritsch, Hartmut Ritter, and Jochen Schiller. The effect of latency and network limitations on mmorpgs: a field study of everquest2. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–9, New York, NY, USA, 2005. ACM.

[FS98] Emmanuel Frecon and Marten Stenius. Dive: a scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 5(3):91, 1998.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[GFSS03] Ralph Goeckelmann, Stefan Frenz, Michael Schoettner, and Peter Schulthess. Compiler support for reference tracking in a type-safe dsm. In *Modular Programming Languages*, volume 2789 of *Lecture Notes in Computer Science*, pages 49–58. Springer Berlin / Heidelberg, 2003.

[Gof66] Erving Goffman. *Behavior in public places: Notes on the social organization of gatherings.* Simon and Schuster, 1966.

[Gri10] Mark Griffiths. The role of context in online gaming excess and addiction: Some case study evidence. *International Journal of Mental Health and Addiction*, 8:119–125, 2010. 10.1007/s11469-009-9229-x.

[Gro01] Network Working Group. Traditional ip network address translator (rfc3022), January 2001.

[GYF95] James Griffioen, Rajendra Yavatkar, and Raphael Finkel. Unify: A scalable approach to multicomputer design. *IEEE Computer Society Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7, 1995.

[HBH06] Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 48, New York, NY, USA, 2006. ACM.

[HL04] Shun-Yun Hu and Guan-Ming Liao. Scalable peer-to-peer networked virtual environment. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 129–133, New York, NY, USA, 2004. ACM.

[HSSB09] Florian Heger, Gregor Schiele, Richard SÂ¨uselbeck, and Christian Becker. Towards an interest management scheme for peer-based virtual environments. In *Workshops der Wissenschaftlichen Konferenz Kommunikation in Verteilten Systemen 2009*, 2009.

[Ibr92] Mamdouh H. Ibrahim. Reflection and metalevel architectures in object-oriented programming. *SIGPLAN OOPS Mess.*, 4:315–318, December 1992.

[IEE89] IEEE. Ieee standards for local area networks: Token ring access method and physical layer specifications. *IEEE Std 802.5-1989*, pages 0–1, 1989.

[ITSB10] Laura Itzel, Verena Tuttlies, Gregor Schiele, and Christian Becker. Consistency management for interactive peer-to-peer-based systems. In *SIMUTools '10: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, pages 1–8, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Jul92] Eric Jul. Emerald paradigms for distributed computing. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, EW 5, pages 1–4, New York, NY, USA, 1992. ACM.

[KC08] James Kinicki and Mark Claypool. Traffic analysis of avatars in second life. In *NOSSDAV '08: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 69–74, New York, NY, USA, 2008. ACM.

[KCC+05] Jaecheol Kim, Jaeyoung Choi, Dukhyun Chang, Taekyoung Kwon, Yanghee Choi, and Eungsu Yuk. Traffic characteristics of a massively multi-player online role playing game. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–8, New York, NY, USA, 2005. ACM.

[KEHMKG01] Gregor Kiczales, Jim Hugunin Erik Hilsdale, Jeffrey Palm Mik Kersten, and William G. Griswold. An overview of aspectj. In *ECOOP 2001 - Object-Oriented Programming*, volume 2072/2001, pages 327–354. Springer Berlin / Heidelberg, 2001.

[KR81] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[Kur97] Brian T. Kurotsuchi. The wonders of java object serialization. *Crossroads*, 4:3–8, November 1997.

[KWSW08] Mirko Knoll, Arno Wacker, Gregor Schiele, and Torben Weis. Bootstrapping in peer-to-peer systems. In *14th International Conference on Parallel and Distributed Systems*, 2008.

[LCP+05] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.

[LDS93] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in thor. In *Distributed Object Management*, pages 79–91. Morgan Kaufmann, 1993.

[Lea10] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43:12–14, 2010.

[Li88] Kai Li. Ivy: A shared virtual memory system for parallel computing. In *International Conference on Parallel Processing*, 1988.

[Lin99] P. Lincroft. The internet sucks: Or, what i learned coding x-wing vs. tie fighter. In *Game Developers Conference*, 1999.

[LWJ10] LWJGL. The lightweight java game library http://lwjgl.org, 2010.

[Mel08] Matthias Melzer. *Second Life-Programmierung mit Linden Scripting Language*. Hanser, 2008.

[MF98] Blair MacIntyre and Steven Feiner. A distributed 3d graphics library. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 361–370, New York, NY, USA, 1998. ACM.

[MGpLNS91] Mesaac Makpangou, Yvon Gourhant, Jean pierre Le Narzul, and Marc Shapiro. Structuring distributed applications as fragmented objects, 1991.

[MJLF84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[MMS10] Marc-Florian Müller, Kim-Thomas Möller, and Michael Schöttner. Commit protocols for a distributed transactional memory. In *Proceedings of PDCAT*, 2010.

[MMSS09] Kim-Thomas Möller, Marc-Florian Müller, Michael Sonnenfroh, and Michael Schöttner. A software transactional memory service for grids. In Arrems Hua and Shih-Liang Chang, editors, *Algorithms and Architectures for Parallel Processing*, volume 5574 of *Lecture Notes in Computer Science*, pages 67–78. Springer Berlin / Heidelberg, 2009.

[Mos93] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27:18–26, January 1993.

[MZP+94] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Paul T. Barham, and Steven Zeswitz. Npsnet: A network software architecture for large scale virtual environments, 1994.

[NA08] European Network and Information Security Agency. Security and privacy in massively-multplayer online games and and corporate virtual worlds, 11 2008.

[NLSG03] Martin Naef, Edouard Lamboray, Oliver Staadt, and Markus Gross. The blue-c distributed scene graph. In *In Proceedings of the IPT/EGVE Workshop 2003*, pages 125–133. Press, 2003.

[NWH05] Brian D. Ng and Peter Wiemer-Hastings. Addiction to the internet and online gaming. *CyberPsychology & Behavior*, 8(2):110–113, 2005.

[Pol93] Andreas Polze. The object space approach: Decoupled communication in c. In *In Proc. Technology of Object-Oriented Languages and Systems (TOOLS 93*. Prentice Hall, 1993.

[PW02] L. Pantel and L. C Wolf. On the suitability of dead reckoning schemes for games. In *1st Workshop on Network and System Support For Games*, 2002.

[RA00] Yoshisuke Ueda Ralph Abraham. *The Chaos Avant-Garde: Memoirs of the Early Days of Chaos Theory*. World Scientific Publishing Company, 2000.

[RRCC10] Paolo Romano, Luis Rodrigues, Nuno Carvalho, and Joäo Cachopo. Cloud-tm: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, 44:1–6, April 2010.

[SA05] A. K. Jha S.P. Ahuja, R. Eggen. A performance evaluation of distributed algorithms on shared memory and message passing middleware platforms. *Informatica*, 29(3), 2005.

[SC92] Paul S. Strauss and Rikk Carey. An object-oriented 3d graphics toolkit. *SIGGRAPH Comput. Graph.*, 26(2):341–349, 1992.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Sec] SecondLife. http://wiki.secondlife.com/wiki/open_source_portal.

[Sha07] Nati Shalom. Data-awareness and low-latency on the enterprise grid. Technical report, GigaSpaces Technologies Inc., 2007.

[SKRR03] D.A. Smith, A. Kay, A. Raab, and D.P. Reed. Croquet - a collaboration system architecture. In *Creating, Connecting and Collaborating Through Computing*, 2003.

[SSK+10] Thilo Schmitt, Patrick Schmidt, Nico Kaemmer, Steffen Gerhold, and Peter Schulthess. Adding multiprocessor support to an uniprocessor distributed operating system with transactional distributed memory. *Computer Engineering and Applications, International Conference on*, 1:309–313, 2010.

[SSW+07] Gregor Schiele, Richard Suselbeck, Arno Wacker, Jorg Hahner, Christian Becker, and Torben Weis. Requirements of peer-to-peer-based massively multiplayer online gaming. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 773–782, Washington, DC, USA, 2007. IEEE Computer Society.

[Ste92] Jonathan Steuer. Defining virtual reality: Dimensions determining telepresence. *JOURNAL OF COMMUNICATION*, 42:73–93, 1992.

[STS98] M. Schoettner, S. Traub, and P. Schulthess. A transactional dsm operating system in java. In *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applicattions, Las*, 1998.

[SWC76] John Short, Ederyn Williams, and Bruce Christie. *The Social Psychology of Telecommunications*. John Wiley & Sons Ltd, 1976.

[Tho68] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11:419–422, June 1968.

[Tra99] Henrik Tramberend. Avocado: A distributed virtual reality framework. *Virtual Reality Conference, IEEE*, 0:14, 1999.

[TvS08] Andrew S. Tanenbaum and Maarten van Steen. *Verteilte Systeme*, chapter 8, pages 400–406. Pearson Studium, 2008.

[Vin93] Steve Vinoski. Distributed object computing with corba. *C++ Report magazine*, July/August, 1993.

[Wal08] Jim Waldo. Scaling in games & virtual worlds. *Queue*, 6(7):10–16, 2008.

[Wor10] Wissenheim Worlds. http://www.wissenheim.de, 2010.

[WS07] Steven Daniel Webb and Sieteng Soh. Cheating in networked computer games: a review. In *DIMEA '07: Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts*, pages 105–112, New York, NY, USA, 2007. ACM.

[WVG92] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11(3):201–227, 1992.

[Yee06] Nick Yee. Motivations for play in online games. *Motivations for Play in Online Games*, 9(6):772–775, 2006.

[YV02] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.

Abbildungsverzeichnis

1.1 MUD 1 10
1.2 World of Warcraft 11
1.3 Second Life 13

2.1 Befehlsorientiertes Modell 18
2.2 Aktualisierungsorientiertes Modell 18
2.3 Konzept des Dead-Reckoning 20
2.4 Daten eines Knotens 22
2.5 Area of Interest 25
2.6 Unterteilung in Sichtbereiche 26
2.7 Schematischer Server-Cluster-Aufbau 27
2.8 P2P-Overlay-Struktur 27
2.9 Aufbau mit multiplen Welten 28
2.10 Konsistenz von Positionsinformationen 29
2.11 Kooperation unterschiedlicher Konsistenzdomänen 32

3.1 Probleme bei ungenügender Kapselung 36
3.2 Konzeptioneller Aufbau eines DSM 38
3.3 Sichten und Replikationsschicht 41
3.4 Push-Operation 43
3.5 Invalidate-Operation 44
3.6 Pull-Operation 45
3.7 Sync-Operation 46
3.8 Order-Operation 47
3.9 Hüllenbildung 48
3.10 Globale und lokale Referenzen 49
3.11 Verkettete Push-Operationen 52
3.12 Exemplarische Freispeichersammlung 55
3.13 Knotenlokale Skalierbarkeit der Replikationsschicht 64

4.1 Transformationshierarchie 73
4.2 Vererbungshierarchie des Szenengraphens 74
4.3 Struktur eines Szenengraphens 75
4.4 Beobachterschnittstelle eines Formobjektes 76

4.5 Ereignisbehandlung . . . 77
4.6 Beobachterschnittstelle eines Szenenobjektes . . . 78
4.7 Lokale und globale Daten . . . 78
4.8 Komponenten einer virtuellen Welt mit TGOS . . . 79
4.9 Ereignisverarbeitung . . . 81
4.10 Aufbau eines Zeitobjektes . . . 83
4.11 Hauptschleifen . . . 84
4.12 Atomare Ausführung des Commits . . . 91
4.13 Volleyballspiel in Wissenheim Worlds . . . 96
4.14 Basiselemente des Spiels . . . 96
4.15 Klassen des Volleyballspiels . . . 97
4.16 Szenengraph der Volleyball-Szene . . . 98
4.17 AoI-Management mit Koordinator . . . 104
4.18 AoI-Management ohne Koordinator . . . 106
4.19 Replikationsschichten . . . 108
4.20 Replikationsereignisse am Beispiel des Verbindungsaufbaus . 109
4.21 Kommunikation zwischen Ereignisbehandlung und Replikationsschicht . . . 111
4.22 Dienste der verteilten Welt . . . 112
4.23 Wissenheim Worlds Replikationsschicht . . . 114
4.24 Konzeptioneller Aufbau des RedDwarf-Frameworks . . . 115
4.25 Asynchrone Auswahl der Spielfigur . . . 117

5.1 Das Transformationsobjekt und seine Komponenten . . . 122
5.2 Marshalling des Transformationsobjekt . . . 123
5.3 Replikationsarchitekturformen . . . 124
5.4 Maximaler Durchsatz mit P2P-Replikation . . . 125
5.5 Maximaler Durchsatz mit CS-Replikation . . . 126
5.6 Latenz in Abhängigkeit der Last bei P2P-Replikation . . . 127
5.7 Latenz in Abhängigkeit der Last bei CS-Replikation . . . 127
5.8 Latenz im hierarchischen Fall bei CS-Replikation . . . 128
5.9 Ausführungszeit der TGOS-Operationen ohne Replikationsschicht . . . 129
5.10 Maximale Empfangsrate für TGOS-Push-Operationen . . . 130
5.11 Transaktionsrate in Abhängigkeit zur Anzahl der Knoten . . 131
5.12 Kosten der transaktionalen Operationen . . . 132
5.13 Aggregierter Bandbreitenbedarf mit AoI-Management . . . 134
5.14 Aggregierte Aktualisierungsrate mit AoI-Management . . . 135
5.15 Aggregierter Bandbreitenbedarf pro Szene . . . 135
5.16 Aggregierte Aktualisierungsrate pro Szene . . . 136
5.17 Last bei Szenenwechsel . . . 137
5.18 Bildwiederholrate des transaktionalen Chats bei steigender Latenz . . . 139
5.19 Standardabweichung des transaktionalen Chats . . . 140

5.20 Wissenheim Worlds auf HTC Desire mit Android 2.1. 141
5.21 Wissenheim Worlds auf mobilen Geräten 141
5.22 Ergebnisse des Feldtests 142
5.23 Latenzempfinden der Benutzer 142

Tabellenverzeichnis

3.1 Syntaxauflistung 51
3.2 Liste der Operationen 67

4.1 Zugriffsart der Komponenten auf den Szenengraphen 84
4.2 Liste der Replikationsereignisse 110

Listing

2.1 Linden Scripting Language 21
4.1 Umsetzung der strikten Konsistenz mit TGOS 87
4.2 Umsetzung der schwachen Konsistenz mit TGOS 87
4.3 Definition des TGOS-Tokens 90
4.4 Transaktionierung einer Listenoperation 93
4.5 Spielfigurauswahl beim onClick-Ereignis 99
4.6 Behandlung der Tastaturereignisse 100
4.7 Berechnung des Spiels 101
4.8 RedDwarf ClientChannel 116
5.1 Codebeispiel für die transaktionale Aufwandsmessung 131