Multiplayer Physik – Synchronisierung einer Physiksimulation über Netzwerk

Matthias Bartsch

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Dezember 2012

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 3. Dezember 2012

Matthias Bartsch

Inhaltsverzeichnis

Kurzfassung

Schlechte Netzwerkbedingungen – vor allem hohe Latenzwerte – können die Performance von Echtzeit-Netzwerkapplikationen wie Online Multiplayer Spielen stark beeinträchtigen. Die vorliegende Arbeit befasst sich mit der Frage, wie eine Physiksimulation möglichst flüssig über Netzwerk synchronisiert werden kann.

Zu Beginn wird dem Leser ein Überblick verschafft, worauf bei der Entwicklung von Netzwerkspielen besonders zu achten ist und welche Architekturformen zur Umsetzung verwendet werden können. Des Weiteren werden verschiedene Synchronisierungsverfahren vorgestellt, mit deren Hilfe schlechte Netzwerkverhältnisse ausgeglichen werden können.

Im Rahmen dieser Arbeit wurde eine eigenständige Netzwerkbibliothek in Java, sowie unter deren Verwendung eine 2D-Fahrsimulation basierend auf der Physik-Engine JBox2D entwickelt. Zur Synchronisierung über das Netzwerk wurden einige der vorgestellten Techniken getestet und miteinander verglichen. In einem objektiven Testverfahren wurde versucht, die Vor- und Nachteile der einzelnen Synchronisierungs-Methoden aufzuzeigen und herauszufinden, welche Variante sich für den Einsatz in Simulationen dieser Art am Besten eignet.

Abstract

Bad network conditions — especially high latency values — can result in a very bad performance for networked realtime applications such as online multiplayer games. The present work tries to answer the question how a physics simulation can be synchronized over a network as smoothly as possible.

In the beginning the reader is given an overview of the most important things to consider when developing a networked game as well as the architectural concepts that can be used for the implementation. Also various synchronization techniques are presented, that can help to compensate for bad network conditions.

In the course of this thesis a network library for java, as well as a driving simulation built on top of it that uses the physics engine JBox2D have been developed. For the synchronization over network some of the presented techniques have been tested and evaluated. In an objective testing process the advantages and disadvantages of the different methods have been investigated in an attempt to find the algorithm that leads to the best results.

Kapitel 1

Einleitung

1.1 Motivation

Mit der steigenden Verfügbarkeit von Breitband-Internet über die letzten Jahre hat auch die Beliebtheit von Online Spielen immer weiter zugenommen. Mitte 2011 handelte es sich bereits bei mehr als 90% aller Internetzugänge in OECD[1] Staaten um Breitband-Anschlüsse. Im Jahr 2010 gaben im Durchschnitt 30% aller Erwachsenen in OECD Staaten an, das Internet zum Spielen bzw. zum Herunterladen von Spielen zu verwenden [28].

Darüber hinaus belegen Online Spiele seit 2010 den zweiten Platz auf der Liste der meist genutzten Online Aktivitäten in Amerika. Online Spiele nehmen damit bereits über 10% der Zeit ein, die ein Einwohner der USA im Durchschnitt im Internet verbringt. Einzig die Zeit, die in sozialen Netzen wie Facebook verbracht wird, kann diesen Wert noch übertreffen mit einem Anteil von knapp 23% [27].

Online Modi in Computerspielen werden somit immer gefragter. Die Entwicklung eines Online Multiplayer Spiels gestaltet sich jedoch deutlich komplexer als die Umsetzung eines Singleplayer Spiels.

1.2 Problemstellung

Durch die Eigenschaften des Netzwerkes ergeben sich für Online Spiele eine ganze Reihe von Einschränkungen, die bereits im Entwicklungsprozess besonders berücksichtigt werden müssen. Vor allem wenn die beteiligten Spieler geografisch gesehen weit voneinander entfernt sind, können die hohen Übertragungszeiten der Netzwerkdaten schnell zu einem Problem werden.

Zur Lösung dieser Probleme wurden verschiedene Kompensationstechniken entwickelt, die dem Ausgleich von wechselnden oder schlechten Netzwerkbedingungen dienen sollen. Die Suche nach guter Literatur, wo diese Verfahren beschrieben werden, oder Implementierungsdetails, gestaltet sich

[1] http://de.wikipedia.org/wiki/OECD

jedoch oft als schwierig. Auch die Anzahl an guten Büchern zu diesem Thema hält sich in Grenzen.

1.3 Zielsetzung

Die vorliegende Arbeit soll dem Leser einen Einblick in die Welt der Netzwerkspiele-Programmierung verschaffen. Sie bietet einen Überblick über die wichtigsten Punkte, die bei der Umsetzung eines Online Spiels beachtet werden sollten, sowie eine Reihe verfügbarer Techniken zur Synchronisierung von Netzwerkspielen. Einige dieser bestehenden Ansätze werden am Beispiel einer im Rahmen dieser Arbeit entwickelten 2D Fahrsimulation genauer analysiert und miteinander verglichen. Durch Auswertung objektiver Testläufe soll schließlich die Methode gefunden werden, die für diesen Anwendungsfall die besten Ergebnisse liefert.

1.4 Aufbau der Arbeit

Der Rest der vorliegenden Arbeit ist wie folgt strukturiert.

Kapitel 2 dient als Einführungskapitel und bietet einen Überblick über verschiedene Aspekte der Netzwerk-Programmerung im Allgemeinen, sowie diverse Punkte, die bei der Entwicklung von Netzwerk-Spielen besonders zu beachten sind. Es werden die gängigsten Architekturformen vorgestellt, die dabei Anwendung finden, sowie das Design eines eigenen, speziell an die Anforderungen von Online-Spielen angepassten, Netzwerk-Protokolls erläutert.

In Kapitel 3 werden mehrere bewährte Techniken vorgestellt, die zur Synchronisierung von Netzwerkspielen oder zur effizienten Kodierung und Komprimierung der übertragenen Daten verwendet werden können, bevor Kapitel 4 und 5 auf den praktischen Teil der Arbeit eingehen.

Kapitel 4 beschreibt den Aufbau und die Implementierung, sowie die grundlegende Verwendung der im Rahmen dieser Arbeit entwickelten Netzwerkbibliothek, während Kapitel 5 die darauf aufbauende Test-Applikation präsentiert. An dieser Stelle werden außerdem die Bedingungen des Test-Verfahrens für die ausgewählten Synchronisierungs-Techniken definiert.

In Kapitel 6 folgt schließlich die Auswertung der Ergebnisse aus den Test-Durchgängen sowie der Vergleich der verschiedenen Techniken.

Kapitel 7 bildet den Abschluss und fasst die Ergebnisse der gesamten Arbeit sowie die aus dem Testverfahren gewonnenen Erkenntnisse noch einmal zusammen. Zusätzlich werden mögliche Verbesserungen und Erweiterungen vorgeschlagen.

Kapitel 2

Grundlagen und Überblick

Das folgende Kapitel soll dem Leser einen Überblick verschaffen, worauf bei der Entwicklung eines Netzwerkspiels besonders zu achten ist. Der erste Abschnitt zeigt, welche Einschränkungen sich für Multiplayer Spiele über Netzwerk ergeben und warum es wichtig ist, dies bereits im Design der Anwendung zu berücksichtigen. Danach folgt ein Vergleich der verschiedenen Architektur-Modelle, welche zur Umsetzung von Netzwerkspielen verwendet werden können, sowie eine kurze Einführung in das Design eines geeigneten Netzwerkprotokolls.

2.1 Netzwerk-Ressourcen

Wie in [41] beschrieben ergeben sich für Netzwerkspiele drei Einschränkungen, die bereits im Design berücksichtigt werden müssen:

> „Distributed simulations face three resource limitations: network bandwidth, network latency, and host processing power for handling the network traffic. These resources refer to the technical attributes of the underlying network and they impose physical restrictions, which the system cannot overcome and which must be considered in the design."

2.1.1 Bandbreite

Die verfügbare Bandbreite bestimmt die maximale Menge an Daten, die pro Zeiteinheit über ein Netzwerk übertragen werden kann. Je öfter und je größere Nachrichten verschickt werden, umso mehr Bandbreite ist dazu erforderlich. Die Anforderung steigt außerdem mit der Anzahl der Spieler und ist zudem von der Übertragungsmethode abhängig. Durch Verwendung von *Multicast*[1] kann das Versenden redundanter Daten vermieden und damit die

[1] http://tools.ietf.org/html/rfc1112

benötigte Bandbreite verringert werden. Es handelt sich dabei um eine Kommunikation zwischen einem einzelnen Sender und mehreren Empfängern.

In den meisten Fällen ist die Bandbreite bei der Entwicklung eines Netzwerkspiels jedoch nur zweitrangig zu beachten. Wie in [10] beschrieben stellt die Latenz das größere Problem dar:

> „The majority of Internet connections has sufficient bandwidth available to play on the Internet as today's games require only a small total amount of bandwidth, e.g. approx. 34kbit/s for Counter Strike or 5kbit/s for Warcraft III. Hence, the main problems for games lie with network latency, jitter, and packet loss."

2.1.2 Latenz

Die Netzwerklatenz (auch Ping oder Round-Trip-Time, kurz RTT) zeigt an, welche Verzögerung beim Versenden einer Nachricht über das Netzwerk auftritt und wird häufig als kritischster Aspekt bei der Programmierung eines Netzwerkspiels genannt, da bereits geringe Verzögerungen fatale Folgen für das Gameplay haben können.

Die Auswirkungen von Latenz und Paketverlust auf Netzwerkspiele wurden bereits mehrfach untersucht. Die Ergebnisse sind zum Teil jedoch unterschiedlich, da diese vor allem auch von der Art des Spiels abhängig sind. Wie in [10] erklärt wird, erfordern temporeiche Actionspiele wie First-Person Shooter (FPS) oder Sportspiele die geringsten Latenzwerte, da der Spieler dabei die direkte Kontrolle über seinen Avatar hat. Dadurch machen sich bereits kurze Verzögerungen zwischen seinen Aktionen und der Reaktion seines Avatars sehr stark bemerkbar. Im Gegensatz dazu haben Echtzeit-Strategie Spiele (RTS) weniger strenge Anforderungen an die Netzwerklatenz. Hier sind höhere Ping-Werte akzeptabel, da die Spieler ihre Einheiten nur indirekt steuern.

Die Autoren von [38] haben zum Beispiel festgestellt, dass Latenzwerte von bis zu mehreren Sekunden in Warcraft III (einem RTS Spiel) nur minimale Auswirkungen auf die Gesamt-Performance der Spieler hatte.

Experimente mit Unreal Tournament 2003 [4,36] haben hingegen gezeigt, dass FPS in dieser Hinsicht weniger tolerant sind. Die Untersuchungen in [4] haben ergeben, dass die Spieler in der Lage waren, bereits Latenzen von 75 ms zu bemerken. Ab Werten von mehr als 100 ms war darüber hinaus eine deutlich geringere Genauigkeit und Treffsicherheit erkennbar. Die Autoren von [36] geben sogar an, dass schon Latenzwerte ab 60 ms als störend empfunden wurden.

Die Autoren von [30] kamen bei ihrer Analyse einer Rennsimulation zu einem ähnliches Ergebnis. Erste Auswirkungen auf das Spielergebnis ergaben sich bei Latenzwerten von 50 ms. Werte über 100 ms sollten laut den Autoren bei Rennspielen vermieden werden, da dies ein unrealistisches Fahrverhalten

zur Folge hat.

Quellen von Latenz

In [3, Kapitel 5.2] werden drei Hauptquellen für Latenz genannt:

- *Übertragungszeit*: Jede Nachricht, die über ein Netzwerk veschickt wird, benötigt eine gewisse Zeit um vom Sender zum Empfänger zu gelangen, da die Übermittlung im besten Fall mit Lichtgeschwindigkeit erfolgen kann. Die Übertragungszeit steigt somit außerdem mit der geografischen Entfernung zwischen Sender und Empfänger - je größer die Distanz, desto mehr Zeit wird für das Versenden der Daten benötigt.
- *Serialisierung*: Auch das Schreiben und Einlesen von Informationen an bzw. von einem Netzwerk nimmt eine bestimmte Zeit in Anspruch - abhängig von der Datenmenge sowie der Geschwindigkeit der Verbindung.
- *Queueing*: Router können eine zusätzliche Quelle von Netzwerklatenz darstellen, da bei Überlastung einer Verbindung Netzwerkpakete in einer Queue zwischengespeichert werden. Dies kann Verzögerungen von mehreren Sekunden zur Folge haben.

Latenzminimierung

Zur Reduktion der Netzwerklatenz gibt es nur wenige Möglichkeiten, da auf die meisten Faktoren kein Einfluss genommen werden kann. So kann etwa nur ein Upgrade des physikalischen Netzwerks zu einer Verbesserung der Übertragungsgeschwindigkeit und damit zu einer geringeren Übertragungszeit führen. Auch die Zeit, die zur Serialisierung der Daten benötigt wird, hängt zum Großteil von der Geschwindigkeit der Netzwerkverbindung und vom verwendeten Netzwerkprotokoll ab. Der einzige Weg zur zusätzlichen Optimierung der Verzögerungen, die sich bei der Serialisierung und Übertragung der Daten ergibt, besteht darin, so wenige Daten wie möglich zu versenden. Um Queueing und die daraus resultierenden Verzögerungen zu vermeiden, muss zudem von Vornherein eine Überlastung des Netzwerkes verhindert werden (siehe Abschnitt 2.3.4).

Im Endeffekt kann Latenz aber nie komplett eliminiert werden, vor allem bei großen Entfernungen zwischen den beteiligten Spielern. Wie bereits zu Beginn dieses Abschnittes erwähnt, ist es demnach erforderlich, schon das Spiel Design für Netzwerkspiele so anzupassen, dass Latenzwerte bis zu einem gewissen Grad vor den Spielern verborgen werden können. Zu diesem Zweck wurden mehrere Techniken entwickelt, wovon die bekanntesten in Kapitel 3 vorgestellt werden.

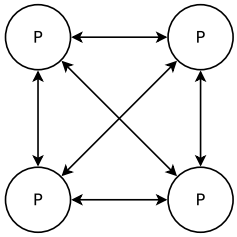


Abbildung 2.1: Peer-to-Peer Modell

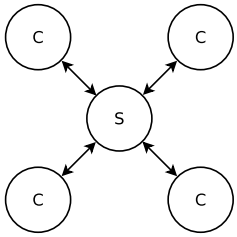


Abbildung 2.2: Client/-Server Modell

2.1.3 Rechenleistung

Auch die Rechenleistung des Computers, auf dem eine verteilte Simulation läuft, ist eine begrenzte Ressource, die leicht übersehen werden kann [41]. Vor allem bei Client/Server Systemen muss sichergestellt sein, dass der Server über genügend Leistung verfügt. Dies kann speziell bei *Massively Multiplayer Online Spielen*(MMOGs) zu einem Problem werden, wo sehr viele Spieler zur selben Zeit an einer Simulation teilnehmen [21].

2.2 Architektur-Konzepte

Die Kommunikation zwischen den beteiligten Spielern eines Netzwerkspiels kann auf Grund verschiedener Architekturmodelle realisiert werden. Die beiden bekanntesten Systeme sind Peer-to-Peer und Client/Server (siehe Abb. 2.1 und 2.2). Je nach Art des Spiels gibt es jedoch auch verschiedene Mischformen, die die Vorteile beider Konzepte miteinander verbinden.

2.2.1 Peer-to-Peer

In einer Peer-to-Peer Architektur sind alle beteiligten Netzwerkknoten gleichgestellt und direkt miteinander verbunden (siehe Abb. 2.1). Alle Nachrichten werden dabei jeweils an alle anderen Teilnehmer verteilt.

Da die Spieler State Updates direkt miteinander austauschen, kann somit die Schwachstelle eines zentralen Servers eliminiert werden. Wie in [42] beschrieben, gibt es keinen so genannten *„Single Point of Failure“*. Das heißt, wenn ein Client die Verbindung verliert, kann das Spiel trotzdem fortgesetzt werden, sogar dann, wenn es sich dabei um den Host handelt. Ein weiterer wesentlicher Vorteil ist die geringere Latenz zwischen den Clients dank des direkten Nachrichtenaustauschs ohne den Umweg über einen zentralen Server.

Durch die direkte Verbindung aller Spieler untereinander steigt jedoch auch die Anforderung an die Bandbreite bei den Clients. Probleme dieser Architektur umfassen außerdem Konsistenz der Game States, Eventreihen-

folge und Cheating. Das Verhindern von Cheating ist besonders schwierig, da die Peers typischerweise die Kontrolle über den eigenen Game State sowie die Verteilung von Ereignissen haben [21].

2.2.2 Client/Server

Bei Client-Server Systemen besteht keine direkte Verbindung mehr zwischen den Clients, sämtliche Kommunikation geht dabei über einen zentralen Server (siehe Abb. 2.2). Der Hauptvorteil für Spiele liegt darin, dass der Server die Kontrolle über den gesamten Game State behält, wodurch es bedeutend einfacher ist, alle Clients synchron zu halten und Cheating zu verhindern [21].

Diese Architektur hat sich besonders bei Actionspielen, die über das Internet gespielt werden, bewährt. Spiele wie Half-Life bedienen sich eines solchen Kommunikationsmodells, genau so wie Spiele basierend auf der Quake 3 Engine oder der Unreal Tournament Engine [5].

Wie Yahn Bernier in [5] erklärt, ist bei diesen Spielen der Server für die Berechnung der Spiellogik zuständig. Auf der Clientseite werden lediglich die Benutzereingaben eingelesen und an den Server geschickt, wo die Input-Kommandos verarbeitet werden. Auf diese Weise wird der neue Game State am Server berrechnet und eine Liste von Objekten zurück an die Clients geschickt, wo diese dargestellt werden.

Diese Kernarchitektur ist zwar sehr simpel, bringt jedoch auch ein großes Problem mit sich. Da die Clients bei allen Benutzereingaben erst auf die Antwort des Servers warten müssen, um das Ergebnis darstellen zu können, ist dieses Kommunikationsmodell besonders sensibel, was die Netzwerklatenz betrifft. Aus diesem Grund sind gerade Client/Server Spiele auf Techniken wie Client Side Prediction und Lag Compensation angewiesen, um diese Verzögerungen auszugleichen (siehe Kapitel 3).

2.2.3 Hybrid-Modelle

Trotz der höheren Latenz, dem Single Point of Failure, sowie Scalability Problemen ist das Client/Server Modell die mit Abstand am häufigsten verwendete Architektur für Multiplayerspiele [8]. Aus diesem Grund wurden verschiedene Misch-Architekturen entwickelt, die versuchen, diese Probleme so gut wie möglich zu beseitigen.

In einer auf dem Client/Server Modell basierenden *Hybrid* Architektur kann es den Clients zusätzlich ermöglicht werden, direkt mit anderen Clients zu kommunizieren, wie bei einer gewöhnlichen Peer-to-Peer Architektur(siehe Abb. 2.3). Die Autoren von [21] haben eine solche Hybrid-Architektur entwickelt, die sich besonders gut für den Einsatz in MMOGs eignet. Dabei können einzelne Spieler als lokale Server bestimmt werden, zu denen sich andere Spieler dann wie bei Peer-to-Peer direkt verbinden können. Während

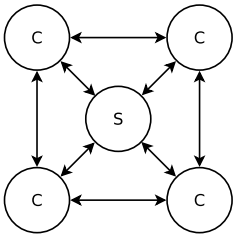


Abbildung 2.3: Client-Server/Peer-to-Peer Hybrid Architektur

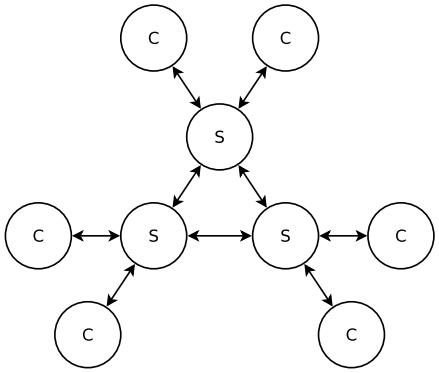


Abbildung 2.4: Server-Network Architektur

kritische Events, die den Game State verändern, nach wie vor von dem zentralen Server behandelt werden, werden Positionsveränderungen nun direkt über die lokalen Server verteilt. Auf diese Weise kann die Server-Bandbreite maßgeblich gesenkt werden, die zentrale Verwaltung des Game States bleibt dabei aber weiterhin erhalten.

Trotzdem stellt in Client/Server Systemen der Server oft einen Performance Engpass sowie einen Single Point of Failure dar. Zur Lösung dieses Problems kann der einzelne Server durch ein Netzwerk von Servern ersetzt werden. Diese Architektur nennt sich *Server-Network* [41] und wird in [8] auch als *Mirrored Game Servers* Architektur beschrieben. Dabei gibt es nun mehrere Server, die untereinander über ein Peer-to-Peer Netzwerk verbunden sind(siehe Abb. 2.4). Die Clients verbinden sich mit dem Server, zu dem sie am wenigsten weit entfernt sind, wie bei einer herkömmlichen Client/Server Architektur. Bei guter Verteilung der lokalen Server kann der zusätzliche Latenz-Overhead, den Client/Server Modelle mit sich bringen, deutlich reduziert werden. Darüber hinaus kann so die Anforderung an die Bandbreite und Rechenleistung der einzelnen Server verringert werden. Dies verbessert die Scalability der Game Architektur, erhöht jedoch auch die Komplexität der Netzwerkkommunikation.

2.3 Netzwerkprotokoll-Design

Neben der Wahl der Kommunikations-Architektur ist auch die Frage, welches Netzwerkprotokoll zum Versenden und Empfangen der Datenpakete verwendet werden soll, eine der wichtigsten Entscheidungen bei der Entwicklung einer jeden Netzwerkapplikation. Die beiden bekanntesten Netzwerkprotokolle, die dabei verwendet werden, sind UDP und TCP. Da diese sehr unterschiedlich sind und verschiedene Vor- und Nachteile haben, hängt die Wahl

hauptsächlich von der Art der entwickelten Applikation ab.

2.3.1 TCP vs. UDP

Transmission Control Protocol

TCP, kurz für Transmission Control Protocol [35], ist ein zuverlässiges, verbindungsorientiertes Transportprotokoll. Das bedeutet, es muss zuerst eine Verbindung hergestellt werden, bevor Daten zwischen zwei Hosts ausgetauscht werden können. Darüber hinaus stellt TCP sicher, dass sämtliche gesendeten Daten auch wirklich auf der Empfägerseite ankommen, und zwar in der selben Reihenfolge, in der die Pakete versendet wurden. Damit die Abarbeitung aller Daten in der korrekten Reihenfolge garantiert werden kann, werden alle eingehenden Pakete in einer Queue gespeichert. Geht eines der gesendeten Pakete unterwegs verloren, werden die fehlenden Daten erneut gesendet, während die Empfängerseite mit der Verarbeitung aller weiteren Pakete so lange wartet, bis das verloren gegangene Paket nachgereicht wurde. Auf diese Weise wird zwar eine zuverlässige Übertragung aller Daten erreicht, was allerdings auch eine erhöhte Verzögerung zur Folge hat, vor allem bei jedem Paketverlust.

User Datagram Protocol

Im Gegensatz zu TCP handelt es sich bei UDP [33] um ein zustandsloses Protokoll, dabei besteht keine direkte Verbindung zwischen zwei Hosts. Desweiteren ist UDP nicht zuverlässig, da verlorene Pakete nicht neu übermittelt werden. Die Datenübertragung erfolgt jedoch schneller als bei TCP, da keine Fehlerbehandlung durchgeführt wird und es nicht wie bei TCP zu hohen Wartezeiten im Fall von Paketverlusten kommt. Aus diesem Grund eignet sich UDP sehr gut für Applikationen, wo eine möglichst schnelle Übertragung von Daten benötigt wird. Dazu gehören auch alle Netzwerkspiele, bei denen eine Darstellung in Echtzeit erforderlich ist.

Fazit

Wie unter Abschnitt 2.1.2 bereits erläutert, stellen die meisten Multiplayer Spiele sehr hohe Ansprüche an die Geschwindigkeit der Netzwerkverbindung. Andere Faktoren wie zum Beispiel Zuverlässigkeit spielen oft nur eine untergeordnete Rolle. Für Echtzeitdaten wie etwa State Updates wird stets nur die aktuellste Version benötigt. Verlorene Daten neu zu übermitteln ist in diesem Fall nicht sinnvoll, da diese bis dahin bereits veraltet und unbrauchbar sind [13].

Aus diesen Gründen verwenden die meisten Computerspiele UDP für die Implementierung der Netzwerkschicht [7], da es sich auf Grund seiner Eigenschaften am Besten dafür eignet. Die Wahl hängt zum Teil aber auch

von der Art des Spiels ab. Warcraft III verwendet zum Beispiel ausschließlich TCP als Transportprotokoll [38]. Bei Spielen mit weniger hohen Ansprüchen an die Übertragungsgeschwindigkeit wie RTS Spielen (vergleiche Abschnitt 2.1.2) kann also auch die Performance von TCP ausreichend sein. Da für die meisten Echtzeitspiele jedoch eine möglichst geringe Latenz erforderlich ist, sollte in diesen Fällen besser UDP zur Übertragung der Netzwerkdaten eingesetzt werden.

Folgen bei gleichzeitiger Verwendung beider Protokolle

Auch wenn es möglich ist, UDP und TCP parallel zu verwenden, empfiehlt es sich für alle Spiele, die auf eine möglichst schnelle Datenübertragung angewiesen sind, auf die Verwendung von TCP komplett zu verzichten. Da beide Protokolle auf das Internet Protokoll (IP [9, 34]) aufsetzen, beeinflussen sie sich gegenseitig. Dies kann zu einer Reihe negativer Effekte wie erhöhtem Paketverlust führen [37]. Brian Hook, Mitentwickler von Quake, schreibt auf seiner Webseite [18] „Just use a reliable system over UDP like ENet[2] and ditch TCP altogether. Really.“.

Das Verwalten von Verbindungen wird dabei der Netzwerkbibliothek überlassen. Desweiteren kann durch die Definition eines eigenen Netzwerkprotokolls auch unter Verwendung von UDP zusätzliche Funktionalität wie eine zuverlässige Datenübermittlung umgesetzt werden (siehe Abschnitt 2.3.3).

2.3.2 Virtuelle Verbindungen über UDP

Da es sich bei UDP um ein zustandsloses Protokoll handelt, kennt es auch kein Konzept einer Verbindung zwischen zwei Computern. Oft ist solch ein Verhalten aber erwünscht, da somit zum Beispiel die Verwaltung aller Spieler deutlich vereinfacht wird. Im Falle einer Echtzeitanwendung wie einem Netzwerkspiel, wo kontinuierlich Daten zwischen beiden Seiten ausgetauscht werden, kann eine Verbindung am einfachsten als ein bestehender Fluss von Daten beschrieben werden. Das heißt, so lange Datenpakete gesendet und empfangen werden, besteht eine Verbindung. Werden von einer Seite über einen bestimmten Zeitraum keine Daten mehr empfangen, gilt die Verbindung als getrennt. Man spricht in diesem Fall von einem *Timeout* der Verbindung [14].

Festlegung eines eigenen Netzwerkprotokolls

Zusätzlich zu dem eben beschriebenen *Timeout*-Mechanismus kann zum Aufbauen einer Verbindung eine bestimmte „Connect“ Nachricht an die andere Seite gesendet werden, bzw. eine „Disconnect“ Nachricht zum Trennen von Verbindungen festgelegt werden. Um zwischen den verschiedenen Arten von

[2] http://enet.bespin.org/

Paketen unterscheiden zu können, wird an jedes Datenpaket ein eigener *Paketheader*[3] mit einer Paket-ID angehängt.

Anhand dieser ID kann die Empfängerseite erkennen, ob es sich um eine Nachricht wie Connect bzw. Disconnect handelt oder um ein Datenpaket, welches weitere Informationen enthält. Im zweiten Fall werden die Headerdaten von der Netzwerkschicht entfernt und der eigentliche Inhalt des Pakets wird an die Applikationsschicht weitergerreicht.

2.3.3 Zuverlässigkeit

Auch bei Netzwerkspielen ist es in vielen Fällen erdorderlich, dass alle beteiligten Spieler über das Auftreten eines bestimmten Ereignisses informiert werden. Für diese Daten ist somit eine zuverlässige Übertragung erforderlich. Da es sich bei UDP aber um ein unzuverlässiges Netzwerkprotokoll handelt, muss in irgendeiner Weise sicher gestellt werden, dass die entsprechenden Daten nicht verloren gehen.

Wie unter Abschnitt 2.3.1 erklärt wurde, ist davon abzuraten, TCP und UDP parallel zu verwenden. Andere Protokolle wie RUDP[4] setzen auf UDP auf und erweitern dieses um zusätzliche Eigenschaften wie zuverlässige und geordnete Datenübertragung für virtuelle Verbindungen. Es handelt sich dabei jedoch um keine Standards und es gibt keine einheitliche Implementierung.

Eine weitere Möglichkeit ist die Verwendung eines eigenen Netzwerkprotokolls zum Erkennen von Paketverlusten. Diese Lösung eignet sich besonders gut für den Einsatz in Computerspielen, da in diesem Fall die Applikationsschicht selbst die Kontrolle darüber hat, wie mit verlorenen Netzwerkpaketen umgegangen werden soll. So kann zwischen den verschiedenen Arten von Daten unterschieden werden, so dass stets nur die Informationen neu übermittelt werden, die auf keinen Fall verloren gehen dürfen [12].

Sequenznummern und Acks

Um auch unter Verwendung von UDP ein zuverlässiges Versenden von Daten erreichen zu können, wird ein Mechanismus zum Erkennen von verloren gegangenen Netzwerkpaketen benötigt. Damit die gesendeten Pakete identifiziert werden können, werden sie mit einer so genannten *Sequenznummer* versehen, die im Paketheader angefügt werden kann. Es handelt sich dabei einfach um eine fortlaufende Nummerierung aller Pakete.

Anhand der Sequenznummern kann die Senderseite darüber benachrichtigt werden, welche Pakete beim Empfäger angekommen sind, indem stets die id des zuletzt empfangenen Pakets an die andere Seite zurück geschickt wird. Auch diese so genannte *Ack*-Nummer (von engl. *Acknowledgement*)

[3] http://de.wikipedia.org/wiki/Header
[4] http://tools.ietf.org/html/draft-ietf-sigtran-reliable-udp-00

kann an den Paketheader angehängt werden. Ein Ack pro Paket ist für eine zuverlässige Rückmeldung jedoch unzureichend, da auch das Antwortpaket verloren gehen könnte. Dieses Problem wird gelöst, indem mit jedem Paket mehrere Bestätigungen mitgeschickt werden. Um diese platzsparend zu speichern, können alle weiteren Acks in einem Bitfeld gespeichert werden. Ist das n-te bit in diesem Bitfeld gesetzt, bedeutet dies, dass der Erhalt des Pakets mit der Sequenznummer *ack*-n bestätigt werden soll.

Erkennen von verlorenen Paketen

Wie bei dem Timeout Mechanismus virtueller Verbindungen wird auch zur Überprüfung, ob ein Paket unterwegs verloren gegangen ist, ein bestimmter Grenzwert festgelegt. Wenn nach Ablauf dieser Zeit ab Versenden des Pakets noch immer keine Empfangsbestätigung erhalten wurde, wird das Paket als verloren angenommen und die Daten können bei Bedarf erneut gesendet werden.

2.3.4 Staukontrolle

Die maximale Datenmenge, die über ein Netzwerk pro Zeiteinheit übertragen werden kann, wird als Durchsatz bezeichnet. Werden mehr Daten gesendet, als von der Verbindung verarbeitet werden können, kommt es zu einer Überlastung des Netzwerks. Die Folge ist ein Datenstau, was zu erhöhtem Paketverlust und starker Latenz führen kann. Um dies zu verhindern, verwendet TCP das Konzept der Staukontrolle bzw. Überlaststeuerung (engl. *Congestion Control* [2]). Dabei handelt es sich um mehrere Algorithmen, die zur Kontrolle des Datenflusses verwendet werden, um eine Überlastung der Netzwerkverbindung zu vermeiden.

UDP verzichtet auf diese Funktionalität, da versucht wird, alle Daten so schnell wie möglich zu übertragen. Bei Verwendung von UDP sollte daher immer darauf geachtet werden, dass die Verbindung nicht zu stark belastet wird. Bei einem Netzwerkspiel empfielt es sich, die Frequenz, mit der Updates verschickt werden, dynamisch zu steuern. Das heißt, bei guten Netzwerkverhältnissen können häufiger Updates gesendet werden als bei geringer verfügbarer Bandbreite. Als Maß hierfür kann entweder die *Verlustrate* oder die *Paketumlaufzeit*[5] verwendet werden. Wenn einer dieser beiden Werte einen festgelegten Schwellenwert übersteigt, kann dies ein Hinweis auf eine Überlastung sein und die Senderate sollte verringert werden [12].

[5] http://de.wikipedia.org/wiki/Paketumlaufzeit

Kapitel 3

Verfahren zur Synchronisierung von Netzwerkspielen

Kapitel 2 soll dem Leser einen groben Überblick über die Einschränkungen verschaffen, die sich bei der Programmierung eines Netzwerkspiels ergeben, sowie über die verschiedenen Architekturen, die zur Umsetzung eines solchen Spiels verwendet werden können. Die Wahl der richtigen Architektur und des Netzwerkprotokolls ist essentiell und hängt auch von der Art des Spiels ab. Im Normalfall ist dies jedoch nicht ausreichend, um die unter Abschnitt 2.1 angeführten Einschränkungen auszugleichen. Das größte Problem ist dabei vor allem die Netzwerklatenz. Aus diesem Grund wurden die in diesem Kapitel vorgestellten Techniken entwickelt, um etwa die Menge an Daten, die zwischen den Spielern ausgetauscht werden muss, so weit wie möglich zu reduzieren oder die Auswirkungen von Netzwerklatenz auf das Gameplay abzuschwächen bzw. im besten Fall ganz vor den Spielern zu verbergen.

3.1 Prediction

Wie unter Abschnitt 2.2.2 erwähnt, stellt die Netzwerklatenz für Client/Server Spiele das größte Problem bei der Kommunikation zwischen den Spielern dar. Bei den ersten First Person Shootern wie *Doom* und *Quake* von id Software konnte man diese Verzögerung als Spieler noch deutlich spüren, da sämtliche Aktionen erst ausgeführt wurden, nachdem die Benutzereingaben an den Server gesendet und von diesem akzeptiert worden sind [15].

Zur Lösung dieses Problems kann es dem Client mit Hilfe von *Prediction* ermöglicht werden, die Antwort des Servers bereits im Voraus zu „erahnen“ und so auf Benutzereingaben direkt zu reagieren. In [3, Kapitel 6.2] wird zudem zwischen *Player Prediction* und *Opponent Prediction* unterschieden.

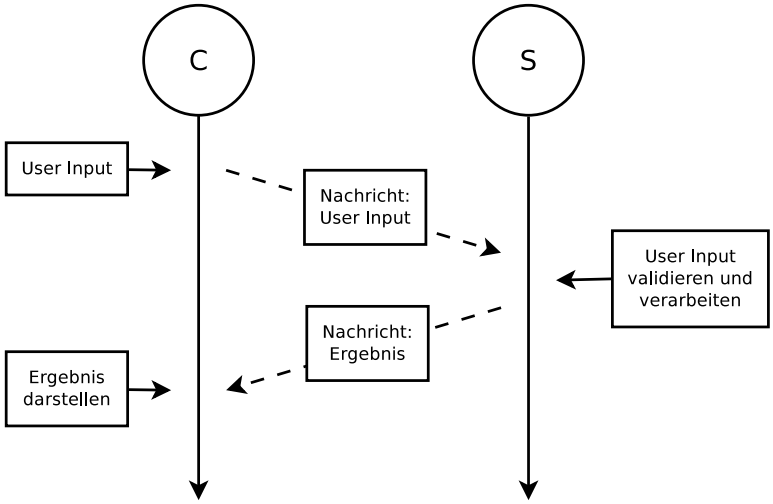


Abbildung 3.1: Client/Server Architektur ohne Player Prediction: das Ergebnis der Benutzereingaben wird erst dargestellt, wenn der Client die Antwort des Servers erhält. Abb. 6.1 aus [3] nachempfunden.

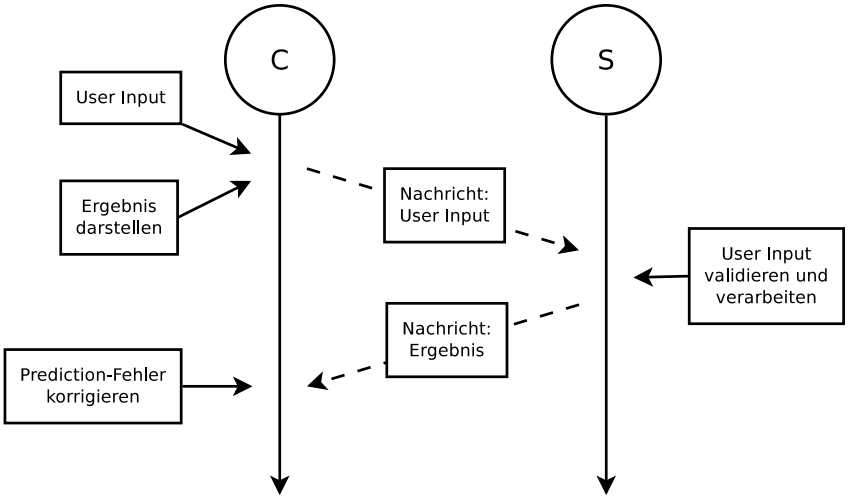


Abbildung 3.2: Durch Player Prediction kann es dem Client ermöglicht werden, sämtliche Benutzereingaben sofort auszuführen, ohne erst auf die Antwort des Servers warten zu müssen. Abb. 6.4 aus [3] nachempfunden.

3.1.1 Player Prediction

Bei der in [3] als Player Prediction beschriebenen Technik handelt es sich um die von John Carmack(Mitbegründer und Lead Programmer von id Softwa-

re[1]) für QuakeWorld[2] entwickelte *Client-Side Prediction* [15].

Mit dieser Methode ist der Client nicht mehr nur für das Einlesen und Versenden von User Input an den Server zuständig, sondern führt sämtliche Kommandos auch sofort lokal aus. Dafür verwendet der Client exakt den selben Code für die Spiellogik wie auf dem Server. Auf diese Weise kann der Client das Ergebnis der Server-Berechnungen selbst vorhersagen und so völlig verzögerungsfrei auf Benutzereingaben reagieren (siehe Abb. 3.1 und 3.2).

Wie Yahn Bernier in [5] erklärt, bedeutet dies jedoch nicht, dass der Client die Kontrolle über seine eigenen Bewegungen hat, wie dies bei einer Peer-to-Peer Architektur der Fall wäre. Der Server behält nach wie vor die Autorität über den gesamten Game State, also auch über die Bewegungen aller Spieler. Die Schwierigkeit bei dieser Technik liegt somit nicht in der Prediction selbst, sondern in der Korrektur des vorrausberechneten Game States der Clients, wenn dieser vom Ergebnis des Servers abweicht [15].

Da die Clients meist nicht über alle Informationen verfügen, auf die der Server Zugriff hat, kann es leicht zu Fehlern bei der Prediction kommen. Vor allem Kollisionen mit anderen Spielern können auf diese Weise nicht vorhergesagt werden. In diesen Fällen müssen sämtliche Abweichungen mit dem Ergebnis des Servers korrigiert werden, sobald der Client eine Antwort erhält, die nicht mit dem voraus berechneten State übereinstimmt.

Mit Hilfe von Player Prediction kann die gefühlte Latenz für alle Bewegungen des eigenen Characters somit praktisch vollständig eliminiert werden, die Korrektur von Prediction-Fehlern kann allerdings ebenso negative Auswirkungen auf das Gameplay haben, da dies zu klar erkennbaren Sprüngen in der Position des Spielers führen kann. In Spielen wie FPS mit einer statischen Spielwelt, wo jedes Objekt von jeweils genau einem Spieler kontrolliert wird, kommt dies nur dann vor, wenn ein Spieler etwa von einer Rakete getroffen wird, oder mit einem Gegner zusammen stößt [11]. Diese Technik eignet sich somit ideal für den Einsatz in FPS Spielen.

3.1.2 Opponent Prediction / Dead Reckoning

Als Opponent Prediction beschreiben die Autoren von [3] die näherungsweise Ortsbestimmung für Objekte, die von einem anderen Spieler (oder Computer) gesteuert werden. Diese Technik wird auch als *Dead Reckoning* (auf Deutsch „Koppelnavigation“[3]) bezeichnet. Damit kann die Position eines bewegten Objektes auf Grund seiner Geschwindigkeit und Bewegungsrichtung sowie der vergangenen Zeit seit der letzten bekannten Position berechnet werden.

Das selbe Prinzip kann in Netzwerkapplikationen verwendet werden, um

[1] http://de.wikipedia.org/wiki/John_Carmack
[2] http://en.wikipedia.org/wiki/QuakeWorld
[3] http://de.wikipedia.org/wiki/Koppelnavigation

die Anforderungen an die Bandbreite zu verringern und Netzwerklatenzen auszugleichen. Dabei werden Update Nachrichten weniger häufig gesendet und die States dazwischen mit Hilfe eines Prediction Algorithmus angenähert [41]. Dieser Ansatz wurde erstmals Mitte der 80er Jahre im Rahmen des von der US Armee entwickelten SIMNET Projekts[4] verwendet und 1993 als Teil des *Distributed Interactive Simulation* Protokolls (DIS, [20]) standardisiert [29].

Funktionsweise

Auf Grund der Limitierung durch die Bandbreite können State Updates in Netzwerkanwendungen üblicherweise nicht mit derselben Frequenz wie die Framerate übertragen werden. Darüber hinaus sind die Updates, bis sie bei den anderen Clients angekommen sind, bereits veraltet, bedingt durch die Netzwerklatenz. Dead Reckoning versucht beide Probleme durch Extrapolation zu lösen. Dazu kann einer der folgenden drei Algorithmen verwendet werden:

$$x(t_1) = x(t_0), \tag{3.1}$$

$$x(t_1) = x(t_0) + v \cdot (t_1 - t_0), \tag{3.2}$$

$$x(t_1) = x(t_0) + v \cdot (t_1 - t_0) + (1/2)a \cdot (t_1 - t_0)^2. \tag{3.3}$$

Dabei steht $x(t)$ für die Position eines Objektes zum Zeitpunkt t während das letzte Positions-Update für das Objekt zum Zeitpunkt t_0 erhalten wurde. Soll nun die Position des Objekts zum Zeitpunkt t_1 ermittelt werden, so kann diese mit Hilfe einer der angeführten physikalischen Gleichungen berechnet werden.

Im einfachsten Fall (Gleichung 3.1) könnte der Client annehmen, dass sich das Objekt zum Zeitpunkt t_1 noch genau auf der selben Position befindet wie zum Zeitpunt t_0. Dies wird in [6] als *Point-to-Point* Methode bezeichnet, da das betroffene Objekt dadurch immer nur dann an einen neuen Punkt bewegt wird, wenn ein neues Positions-Update empfangen wird. Mit dieser Methode kann somit kaum eine flüssige Bewegung erzielt werden, sie eignet sich daher nicht für den Einsatz in Computerspielen.

Geht man stattdessen davon aus, dass sich das Objekt mit einer konstanten Geschwindigkeit (v) fortbewegt, kann Gleichung 3.2 zur Positionsbestimmung verwendet werden. Um die Genauigkeit der Berechnung zu erhöhen, kann wie in Gleichung 3.3 zusätzlich die Beschleunigung (a) miteinbezogen werden. Wie in [40, Kapitel 9.3] beschrieben, kann mit der letzteren Methode relativ genau die Bewegung von Fahrzeugen modelliert werden, während sich Variante 3.2 besser für Objekte eignet, deren Bewegungen weniger vorhersehbar sind, wie zum Beispiel menschliche Charactere.

[4] http://en.wikipedia.org/wiki/SIMNET

Verwendung in Spielen

Wie in [22] beschrieben, ist Dead Reckoning eine im Game Design sehr weit verbreitete Technik, die in bekannten Titeln wie Half-Life, QuakeWorld und Quake3 eingesetzt wird. Die Autoren von [29] haben mehrere verschiedene Prediction Varianten miteinander verglichen und kamen zu dem Ergebnis, dass die in DIS definierte Methode auch für FPS Spiele am besten geeignet ist, um präzise Positionen zu erhalten. Die Autoren von [31] stellten bei ihren Untersuchungen fest, dass vor allem physikalische Simulationen wie Autorennspiele gute Voraussetzungen für den Einsatz von Prediction Methoden bieten. Dies liegt daran, dass Spiele in diesem Genre einem physikalischen Bewegungsmodell folgen. Die Steuerung von Autos erfolgt zum Beispiel durch Beschleunigen und Abbremsen, im Gegensatz zu FPS Spielen, wo die Geschwindigkeit des eigenen Characters meist direkt beeinflusst werden kann, wodurch viel abruptere Bewegungen möglich sind.

Genauigkeit und Fehlerkorrektur

Wie bei der unter Abschnitt 3.1.1 beschriebenen Player Prediction kann es auch bei Dead Reckoning zu Prediction Fehlern kommen. Bei jeder neuen Update Nachricht, die ein Client erhält, ist es sogar sehr wahrscheinlich, dass der anhand von Dead Reckoning voraus berechnete State eines Objektes von den gerade empfangenen Informationen abweicht. In diesem Fall muss das Objekt unter Verwendung einer *Konvergenz* Methode auf den neuen State aktualisiert werden. Wie in [40, Kapitel 9.3.2] beschrieben, ist die einfachste Form davon *Zero-Order Convergence* (oder „Snapping“), wo die Zustandsänderung unverzüglich und ohne *Smoothing* erfolgt. Dies kann jedoch zu unangenehm ruckartigen oder sogar unmöglichen Bewegungen führen. Es empfiehlt sich daher, die Überführung von dem alten in den neuen Zustand durch Interpolation über einen gewissen Zeitraum zu glätten. Das hat zwar wiederum eine entsprechende zusätzliche Verzögerung zur Folge, führt jedoch zu deutlich flüssigeren Bewegungen.

In [6] wird ein weiterer Ansatz vorgestellt, wobei Dead Reckoning Prediction mit *Spline Interpolation* kombiniert wird, um noch realistischere Bewegungen zu erreichen. Wie in [1] beschrieben, kann außerdem die Synchronisierung der lokalen Timer (siehe Absschnitt 3.2.1) und das Versehen der Update Nachrichten mit *Time Stamps* zur Erhöhung der Genauigkeit sowie der Fairness von Dead Reckoning beitragen.

3.2 Zeitmanipulation

In [3, Kapitel 6.3] wird darauf hingewiesen, dass auch ohne Prediction die Game States, die bei den Clients dargestellt werden, in fast allen Fällen voneinander abweichen. Der Grund dafür ist wiederum die Netzwerklatenz.

Wenn ein Client weiter vom Server entfernt ist als ein anderer Client, dauert es länger, bis die Netzwerk-Nachrichten bei ihm ankommen. Er kann Ereignisse daher erst später wahrnehmen als ein anderer Spieler und hat somit weniger Zeit um darauf zu reagieren.

Die in diesem Abschnitt vorgestellten Techniken können dazu beitragen, unterschiedliche Latenzwerte der Spieler auszugleichen, wodurch die *Fairness* des Spielverlaufs auch bei ungleichen Netzwerkbedingungen besser bewahrt werden kann.

3.2.1 Zeitsynchronisierung

Die im Folgenden beschriebenen Techniken basieren alle auf dem selben Prinzip. Durch Manipulation des Zeitflusses sollen negative Auswirkungen schlechter oder wechselnder Netzwerkbedingungen auf das Gameplay minimiert werden. Dazu können Netzwerknachrichten zum Beispiel mit einem Timestamp versehen werden, damit sicher gestellt werden kann, dass Kommandos nicht nur in der richtigen Reihenfolge, sondern auch zum richtigen Zeitpunkt ausgeführt werden [23]. Da in verteilten Systemen wie Online Multiplayer Spielen aber keine globale Uhr für die Zeitmessung existiert, handelt es sich dabei um keine einfache Aufgabe. Timestamps, die über das Netzwerk übertragen werden, können nicht mit der lokalen Zeit des Empfängers verglichen werden, da die Messung des Zeitpunktes relativ zur Systemzeit des Senders erfolgt. Wie in [30] beschrieben, ist es aus diesem Grund erforderlich, die Timer aller beteiligten Rechner eines Netzwerk-Spiels miteinander abzugleichen:

> „Many system aspects in real-time multiplayer games are based on assumptions about times. However, a global wall-clock time is not existing in such systems and the forward and backward transmission delays might be different. To reach some common time-base, mechanisms such as NTP (network time protocol) or simple probe-based approaches to determine the skew of clocks can be used.“

Netzwerkprotokolle wie NTP [25] oder SNTP [24] wurden dafür entwickelt um eine solche Zeitsynchronisierung in Computersystemen zu erreichen. Wie in [39] erklärt wird, eignen sich diese Protokolle aufgrund ihrer Komplexität (NTP) oder Ungenauigkeit (SNTP) jedoch nur schlecht für den Einsatz in Computerspielen. Es wird ein einfacherer Algorithmus vorgestellt, der die durchschnittliche Paketumlaufzeit dazu verwendet, um die Uhren aller Hosts in einem Netzwerkspiel zu synchronisieren.

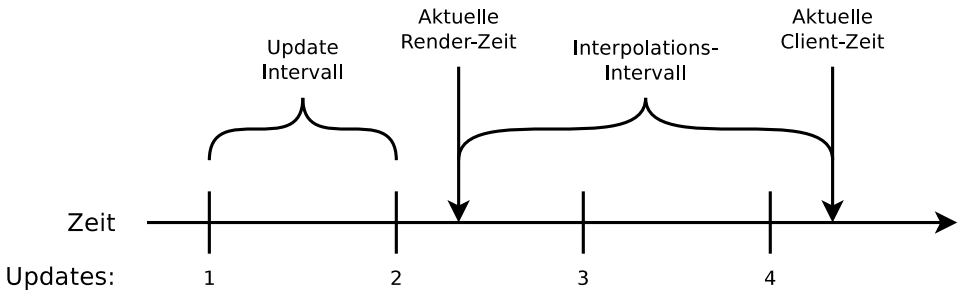


Abbildung 3.3: Entity Interpolation: durch zeitversetzte Darstellung kann zwischen den empfangenen State Updates interpoliert werden um flüssigere Bewegungen zu erhalten. Grafik übernommen aus [44].

3.2.2 Entity Interpolation

Als Alternative zu Dead Reckoning kann für die Darstellung von Gegnern und anderen Objekten auch Interpolation verwendet werden. Während Dead Reckoning versucht, die korrekten Positionen der Objekte zum aktuellen Zeitpunkt durch Extrapolation anzunähern, erfolgt die Darstellung bei Verwendung von Interpolation *in der Vergangenheit*.

Diese Technik wird zum Beispiel in allen Multiplayer Spielen verwendet, die auf der von *Valve* entickelten *Source Engine* [43] basieren. Wie in [44] beschrieben, erhalten die Clients bei diesen Spielen standardmäßig 20 State Updates pro Sekunde vom Server. Das bedeutet, dass im Durchschnitt alle 50 Millisekunden ein neues State Update eintrifft. Source verwendet einen Standardwert von 100 Millisekunden für die Interpolations-Periode, welche auch als *lerp* bezeichnet wird. Das heißt, dass alle Objekte, die nicht vom Client selbst gesteuert werden, an der Position dargestellt werden, wo sie sich vor 100 Millisekunden befunden haben. Auf diese Weise kann sichergestellt werden, dass immer zwei State Updates verfügbar sind, zwischen denen interpoliert werden kann, auch wenn einmal eine Nachricht beim Versenden über das Netzwerk verloren geht.

Abb. 3.3 soll den Ablauf veranschaulichen. Die letzte Nachricht, die der Client über das Netzwerk erhalten hat, war das State Update mit der Nummer 4. Zur Darstellung eines Frames wird nun nicht die aktuelle Client-Zeit verwendet, sondern die Client-Zeit minus das Interpolations-Intervall. Anhand dieser Render-Zeit wird nun zwischen Update 2 und 3 interpoliert. Im Fall eines Paketverlustes, wo Update 3 fehlt, kann stattdessen Update 4 verwendet werden. Ein Problem entsteht erst, wenn mehr als ein Update hintereinander verloren geht. In dem Fall kann auf eine der zuvor beschriebenen Extrapolations-Methoden zurück gegriffen werden, um die Positionen der Objekte ausgehend von dem zuletzt erhaltenen State Update zu berechnen.

Vergleich zu Dead Reckoning

Als Hauptgrund, warum Half-Life den eben beschriebenen Interpolations-Mechanismus zur Darstellung der Gegner verwendet, und nicht Dead Reckoning, gibt Yahn Bernier in [5] die unrealistische Spieler Physik an, sowie die damit zusammenhängenden nicht-deterministischen Bewegungen, welche mit den zuvor beschriebenen Prediction-Methoden nur schwer vorherzusehen sind. Die Verwendung von Interpolation ermöglicht für diese Art von Spielen somit flüssigere Bewegungen, weil dabei keine Prediction-Fehler korrigiert werden müssen.

Der offensichtliche Nachteil ist jedoch, dass für die Darstellung keine aktuellen Daten verwendet werden. Die Spieler sehen die Spielwelt also immer nur so, wie sie in der Vergangenheit ausgesehen hat. Für FPS würde das bedeuten, dass ein Spieler nicht direkt auf einen anderen Spieler zielen kann, um ihn mit einem Schuss zu treffen, da sich der Gegner in der Zwischenzeit mit hoher Wahrscheinlichkeit bereits weiter bewegt hat. Wenn der Server über die Interpolationszeiten der Clients Bescheid weiß, können diese Werte jedoch genutzt werden, um die verzögerte Darstellung auszugleichen, indem der Server den gesamten Game State für die Verarbeitung von Kommandos bis zu dem Zeitpunkt „zurückspult“, zu dem das Ereignis ausgelöst wurde. Diese Technik nennt sich Lag Compensation (siehe Abschnitt 3.2.4).

3.2.3 Time Delay

Während bei dem gerade beschriebenen Interpolations-Verfahren nur Gegner und andere Objekte, die nicht von dem jeweiligen Spieler selbst kontrolliert werden, zeitversetzt dargestellt werden, gehen Time Delay Techniken noch einen Schritt weiter. In [23] stellen die Autoren das Konzept von *Local Lag* vor, wobei auch für Events und Bewegungen des lokalen Spielers eine Verzögerung eingeführt wird, um solche Fälle so gut wie möglich zu vermeiden, wo ein Spieler sich lokal schon mehrere Schritte weiter bewegt haben kann, während er bei anderen Clients immer noch an der Position zu sehen ist, wo er sich etwa 100 Millisekunden zuvor befunden hat. Es handelt sich dabei um einen Kompromiss zwischen *Responsiveness* und *Consistency*. Je höher der Wert für die Verzögerung der Events gewählt wird, desto geringer ist die Wahrscheinlichkeit, dass es zu solchen Abweichungen bei der Darstellung zwischen den Clients kommt. Ist der Wert zu hoch, leidet darunter jedoch das Gameplay, da für den Spieler das Gefühl entsteht, dass das Spiel nicht mehr gut auf Benutzereingaben reagiert. Im Rahmen von [30] wurden die Auswirkungen verschiedener Verzögerungswerte für dieses Verfahren auf ein Autorennspiel untersucht. Die Autoren kamen dabei zu dem Ergebnis, dass Werte unter 50 Millisekunden unkritisch sind. Werte über 100 Millisekunden sollten jedoch vermieden werden. Auf Grund dieser Ergebnisse wird ein Hybrid-Konzept vorgeschlagen als Kombination von Time Delay und Dead

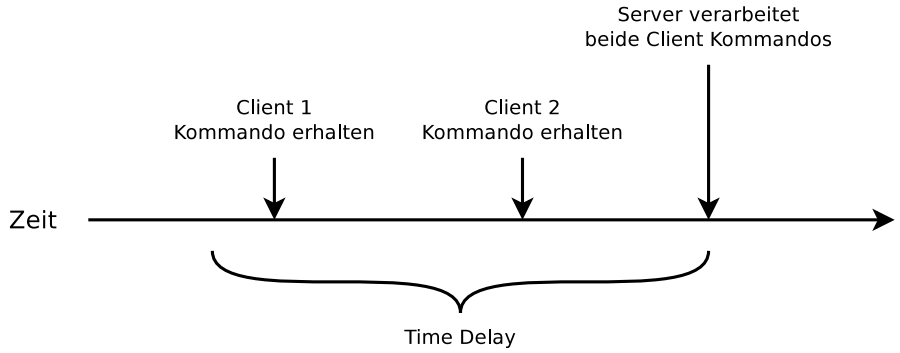


Abbildung 3.4: Server-seitige Time-Delay Technik: um unterschieliche Netzwerklatenzen auszugleichen, werden eingehende Benutzer-Kommandos am Server zwischen gespeichert und zeitverzögert abgearbeitet. Abb. 6.8 aus [3] nachempfunden.

Reckoning, wobei die Darstellung um einen fixen Wert verzögert wird (z.Bsp. 50 ms), während höhere Netzwerklatenzen durch Prediction ausgeglichen werden. Auf diese Weise kann das Intervall für die Extrapolation reduziert werden, wodurch es zu weniger Predictionfehlern kommt.

Eine weiterer Möglichkeit, wie Time Delay zu mehr Fairness in Online Spielen beitragen kann, wird in [3, Kapitel 6.3.1] vorgestellt. Dabei werden alle Client Kommandos, die beim Server eintreffen, zeitverzögert abgearbeitet, damit auch die Spieler mit einer höheren Netzwerklatenz die Möglichkeit haben, auf Ereignisse zu reagieren. Abb. 3.4 skizziert den Prozess, wobei alle Kommandos am Server zwischengespeichert werden, bevor sie verarbeitet werden. Auch bei diesem Verfahren handelt es sich somit um einen Tradeoff zwischen Fairness und Responsiveness, da durch das Buffern der Kommandos zusätzliche Verzögerungen entstehen.

3.2.4 Time Warp / Lag Compensation

Als Time Warp wird in [23] und [3, Kapitel 6.3.2] eine weitere Technik beschrieben, die zur Vermeidung von Inkonsistenzen in verteilten Simulationen wie Online Multiplayer Spielen verwendet werden kann. Das Problem ist, dass auf Grund der Netzwerklatenz Input Kommandos, bis sie beim Server eintreffen, bereits veraltet sind. In [3] wird als Beispiel ein Fall angeführt, wo ein Spieler in einem FPS zum Zeitpunkt t_0 auf einen Gegner schießt. Bis das Kommando zum Zeitpunkt t_2 vom Server verarbeitet werden kann, ist es jedoch möglich, dass sich der Gegner in der Zwischenzeit zu einem Zeitpunkt t_1 schon wieder weiter bewegt hat. Um dieses Problem zu lösen, kann der Server mit Hilfe von Time Warp den Game State zum Zeitpunkt t_0 wieder herstellen. Das bedeutet, dass alle Gegner an die Position zurück bewegt werden, wo sie sich zu dem Zeitpunkt befunden haben, sodass die Überprüfung,

ob es sich bei dem Schuss um einen Treffer handelt, korrekt durchgeführt werden kann. Dieser Vorgang wird auch als *Server Rollback* bezeichnet. Anschließend müssen sämtliche Ereignisse in dem Zeitraum zwischen t_0 und t_2, die während des Rollback Vorgangs rückgängig gemacht wurden, wieder ausgeführt werden, um das Ergebnis zum aktuellen Zeitpunkt t_2 zu erhalten.

Entwickelt wurde dieses Verfahren von Yahn Bernier unter dem Namen *Latency Compensation* bei Valve für Counter Strike und ist seitdem fixer Bestandteil des Netzwerkcodes der Source Engine [15, 44]. Der große Vorteil dieser Technik besteht darin, dass sie sich ideal für den Einsatz in Kombination mit der unter Abschnitt 3.2.2 beschriebenen Entity Interpolation eignet, wobei die Darstellung der Gegner in der Vergangenheit erfolgt. Wenn der Server über diese zeitversetzte Darstellung Bescheid weiß, kann dieser Wert für das Server Rollback verwendet werden, so dass der Game State genau so wieder hergestellt wird, wie der jeweilige Client die Spielwelt zu dem Zeitpunkt gesehen hat, als das Kommando ausgelöst wurde, das gerade beim Server angekommen ist.

Auswirkungen auf das Gameplay

Yahn Bernier weist jedoch in [5] darauf hin, dass es sich auch bei dieser Technik um einen Game Design Tradeoff handelt. Der Vorteil gegenüber dem alten System ist, dass die Spieler nun direkt auf ihre Gegner zielen können. Ohne Lag Compensation war es notwendig, ein gewisses Stück vor andere Spieler zu zielen, wobei die Distanz von der Latenz zum Server abhängig war. Durch die Verwendung von Server Rollbacks kann es aber zu einer anderen Art von Inkonsistenzen kommen. Dies betrifft nun vor allem den Spieler, auf den gerade geschossen wird, für den es in manchen Fällen so wirken kann, als könnte der Gegner „um die Ecke“ schießen. Dieser Fall kann auftreten, wenn der Spieler, der schießt, eine höhere Latenz zum Server hat als der Spieler, der unter Beschuss steht. Dabei kann es zum Beispiel vorkommen, dass der Spieler mit der geringeren Latenz sich bereits hinter eine Ecke in Sicherheit gebracht hat. Der Spieler mit der hohen Latenz sieht aber alles in der Vergangenheit, was bedeutet, dass er seinen Gegner trotzdem noch direkt ins Visier nehmen kann. Wenn dann das Feuer-Kommando beim Server ankommt, wird der Spieler mit der geringeren Latenz in der Zeit an den Ort zurück gesetzt, wo er noch nicht in Sicherheit war und kann so getroffen werden.

Dieses Szenario tritt bei FPS allerdings nur recht selten auf, verglichen mit Situationen, wo zwei Spieler direkt aufeinander zu laufen, oder ein Spieler von einem Gegner abgeschossen wird, ohne dass dieser sich überhaupt in dessen Blickfeld befindet. Lag Compensation eignet sich daher besonders gut für den Einsatz in FPS Spielen.

3.3 Datenkompression

Während die maximale Datenmenge, die pro Zeiteinheit über ein Netzwerk versendet werden kann, durch die verfügbare Bandbreite beschränkt ist, kann durch Reduzierung der Nachrichtengröße außerdem die Latenz verringert werden [3]. Wie in Absatz 2.1.2 erwähnt, ist die Netzwerklatenz auch von der Größe der übertragenen Datenpakete abhängig. Kleine Pakete haben somit eine kürzere Übertragungszeit als große Pakete. Aus diesem Grund sollte darauf geachtet werden, sämtliche Netzwerkdaten so kompakt wie möglich zu übermitteln. Um die Größe der Datenpakete zu minimieren, können die Daten zusätzlich komprimiert werden. Eine solche Datenreduktion kann auf mehrere Arten erreicht werden.

3.3.1 Bitstreams

Eine Möglichkeit zur Komprimierung liegt in der effizienten Kodierung der Daten. Durch die Verwendung von Bitstreams kann sichergestellt werden, dass beim Versenden eines Wertes über das Netzwerk auch nur genau so viele Bits verwendet werden, wie notwendig sind, um den kompletten Wertebereich abzudecken [16]. Soll beispielsweise ein Integer Wert mit den Maximalwerten 0 bis 100 verschickt werden, so werden dafür keine 4 Bytes benötigt. Es reichen bereits 7 bit um alle möglichen Werte abbilden zu können. Wenn sowohl der Sender als auch der Empfänger über diesen Wertebereich Bescheid wissen, kann in diesem Fall mithilfe von Bitstreams eine Kompression um 78% erreicht werden.

3.3.2 Delta Kodierung

Wikipedia beschreibt Delta-Kodierung als „eine Möglichkeit der Datenreduktion, die der platzsparenden Speicherung korrelierter Daten wie zum Beispiel sequenzieller Daten (= Daten die in mehreren Versionen vorliegen) dient“[5]. Man spricht dabei auch von *Differenzspeicherung*, da jeweils nur die Änderungen zu einer vorigen Version der vorliegenden Daten gespeichert wird und nicht die absolute Information. Wenn sich zwei aufeinanderfolgende Versionen eines Datensatzes nur geringfügig voneinander unterscheiden, ist die Speicherung dieser Delta Informationen meist effizienter, da dabei auf alle redundanten Daten, die unverändert bleiben, verzichtet werden kann.

In [26] wird grob erklärt, wie Delta-Kodierung zur Optimierung eines Netzwerkprotokolls für Multiplayer Spiele eingesetzt werden kann, indem mit jeder Nachricht nur die Werte verschickt werden, die sich seit dem letzten Update verändert haben. Als Referenz dient hier immer der letzte Zustand, dessen Erhalt von der anderen Seite bereits über das Ack-System bestätigt wurde.

[5] http://de.wikipedia.org/wiki/Delta-Kodierung

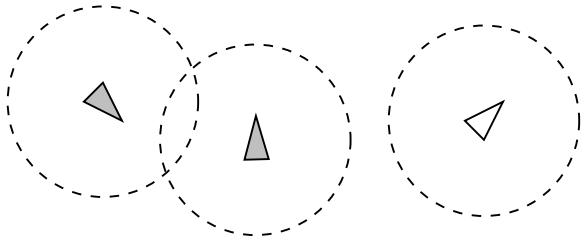


Abbildung 3.5: Symmetrisches Interest-Management: Wenn sich die Auren von zwei Objekten überschneiden, können sie sich gegenseitig wahrnehmen und erhalten Update-Nachrichten. Abb. 9.16 aus [40] nachempfunden.

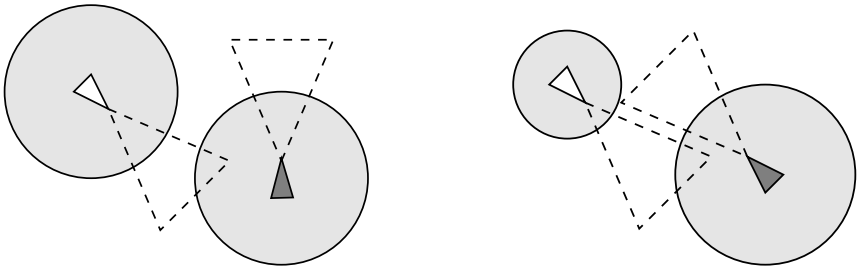


Abbildung 3.6: Asymmetrisches Interest-Management mit Fokus (gestrichelte Bereiche) und Nimbus (graue Bereiche): Der Fokus des weißen Objekts überschneidet sich mit dem Nimbus des grauen Objekts, das weiße Objekt erhält somit Update-Nachrichten von dem grauen Objekt. Das graue Objekt erhält jedoch keine Nachrichten von dem weißen Objekt. Abb. 9.18 aus [40] nachempfunden.

Diese Technik wurde bereits in den Titeln *Quake II* und *Quake III* zur Übertragung von State Updates verwendet, wie Brian Hook, der an der Entwicklung beider Spiele beteiligt war, auf seiner Webseite beschreibt [19].

3.3.3 Interest Management

Bei Interest Management oder Area-of-Interest Filtering handelt es sich um eine Technik, mit der die benötigte Bandbreite eines Spiels maßgeblich gesenkt werden kann, indem mit jedem Update an die Clients nur ein Subset aller Daten anstatt des kompletten Game States übertragen wird [3]. Um zu ermitteln, welche Daten für einen Client von Bedeutung sind, wird eine Area of Interest oder *Aura* verwendet. Das bedeutet, jeder Client erhält nur Update Nachrichten von anderen Objekten, wenn sich ihre Aura mit der des Clients überschneidet (siehe Abb. 3.5).

Wie Abb. 3.6 zeigt, müssen Auren aber nicht symmetrisch sein und können weiter in einen *Fokus* und einen *Nimbus* unterteilt werden, wobei der Fokus den Bereich in der Spielwelt repräsentiert, den ein Objekt selbst wahrnehmen kann, und der Nimbus jenen Bereich, in dem ein Objekt von anderen

Objekten wahrgenommen werden kann [40]. Auf diese Weise könnte zum Beispiel in einem Versteckspiel der Nimbus der Person, die sich versteckt, kleiner sein als der Nimbus des Suchers. Auf diese Weise könnte der Versteckte zwar den Sucher sehen, nicht jedoch umgekehrt.

3.3.4 Message Aggregation

Wie in [40, Kapitel 9.2.2] beschrieben, reduziert Message Aggregation die Frequenz, mit der Update Nachrichten versendet werden, indem mehrere Nachrichten in einem Netzwerkpaket zusammengefasst werden. Somit entsteht weniger Overhead durch Header Informationen, wodurch weitere Bandbreite gespart werden kann. Als Konsequenz verschlechtert sich dabei aber die *Responsiveness* der Anwendung, da das Abwarten weiterer Nachrichten eine zusätzliche Verzögerung bei der Datenübertragung zur Folge hat. Es sollte daher ein fixer Grenzwert festgelegt werden, so dass Nachrichten nie länger als für den bestimmten Zeitraum gebuffert werden.

Kapitel 4

Implementierung der Netzwerkanbindung

Dieses Kapitel bietet zunächst einen Überblick über die Anforderungen an die Netzwerkschicht der im Rahmen dieser Arbeit entwickelten Test-Applikation. Danach folgt eine Auflistung bestehender Netzwerkbibliotheken in Java und eine Beschreibung der Architektur, die schlussendlich zur Umsetzung der Netzwerkanbindung verwendet wurde. Der letzte Abschnitt umfasst eine detailliertere Beschreibung der beteiligten Services sowie deren Aufgaben innerhalb der Netzwerkschicht.

4.1 Anforderungen

Die folgende Auflistung bietet einen Überblick über alle Eigenschaften, welche die im Rahmen dieser Arbeit verwendeten Netzwerkbibliothek erfüllen sollte.

4.1.1 Möglichst geringe Latenz

Da bei einer Echtzeit-Netzwerkanwendung wie einer Physiksimulation die Latenz eine besonders kritische Rolle spielt, ist es wichtig, dass die Datenübertragung so schnell wie möglich ist. Für alle Echtzeitdaten wie State-Updates ist die Übertragungsdauer von größerer Relevanz als eine zuverlässige Übertragung, da ohnehin jeweils nur die aktuellste Version der Daten von Bedeutung ist. Das Hauptaugenmerk der Netzwerkanbindung sollte demnach auf der Geschwindigkeit liegen, was bedeutet, dass die Übertragung der Daten über UDP erfolgen sollte.

4.1.2 Zuverlässiges Versenden von Daten möglich

Auch wenn für den Großteil der zu übertragenden Daten kein zuverlässiges Versenden notwendig ist, ist es in manchen Fällen trotzdem erforderlich si-

cher stellen zu können, dass ein Netzwerkpaket nicht verloren gehen kann. Die Netzwerkanbindung muss also die Möglichkeit bieten, für manche Daten eine zuverlässige Übertragung zu garantieren, während andere Daten unzuverlässig versendet werden können.

4.1.3 Effiziente Kodierung der Daten

Neben der Latenz muss bei der Übertragung von Daten über das Netzwerk auch auf die verfügbare Bandbreite geachtet werden. Diese ist üblicherweise zwar ein geringeres Problem als die Einschränkung durch die Netzwerklatenz, dennoch sollten alle Daten so effizient wie möglich übertragen werden. Das bedeutet, es sollen wirklich nur so viele Bits zum Versenden eines Wertes verwendet werden, wie dafür tatsächlich benötigt werden. Im Optimalfall sollte somit bereits die Implementierung der Netzwerkschicht einen der unter Abschnitt 3.3 vorgestellten Mechanismen zur Kompression der zu versendenden Nachrichten anbieten.

4.1.4 Synchronisierung der Spielzeit

Wie unter Abschnitt 3.2.1 erläutert ist für die Umsetzung einer Echtzeitanwendung über das Netzwerk die Synchronisierung der Systemuhren von zentraler Bedeutung. Auch diese Aufgabe sollte von der Netzwerkschicht übernommen werden.

4.1.5 Flexibel einsetzbar

Zusätzlich zur Bereitstellung der aufgelisteten Funktionen sollte die Netzwerkschicht außerdem so flexibel wie möglich verwendet werden können und folgende Eigenschaften erfüllen:

- Architektur-Unabhängigkeit: Es soll möglich sein, mit den Klassen der Netzwerkschicht sowohl eine Client-Server Anwendung zu implementieren, als auch eine Peer-to-Peer Architektur zu realisieren.
- Genre-Unabhängigkeit: die Bibliothek soll darüber hinaus in jeder Art von Netzwerkanwendung zur Übertragung der Daten verwendet werden können.

4.2 Überblick über Netzwerkbibliotheken in Java

Während der im Rahmen dieser Arbeit durchgeführten Recherche konnten die folgenden bestehenden Netzwerkbibliotheken für Java gefunden werden:

- *Netty*
  https://netty.io/
- *Java Game Server*
  https://github.com/menacher/java-game-server

- *jMonkey Engine*
  http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:networking
- *jgn Java Game Networking*
  http://code.google.com/p/jgn/
- *Narya*
  https://github.com/threerings/narya
- *KryoNet*
  http://code.google.com/p/kryonet/

4.3 Beschreibung der verwendeten Architektur

Die Implementierung der Netzwerkschicht des im Rahmen dieser Arbeit umgesetzten Projektes erfolgte als völlig eigenständige Java Bibliothek, welche sich für den Einsatz in einer Vielzahl von verschiedenen Multiplayer-Spielen zur Kommunikation über das Netzwerk eignet. Es handelt sich dabei um eine Reihe sogenannter *Services* für die Komponenten-basierte Game Engine *Cogaen*[1], welche an der FH Hagenberg unter der Leitung von DI Roman Divotkey entwickelt wird.

4.3.1 Trennung von Spiellogik und Darstellung

Eine grundlegende Eigenschaft der verwendeten Architektur ist die an das *Model-View-Controller*-Muster[2] angelehnte Auftrennung der Applikation in *Logik* und *View*. Diese Architektur ist typisch für die Verwendung von Cogaen und bedeutet konkret die Aufteilung in zwei verschiedene Klassen (*GameLogic* und *InGameView*).

Die Logik verwaltet dabei sämtliche Objekte, die sich in der Spielwelt befinden (sogenannte *Entities*) und kümmert sich um alle Berechnungen.

Die View übernimmt die Erzeugung visueller Repräsentationen für alle sichtbaren Objekte (*EntityRepresentation*) und deren Darstellung auf dem Bildschirm. Darüberhinaus ist die View auch für das Einlesen und Weiterleiten von Benutzereingaben zuständig. Sie bildet somit die Schnittstelle zwischen Spiel und Spieler.

4.3.2 Kommunikation über Events

Die Kommunikation zwischen Logik und View erfolgt ausschließlich über Events. Es handelt sich dabei um eine entkoppelte Form der Kommunikation zwischen Programmobjekten ähnlich dem *Observer*-Pattern [17, Kapitel 5]. So werden auf Seite der Darstellung Input Events für alle relevanten Benutzereingaben generiert, welche des Weiteren an die Spiellogik gesendet und

[1] https://github.com/divotkey/cogaen3-java
[2] http://de.wikipedia.org/wiki/Model_View_Controller

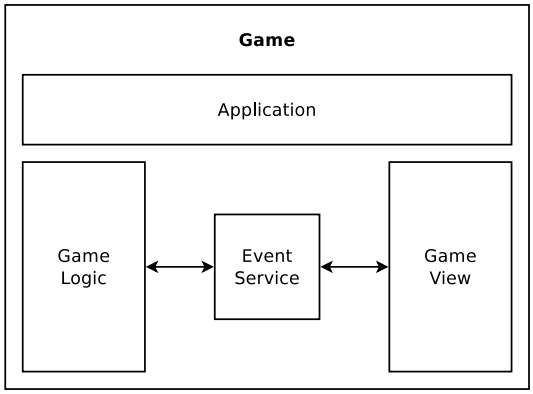


Abbildung 4.1: Architektur Lokal

verarbeitet werden. Auf diese Weise fließen die User Inputs in die Berechnung der Spiellogik ein. Um die Ergebnisse darstellen zu können, wird die View nach jedem Update wiederum über Events benachrichtigt. So können alle Entity-Repräsentationen aktualisiert und an der korrekten Position gerendert werden. Da zwischen der Spiellogik und der Darstellung keine direkte Kommunikation erfolgt, sind die beiden Module voneinander völlig unabhängig (siehe Abb. 4.1).

4.3.3 Client-Server Kommunikation

Der Netzwerkmodus bedient sich einer üblichen Client-Server Architektur, wobei sich mehrere Clients über einen zentralen Server miteinander verbinden können. Der Server ist dabei für die Logik verantwortlich und übernimmt alle Berechnungen. Jeder Client verwaltet eine eigene View und ist somit für die Darstellung der Spielwelt und das Einlesen von Benutzereingaben zuständig.

Dank der losen Kopplung zwischen Logik und View kann im Netzwerkmodus exakt der selbe Programmcode für die Spiellogik und die Darstellung verwendet werden wie in einer lokal ausgeführten Simulation. Der einzige Unterschied besteht dabei darin, dass die Event Nachrichten, die zwischen den beiden Seiten ausgetauscht werden, nun serialisiert und über das Netzwerk verschickt werden müssen. Da dies weiterhin über die Schnittstellen des Eventsystems geschieht, muss an der Implementierung der Logik und der View nichts geändert werden.

Abb. 4.1 zeigt den Aufbau einer lokal ausgeführten Applikation, Abb. 4.2 zeigt den Aufbau einer Applikation im Netzwerkmodus. Aus dem Vergleich ist sofort ersichtlich, dass Logik und View in beiden Fällen nur mit dem Eventsystem kommunizieren.

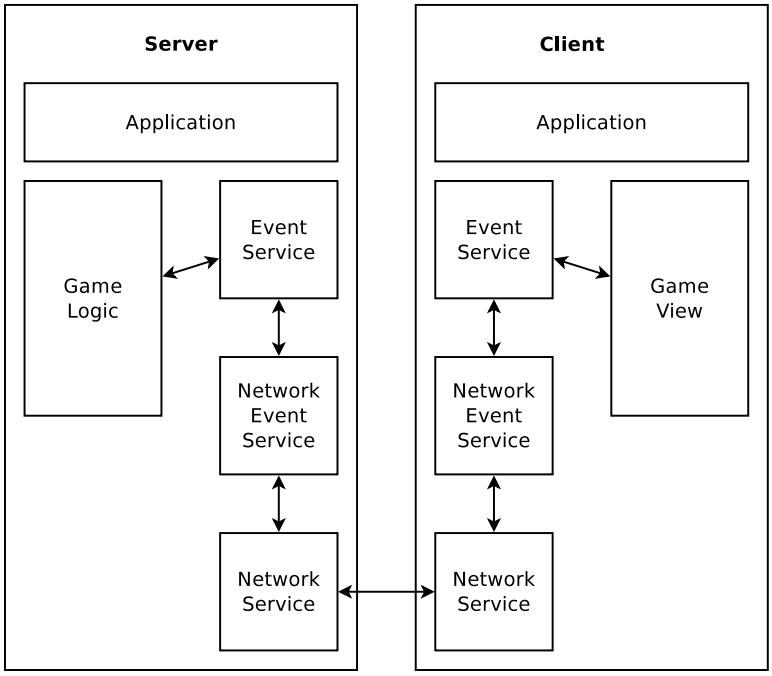


Abbildung 4.2: Architektur Netzwerkmodus

4.3.4 Unterteilung in mehrere Schichten

Eine weitere Eigenschaft der Architektur, die sich bei allen Cogaen Applikationen wiederfindet, ist die Auftrennung in mehrere Schichten, so genannte *Services*. Wie aus Abb. 4.2 ersichtlich, erfolgt die eigentliche Netzwerkkommunikation über die Klasse *Network Service*. Logik und View sind aber nur von der Schnittstelle des *Eventsystem*s abhängig. Die Verbindung zwischen den beiden Schichten wird über den *Network-Event Service* hergestellt, der die Event Objekte serialisiert und an die Netzwerkschicht weiterleitet. Dazu wird ein weiterer Service verwendet, der *IOService*, welcher in der Grafik nicht dargestellt ist.

4.4 Beschreibung der beteiligten Services

4.4.1 Event Service

Der Event Service bildet nicht nur die Schnittstelle zwischen Spiellogik und Darstellung, sondern auf oberster Ebene der Netzwerkschicht auch die Verbindung zwischen zwei Hosts. Um Nachrichten über das Netzwerk zu verschicken, muss die Anwenderschicht nur mit dem Event Service kommunizieren. Wird ein Objekt einer Subklasse von *NetworkEvent* an das Eventsystem verschickt, so wird dieses automatisch an die Netzwerkanbindung weitergereicht

und an den zuvor definierten Empfänger verschickt. Um die Nachricht auf der Gegenseite zu empfangen, genügt die Registrierung eines entsprechenden *Listener* Objektes beim Eventsystem.

Funktionsweise

Der Aufbau des Eventsystems basiert auf dem *Observer* Programmiermuster [17, Kapitel 5]. Das Eventsystem selbst agiert dabei als *Broadcaster*, über den Event Objekte an zuvor registrierte Listener Objekte verteilt werden können.

Event Objekte, die auf diese Weise verschickt werden können, müssen in jedem Fall von der abstrakten Überklasse *Event* ableiten und die Methode *processEvent* implementieren. Um ein Objekt als Listener für eine Klasse von Events registrieren zu können, muss dieses einem bestimmten Interface entsprechen. Dieses ist wiederum von der Klasse des Events abhängig, das der Listener empfangen soll und per Konvention als innere Klasse des Events definiert. So muss ein Objekt etwa die Schnittstelle *KeyEvent.Listener* implementieren, um sich bei dem Eventsystem für den Empfang von Key Events registrieren zu können. Die Registrierung eines Listeners erfolgt dabei über die Methode *addListener*; das Versenden von Events über die Methode *dispatchEvent*.

Das Codebeispiel unter Listing 4.1 zeigt, wie ein Objekt als Listener für KeyEvents registriert werden kann.

Listing 4.1: KeyListenerExample.java

```
1 public class KeyListenerExample implements KeyEvent.Listener {
2
3   public void register(EventSystem eventSystem) {
4     eventSystem.addListener(this, KeyEvent.class);
5   }
6
7   public void unregister(EventSystem eventSystem) {
8     eventSystem.removeListener(this);
9   }
10
11  @Override
12  public void onKeyDown(KeyEvent e) {
13    System.out.println("Key pressed: " + e.getKey);
14  }
15
16  @Override
17  public void onKeyUp(KeyEvent e) {
18    System.out.println("Key released: " + e.getKey);
19  }
20 }
```

Um ein Event Objekt auch über das Netzwerk verschicken zu können, genügt es, wenn dessen Klasse statt von *Event* von *NetworkEvent* ableitet. In dem Fall muss zusätzlich zu der processEvent Methode auch eine *pack* und eine *unpack* Methode bereitgestellt werden, über die definiert werden

kann, welche konkreten Daten dabei versendet werden sollen (genauere Beschreibung siehe Abschnitt 4.4.3 und 4.4.4).

Definition eigener Events

Da jede Event Klasse über ein eigenes Listener Interface verfügt, umfasst die Definition eigener Events üblicherweise zwei Schritte. Zum Einen muss die Klasse Event (bzw. NetworkEvent) erweitert werden und zum anderen eine passende Schnittstelle definiert werden, die über geeignete Methoden zum Empfangen dieser Event Objekte verfügt. Danach muss noch die *processEvent* Methode der Klasse *Event* überschrieben werden. Diese wird vom Event Service aufgerufen, wenn ein Event Objekt verarbeitet wird. Als Parameter wird dabei das Listener Objekt übergeben, welches das Event empfangen soll. Hier muss demnach vom Event die dafür vorgesehene Methode aufgerufen werden, die zuvor über das eigene Listener Interface definiert worden ist. Die folgende Implementierung der Klasse *KeyEvent* soll die Anwendung dieses Konzeptes verdeutlichen.

Listing 4.2: KeyEvent.java

```
1  public class KeyEvent extends Event<KeyEvent.Listener> {
2
3    /**
4     * Defines the different types of key events.
5     */
6    public static enum Type {
7      KEY_PRESSED, KEY_RELEASED;
8    }
9
10   private Type  mType;
11
12   [...]
13
14   @Override
15   public void processEvent(Listener listener) {
16     switch (mType) {
17       case KEY_PRESSED: listener.onKeyDown(this); return;
18       case KEY_RELEASED: listener.onKeyUp(this); return;
19     }
20   }
21
22   public static interface Listener {
23
24     /**
25      * Invoked when a key on the keyboard is pressed.
26      */
27     public void onKeyDown(KeyEvent e);
28
29     /**
30      * Invoked when a key on the keyboard is released.
31      */
```

```
32     public void onKeyUp(KeyEvent e);
33   }
34 }
```

4.4.2 Network Service

Der Network Service bildet die unterste Schicht der Netzwerkanbindung und ermöglicht die Erstellung virtueller Verbindungen zwischen zwei Hosts. Darüberhinaus werden Methoden zum Versenden und Empfangen von Datenpaketen bereitgestellt. Für die Übertragung wird UDP verwendet.

Diese Ebene bietet noch keine Möglichkeit der zuverlässigen Übertragung von Daten über das Netzwerk. Stattdessen wird ein Mechanismus zur Benachrichtigung über den Zustellungs-Status von gesendeten Paketen verwendet. So können verloren gegangene Pakete erkannt und wichtige Daten erneut übermittelt werden. Die Implementierung einer Zustellungs-Garantie bleibt somit dem übergeordneten Network Event Service überlassen. Das bringt zwei entscheidende Vorteile mit sich. Zum einen können Pakete nicht nur zuverlässig und unzuverlässig versendet werden (vgl. UDP vs. TCP), sondern beliebig viele verschiedene Zustellungsgarantien definiert werden. Zusätzlich kann die Art der Übermittlung so nicht nur per Paket gewählt werden, sondern für jede Nachricht innerhalb eines Netzwerkpaketes eine eigene Übertragungsart bestimmt werden.

Erstellen Virtueller Verbindungen

Um eine Netzwerkverbindung zwischen zwei Hosts aufzubauen, wird zuerst ein Objekt der Klasse *Peer* benötigt. Über dieses Objekt können daraufhin Datenpakete zwischen den verbundenen Hosts ausgetauscht werden. Um ein solches Peer Objekt zu erzeugen, verfügt die Klasse NetworkService über eine *createPeer* Methode, der die IP-Adresse sowie die Port-Nummer des Hosts übergeben werden, zu dem eine Verbindung hergestellt werden soll.

Ein neu erzeugtes Peer Objekt befindet sich zunächst in einem unverbundenen Zustand. Über einen Aufruf der *connect* Methode kann der Aufbau einer Verbindung initiiert werden. Dabei wird wie in Kapitel 2.3.2 beschrieben eine simple Connect-Nachricht verschickt. Wenn diese Nachricht auf der anderen Seite ankommt, wird auch beim Empfänger ein Peer Objekt zur Verwaltung der neuen Netzwerkverbindung angelegt. Die beiden Peer Objekte sind nun logisch miteinander verknüpft und befinden sich in einem verbundenen Zustand, so lange bis eine der beiden Seiten die Verbindung wieder trennt. Dies geschieht über einen Aufruf der *disconnect* Methode.

Hat die Gegenseite den Aufbau einer Verbindung akzeptiert, bzw. die Verbindung getrennt, sendet sie wiederum eine *connect* bzw. *disconnect* Nachricht zurück an den ersten Host. Daraufhin wird vom Network Service ein

Event[3] ausgelöst, das registrierte Listener Objekte über den Aufbau bzw. die Trennung der Verbindung informiert.

Versenden und Empfangen von Daten

Die einfachste Möglichkeit, Nachrichten zwischen zwei verbundenen Hosts auszutauschen, ist das Versenden von Event Objekten, wie bereits unter Abschnitt 4.4.1 erklärt. Wird ein Objekt einer Subklasse von *NetworkEvent* an das EventSystem übergeben, wird dieses automatisch an den *NetworkEvent-Service* weitergericht. Dieser übernimmt in weiterer Folge die Serialisierung der Event Objekte sowie das Versenden der Daten mit Hilfe des Network-Services. Für eine detailliertere Beschreibung siehe Abschnitt 4.4.4.

Es ist allerdings auch möglich, Daten über das Netzwerk direkt zu versenden und auf die Verwendung von Netzwerk Events zu verzichten. Auf diese Weise könnte etwa eine eigene High-Level Netzwerkschicht aufbauend auf den Network Service implementiert werden. Der direkte Zugriff auf die Netzwerkverbindung erfolgt dabei wieder über ein entsprechendes Peer Objekt. Die Peer Klasse verfügt dazu über eine *send* sowie eine *receive* Methode. Beide Methoden erwarten ein *ByteBuffer* Objekt[4] als Übergabeparameter. Die send Methode übernimmt die in dem Buffer gespeicherten Daten und erzeugt daraus ein Datagramm Paket, welches daraufhin an den mit dem Peer Objekt verbundenen Host verschickt wird. Beim Empfangen von Daten wird der übergebene Bytebuffer mit den Daten aus dem zuletzt erhaltenen Datagramm Paket befüllt.

Zustellungs-Benachrichtigungen

Das zu Beginn dieses Abschnitts [4.4.2] beschriebene Konzept der verwendeten Zustellungs-Benachrichtigungen stammt ursprünglich aus dem *Tribes Netzwerkmodell*, welches zuerst in *Starsiege Tribes* implementiert wurde. Dabei handelt es sich um ein Multiplayer-Onlinespiel, das von *Dynamix* entwickelt und im Dezember 1998 über das Entwicklerstudio *Sierra Entertainment* veröffentlicht wurde. In [16] beschreiben die Entwickler diesen Mechanismus wie folgt:

> „The Connection manager provides a virtual connection between two hosts and while it does not provide delivery guarantees, it does provide packet delivery status notifications. [...] How each delivery mode is handled is delegated to a higher level - the connection manager only guarantees the correct notification of a sent packet's status."

[3] ConnectionEvent.java
[4] http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html

In Bezug auf die Netzwerkanbindung, die im Rahmen dieser Arbeit entwickelt wurde, bedeutet das, dass der Network Service alle benötigten Funktionen zum Versenden und Empfangen von Daten über das Netzwerk zur Verfügung stellt (siehe 4.4.2). Die Übertragung der Daten erfolgt jedoch unter Verwendung des unzuverlässigen UDP Protokolls. Das heißt, es ist nicht sicher gestellt, dass gesendete Daten auch wirklich beim Empfänger ankommen. Geht ein Datenpaket unterwegs verloren, wird dies jedoch vom Network Service erkannt, der daraufhin den Network-Event Service in Form eines Events[5] informiert. Auf diese Weise bleibt es der höheren Schicht überlassen, wie im Fall von Paketverlusten reagiert werden soll. Dies ermöglicht wiederum die Implementierung verschiedener Zustellungsgarantien für jede einzelne Nachricht, die verschickt werden soll. Für eine genauere Beschreibung dazu siehe Abschnitt 4.4.4.

Netzwerkprotokoll

Zur Erkennung verloren gegangener Pakete verwendet der Network Service das unter Abschnitt 2.3.3 beschriebene Konzept von Sequenznummern und Acks. Jedes versendete Paket wird dabei mit folgenden Header Informationen versehen:

- *Paket ID* (2 bit): Dient zur Unterscheidung zwischen connect/disconnect Nachrichten und Datenpaketen.
- *Sequenznummer* (11 bit): Fortlaufende Nummerierung zur Identifikation der Pakete.
- *Ack* (11 bit): Nummer des zuletzt empfangenen Pakets.
- *Ack-Bitfield* (15 bit): Das n-te Bit bestätigt jeweils den Erhalt des Pakets mit der Sequenznummer ack - n.

Daraus ergibt sich ein Overhead von 5 Byte pro Datenpaket. So können an jedes Paket, das verschickt wird, Empfangsbestätigungen für die letzten 16 eingegangenen Pakete angefügt werden. Die erfolgreiche Übermittlung eines Pakets ist gesichert, sobald eine solche Empfangsbestätigung für das jeweilige Paket erhalten wurde. Um ein Paket als verloren zu melden, muss hingegen ein bestimmter Grenzwert festgelegt werden. Dieser ist von der durchschnittlichen Paketumlaufzeit abhängig. Dauert es bei einem Paket also vergleichsweise erheblich länger, ohne eine Empfangsbestätigung dafür zu erhalten als im Durchschnitt, so kann davon ausgegangen werden, dass es am Weg verloren ging. In dem Fall wird vom Network Service eine Verlustmeldung ausgelöst.

[5] PackageDeliveryEvent.java

Zeitsynchronisierung

Neben der Verwaltung der aktiven Verbindungen zwischen den Hosts kümmert sich der Network Service auch um die unter Abschnitt 3.2.1 beschriebene Synchronisierung der Systemuhren. Jedes Peer Objekt verfügt über ein eigenes *Timer* Objekt, dessen Zeit mit der Spielzeit des verbundenen Hosts abgestimmt wird. Der dazu verwendete Algorithmus ist eine Abwandlung der in [39] vorgestellten Methode und umfasst folgende Schritte:

1. Für jedes Paket, das von A versendet wird, wird der Zeitpunkt des Versendens lokal gespeichert.
2. Wenn B das Paket erhält, wird ein Antwortpaket mit der aktuellen Spielzeit von B zurück an A geschickt
3. Wenn das Antwortpaket wieder bei A ankommt, wird erneut die lokale Zeit gemessen.
4. A verfügt nun über 3 Zeitwerte. Den ursprünglichen Timestamp (Zeitpunkt des Versendens), den Timestamp von B und den aktuellen lokalen Timestamp.
5. A kann somit die Roundtriptime des Pakets als die Differenz aus aktuellem und ursprünglichem Timestamp berechnen.
6. Die letzten 32 Werte für die Roundtriptime werden in einem Array gespeichert.
7. Als durchschnittliche Latenz wird die Hälfte der Roundtriptime angenommen (Zeit, die das Antwortpaket für den Weg von B nach A benötigt).
8. Dieser Latenzwert wird zu dem Timestamp von B addiert, woraus sich (in etwa) die aktuelle Spielzeit von B ergibt.
9. Liegt die aktuelle Roundtriptime innerhalb des 20. Perzentils und ist die Differenz der Zeitwerte von A und dem berechneten Wert für B größer als 5 Millisekunden, wird der lokale Timer auf den neuen Wert korrigiert.

Die Systemzeit des Remote Hosts kann über die Methode *getRemoteTime* der Klasse Peer abgefragt werden.

Erweitertes Netzwerkprotokoll mit Timestamps

Die Implementierung des beschriebenen Zeitsynchonisierungs-Algorithmus macht es erforderlich, dass an jedes Netzwerkpaket zusätzlich ein Timestamp angefügt wird. Dazu kann ein weiteres Feld im Paketheader verwendet werden. Da von Cogaen die aktuelle Spielzeit als Double Wert abgebildet wird, würde das jedoch bedeuten, dass dadurch 8 weitere Bytes zur Übertragung der Header Daten benötigt würden. Durch Anwendung des unter Abschnitt 3.3.2 beschriebenen Konzepts der *Delta-Kodierung*, reichen jedoch bereits 2 Bytes zum Versenden der Timestamps aus. Bei dieser Kompressionsmethode

werden statt absoluter Werte immer nur die Änderungen in Bezug auf einen vorherigen Wert gesendet. Das heißt, es wird mit jedem Paket nur mitgeschickt, wie viel Zeit seit einem bereits zuvor gesendeten Zeitwert vergangen ist. Zur Sicherstellung, dass die Empfängerseite den jeweiligen Referenzwert bereits erhalten hat, wird das zuvor beschriebene Ack-System verwendet. Die Zeitwerte werden dabei mit Millisekunden-Genauigkeit übermittelt. Um Rundungsfehler zu vermeiden, wird in periodischen Abständen (alle zehn Sekunden) ein absoluter Timstamp unter Verwendung von 8 Bytes verschickt. Insgesamt ergibt sich somit eine Headergröße von 7 Bytes pro Paket. Jedesmal wenn der Timestamp als Absolutwert gesendet wird, werden 13 Bytes für die Headerdaten benötigt.

4.4.3 IO Service

Die Aufgabe des IO Service besteht in der Serialisierung von Objekten. Die ursprüngliche Anforderung ergab sich aus der Notwendigkeit, Daten in einen *ByteBuffer* zu schreiben, um sie mit dem Network Service verschicken zu können. Dies sollte möglichst einfach und effizient sein. Darüber hinaus sollte es außerdem möglich sein, Objekte unter Verwendung derselben Schnittstellen in eine Datei zu schreiben, so dass für die Übertragung über das Netzwerk als auch für das lokale Abspeichern der Daten jeweils ein und das selbe Interface verwendet werden kann.

Funktionsweise

In Zusammenhang mit dem IO Service spielen die folgenden drei Schnittstellen eine wesentliche Rolle:

- *Packable*: Definiert eine *pack* und eine *unpack* Methode. Alle Objekte, die diesem Interface entsprechen, können mit Hilfe des IO Service serialisiert werden. Die Implementierung der pack/unpack Methode bestimmt dabei, welche Werte tatsächlich übertragen werden sollen.
- *DataOutput*: Stellt Methoden zum Schreiben von Werten zur Verfügung. Wie die Daten weiterverarbeitet werden, hängt dabei von der konkreten Implementierung ab.
- *DataInput*: Kann zum Lesen von verschiedenen Werten verwendet werden. Woher die Daten kommen, ist wiederum von der Implementierung abhängig.

Das Serialisieren eines Objektes umfasst zwei Schritte. Zuerst wird vom IO Service die Klassen-ID des Objektes in den Output-Stream geschrieben und danach die virtuelle pack Methode des Objektes mit dem jeweiligen Output-Stream als Argument aufgerufen. Beim Deserialisieren wird zunächst wiederum die Klassen-ID aus dem Input-Stream gelesen und ein Objekt des entsprechenden Typs allokiert. Daraufhin wird die unpack Methode des neu

erzeugten Objektes aufgerufen, welches somit alle benötigten Daten aus dem Input-Stream einlesen kann.

Auf diese Weise muss der IO Service nicht zwischen den verschiedenen Arten von Objekten unterscheiden und benötigt keinerlei Kenntnis darüber, welche Werte eines Objektes beim Serialisieren verwendet werden sollen. Des Weiteren ist dieser Vorgang an kein bestimmtes Datenformat gebunden, da die Art und Weise, wie die Daten gespeichert bzw. übertragen werden, rein von der Implementierung des Input- und Output-Streams abhängig ist.

Schreiben und Lesen von Objekten

Jedes zu serialisierende Objekt muss folgende Bedingungen erfüllen:

- Es muss das Packable Interface implementieren.
- Es muss einen Default-Konstruktor zur Verfügung stellen.
- Die Klasse des Objekts muss zuvor beim IO Service registriert werden.

Zusätzlich zu der *pack* und *unpack* Methode der *Packable* Schnittstelle muss das Objekt also auch über einen Default-Konstruktor verfügen, da dieser vom IO Service zur Konstruktion eines neuen Objekts beim Einlesen der Daten verwendet wird. Die vorherige Registrierung der Klasse beim IO Service ist deshalb nötig, weil so für die Klasse die eindeutige Klassen-ID generiert wird, welche in weiterer Folge zur Identifikation der serialisierten Objekte dient.

Das Schreiben eines Objektes *object* in einen ByteBuffer *buffer* funktioniert beispielsweise wie folgt:

```
1  BufferOutput out = new BufferOutput(buffer);
2  ioService.writeObject(object, out);
3  out.flush();
```

Um das Objekt wieder aus dem Buffer einzulesen, kann folgender Programmcode verwendet werden:

```
1  BufferInput in = new BufferInput(buffer);
2  Packable serializedObject = ioService.readObject(in);
```

Auf die selbe Weise kann ein Objekt auch in eine Datei geschrieben werden. Dazu muss lediglich das *BufferOutput* Objekt durch eine andere DataOutput Implementierung ersetzt werden. So kann etwa *BinaryOutput* verwendet werden, um Objekte binär in einen beliebigen OutputStream zu schreiben. Eine weitere Implementierung ermöglicht das Schreiben von Daten im CSV Format (*CsvOutput*). Um das korrekte Einlesen der serialisierten Daten zu ermöglichen, ist es wichtig, dass auch ein entsprechendes DataInput Objekt verwendet wird.

Datenkompression über Bitstreams

Zum Lesen und Schreiben von binären Daten wird eine eigene Bitstream Klasse verwendet. Dies hat den unter Abschnitt 3.3.1 erklärten Vorteil, dass alle Daten so kompakt wie möglich übertragen bzw. gespeichert werden können, da nur so viele Bits verwendet werden, wie tatsächlich notwendig sind, um die Werte abzubilden.

Die Verwendung der Bitstream Funktionen erfolgt dabei direkt über das DataInput bzw. DataOutput Interface. Diese bieten Methoden zum Lesen und Schreiben einzelner Bits bzw. Bitfields sowie Integer und Float Datentypen variabler Länge. Das Codebeispiel unter Listing 4.3 soll verdeutlichen, wie einzelne Felder eines Objektes unter Verwendung der minimalen benötigten Bit-Anzahl serialisiert werden können. Bei dem Programmcode handelt es sich um die Implementierung der Header Klasse, welche vom Network Service zur Übertragung der Header Daten verwendet wird (vergleiche Abschnitt 4.4.2).

Listing 4.3: Header.java

```
1  class Header implements Packable {
2
3    static final int BITFIELD_SIZE        = 15;
4    static final int MAX_SEQUENCE_NUMBER  = 2047;
5
6    [...]
7
8    private PackageId  mPackageId;
9    private int        mSeqenceNumber;
10   private int        mAck;
11   private int        mAckBitfield;
12   private double     mTimestamp;
13
14   [...]
15
16   @Override
17   public void pack(DataOutput out) throws IOException {
18     out.writeEnum(mPackageId);                          // uses 2 bits
19     out.writeInt(mSeqenceNumber, MAX_SEQUENCE_NUMBER);  // uses 11 bits
20     out.writeInt(mAck, MAX_SEQUENCE_NUMBER);            // uses 11 bits
21     out.writeBitField(mAckBitfield, BITFIELD_SIZE);     // uses 15 bits
22     writeTimestamp(out);                                // uses 17 bits
23   }
24
25   @Override
26   public void unpack(DataInput in) throws IOException {
27     mPackageId = in.readEnum(PackageId.class);
28     mSeqenceNumber = in.readInt(MAX_SEQUENCE_NUMBER);
29     mAck = in.readInt(MAX_SEQUENCE_NUMBER);
30     mAckBitfield = (int)in.readBitField(BITFIELD_SIZE);
31     readTimestamp(in);
32   }
33
```

```
34   [...]
35 }
```

4.4.4 Network-Event Service

Wie der Name schon andeutet, bildet der Network-Event Service die Verbindungsstelle zwischen Event Service und Network Service. Zwischen Applikationsschicht und dem Network-Event Service findet zwar keine direkte Kommunikation statt, dieser macht es aber erst möglich, Event Nachrichten über das Netzwerk zu verschicken. Der Service agiert dabei als Vermittler, indem er alle Event Objekte, die von NetworkEvent ableiten, abfängt und an einen oder mehrere verbundene Hosts weiterleitet. Durch die Erweiterung von NetworkEvent müssen die zu versendenden Nachrichten außerdem das unter Abschnitt 4.4.3 vorgestellte Packable Interface implementieren. Auf diese Weise können die Objekte mit Hilfe des IO Services serialisiert werden. Der Network-Event Service kümmert sich um das Schreiben aller Daten in einen ByteBuffer, der daraufhin an die send Methode des entsprechenden Peer Objektes übergeben wird.

Versenden und Empfangen von Netzwerk-Events

Wie bereits erwähnt, funktioniert das Versenden von Netzwerk-Events automatisch sobald ein entsprechendes Event Objekt an den Event Service geschickt wird. Für jede Nachricht kann dabei seperat festgelegt werden, an welchen Host es übermittelt werden soll. Wird kein Empfänger angegeben, wird das Objekt an alle aktiven Verbindungen versandt.

Der folgende Programmcode könnte demnach verwendet werden, um ein NetworkEvent Objekt *event* an alle verbundenen Hosts zu verschicken:

```
1   eventSystem.dispatchEvent(event);
```

Soll eine Nachricht nur an einen bestimmten Host geschickt werden, kann der Empfänger über das entprechende Peer Objekt bestimmt werden. Das folgende Codebeispiel zeigt, wie ein NetworkEvent *event* an den Host geschickt wird, der mit dem Peer Objekt *receiver* verbunden ist:

```
1   eventSystem.dispatchEvent(event.to(receiver));
```

Um die gesendeten Nachrichten auf der Gegenseite zu empfangen, muss ein entsprechendes Listener Objekt beim Event Service registriert werden. Von welchem Host das Event ausgelöst wurde, kann über die Methode *getSender* überprüft werden, welche wieder das jeweilige Peer Objekt zurückliefert, das die Verbindung verwaltet. Weiters stellt die Klasse NetworkEvent eine *getTimestamp* Methode zur Verfügung, die den Zeitpunkt, zu welchem das Event erzeugt und versendet wurde, zurückgibt. Dieser Wert kann allerdings nicht mit der lokalen Spielzeit verglichen werden, da der Wert relativ zu der Spielzeit des Hosts gemessen wird, der das Event generiert hat. Um einen

sinnvollen Vergleich zu erhalten, muss die Systemzeit des Absenders verwendet werden, wie das folgende Codebeispiel demonstriert:

```
1  double eventTime = event.getTimestamp();
2  double remoteTime = event.getSender().getRemoteTime();
3  double timeDiff = remoteTime - eventTime;
```

Zustellungsgarantien für Netzwerk-Events

Über die Eigenschaft *DeliveryPolicy* der Klasse NetworkEvent kann für jedes Event, das über das Netzwerk versendet werden soll, eine eigene Übertragungsmethode gewählt werden. Folgende Arten sind dabei im Moment möglich:

- *Guaranteed*: Die Daten werden zuverlässig übertragen und es wird sichergestellt, dass die Events beim Empfänger in der selben Reihenfolge abgearbeitet werden, in der sie versendet wurden. Das bedeutet zum Einen, dass Daten, die verloren gehen, erneut gesendet werden und zum Anderen, dass die Empfängerseite alle eingehenden Events in einer sortierten Queue zwischenspeichert, um die richtige Reihenfolge garantieren zu können. Dadurch entsteht ein zusätzlicher Overhead von einem Byte pro Event, da auch die Nachrichten mit einer fortlaufenden Sequenznummer versehen werden, um die Sortierung überhaupt erst zu ermöglichen. Darüber hinaus haben Paketverluste wie bei TCP zur Folge, dass der Empfänger auch mit der Verarbeitung aller weiteren Events warten muss, bis das fehlende Event nachgereicht worden ist.
- *Non-Guaranteed*: Diese Nachrichten werden vom Empfänger sofort verarbeitet, wenn sie empfangen werden. Es kommt somit zu keiner Verzögerung durch Buffering und es wird kein zusätzlicher Speicher für die Übertragung benötigt. Allerdings kann es vorkommen, dass Events verloren gehen und nie auf der anderen Seite ankommen. Auch die Reihenfolge, in der die Events ankommen kann nicht sichergestellt werden.

Kapitel 5

Beschreibung der Test-Applikation

Das folgende Kapitel beschreibt die Test Applikation, die im Rahmen dieser Arbeit entwickelt wurde. Es handelt sich dabei um eine Fahrsimulation, bei der mehrere Spieler jeweils ein Fahrzeug steuern können, entweder alleine lokal oder in einer gemeinsamen virtuellen Welt über Netzwerk. Dabei wurden einige der in Kapitel 3 vorgestellten Techniken unter Verwendung verschiedener Parameter getestet und miteinander verglichen. Die Ergebnisse dieser Tests werden in Kapitel 6 präsentiert und analysiert.

5.1 Aufbau der Test-Applikation

Die Implementierung der Test-Applikation erfolgte in Java mit Cogaen[1]. Für die gesamte Netzwerkkommunikation wurde die in Kapitel 4 beschriebene Netzwerkbibliothek verwendet. Für die Berechnung der Fahrzeugphysik kam *JBox2D*[2] zum Einsatz, ein Java Port der C++ Bibliothek *Box2D*[3]. Wie der Name schon sagt, handelt es sich dabei um eine 2D Physik Engine. Für die entwickelte Fahrsimulation wurde demnach eine 2D Fahrzeugphysik implementiert. Es wurde dabei versucht, ein annähernd realistisches Fahrverhalten zu erreichen. Die Darstellung erfolgt in einer Top-Down Ansicht. Abb. 5.1 zeigt einen Screenshot der Test Applikation im Netzwerkmodus mit zwei Spielern.

5.1.1 Netzwerk-Kommunikation

Die entwickelte Test-Applikation basiert auf einer herkömmlichen Client-Server Architektur, wie unter Abschnitt 4.3 bereits beschrieben wurde (siehe

[1] https://github.com/divotkey/cogaen3-java
[2] http://www.jbox2d.org/
[3] http://box2d.org/

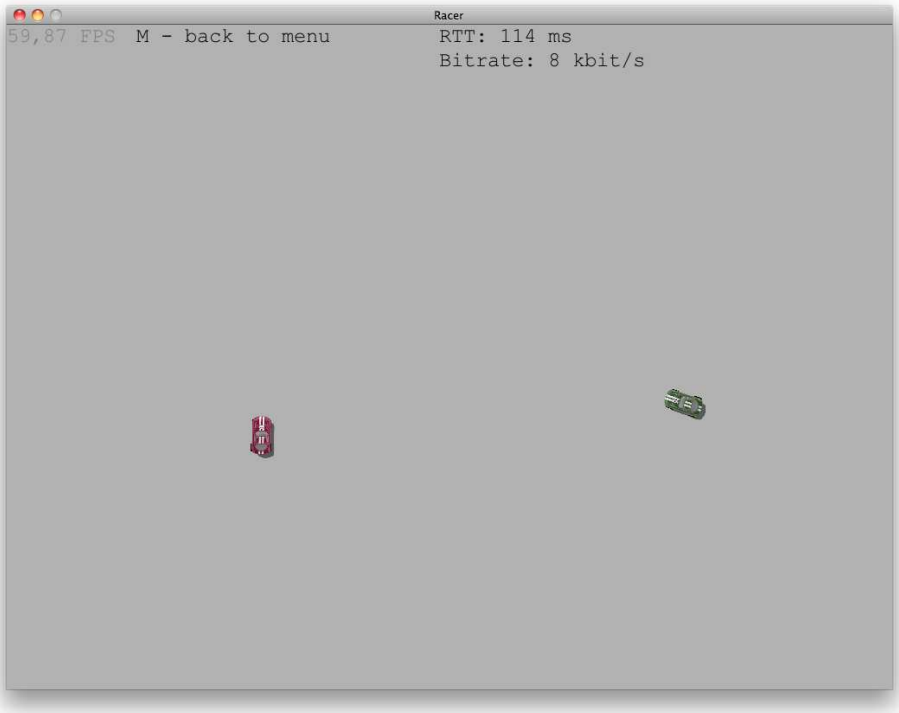


Abbildung 5.1: Screenshot der entwickelten Fahrsimulation: Mehrere Spieler können über Netzwerk ein Fahrzeug in einer gemeinsamen virtuellen Welt steuern. Die Ausgaben am rechten oberen Bildschirmrand geben Auskunft über die aktuellen Netzwerkverhältnisse:

- RTT = aktuelle Round Trip Time in Millisekunden
- Bitrate = aktuell übertragene Datenmenge in Kilobit pro Sekunde

auch Abb. 3.1).

Auf der Clientseite (= Darstellung) wird dabei für jedes Input Event eine Nachricht an den Server geschickt, der die Spiellogik berechnet. Dieser generiert zwanzig mal pro Sekunde einen Snapshot des aktuellen Game States und schickt diese State Updates zurück an alle verbundenen Clients, die sich daraufhin um die Darstellung der empfangenen Daten kümmern. Diese Update-Nachrichten enthalten Positions- und Geschwindigkeitsdaten aller Objekte, sowie deren Ausrichtung in der 2D-Welt. Die Geschwindigkeitswerte können durch Festlegung einer bestimmten Minimal- und Maximalgeschwindigkeit zum Versenden über das Netzwerk mit Hilfe von Bitstreams komprimiert werden (siehe Abschnitt 3.3.1).

5.2 Beschreibung des verwendeten Synchronisierungs-Mechanismus

Für die Synchronisierung der Client-seitigen Darstellung (View) mit der Server-seitigen Spiellogik (GameLogic) wurde eine Kombination der folgenden Techniken verwendet:

- Entity Interpolation / Time Delay (siehe Abschnitt 3.2.2 bzw. 3.2.3),
- Extrapolation / Dead Reckoning (siehe Abschnitt 3.1.2),
- und Spline Interpolation [6, 32].

5.2.1 Verzögerte Darstellung

Wie bei dem unter Abschnitt 3.2.3 vorgestellten Time Delay Verfahren erfolgt die Darstellung der gesamten Spielwelt am Client im Normalfall zeitversetzt. So können Verzögerungen bei der Übertragung der Netzwerkdaten auf Grund von Latenz ausgeglichen werden. Zur Ermittlung der aktuellen Position der Fahrzeuge wird am Client, wie in Abschnitt 3.2.2 beschrieben, zwischen den vom Server erhaltenen State Updates interpoliert. Anstatt einer linearen Interpolation wird jedoch ein *Kubisch Hermitescher Spline*[4] verwendet. Dabei wird zusätzlich zu den Positionsdaten auch die Geschwindigkeit in die Berechnung der Zwischenwerte miteinbezogen. Der direkte Vergleich von Abb. 5.2 und 5.3 zeigt, dass dies eine flüssigere und realistischere Bewegung zur Folge hat. Wie die Autoren von [30] vorschlagen, kann der Wert, um den die Darstellung verzögert wird, außerdem dynamisch bestimmt werden, da bei guten Netzwerkbedingungen ein geringerer Wert ausreichend ist.

5.2.2 Kombination mit Extrapolation

Wie bereits in Abschnitt 3.2.3 beschrieben wurde, sollten zu hohe Verzögerungen bei der Darstellung eines Spiels vermieden werden, da dadurch der Eindruck entsteht, dass das Spiel nicht mehr gut auf Benutzereingaben reagiert. Aus diesem Grund wurde im Rahmen dieser Arbeit zusätzlich zu dem Time-Delay Verfahren eine Form von Dead Reckoning Prediction implementiert. Dank der Verwendung von Spline Interpolation lassen sich diese beiden Techniken gut vereinen [6]. Dazu wird einfach jedes Mal, wenn am Client ein neues State Update vom Server ankommt, mit Hilfe einer der unter Abschnitt 3.1.2 vorgestellten Extrapolations-Methoden vorrausgerechnet. Somit kann die Position, an der sich das Fahrzeug nach Ablauf einer gewissen Zeit (z.Bsp. in einer Sekunde) befinden wird, angenähert werden. Das Ergebnis wird dann am Ende der konstruierten Kurve eingefügt. Auf diese Weise kann im Weiteren das Zeitintervall, welches für das Time Delay verwendet wird, entsprechend verringert werden, da nun nicht mehr die Gefahr besteht, am Ende

[4] http://de.wikipedia.org/wiki/Kubisch_Hermitescher_Spline

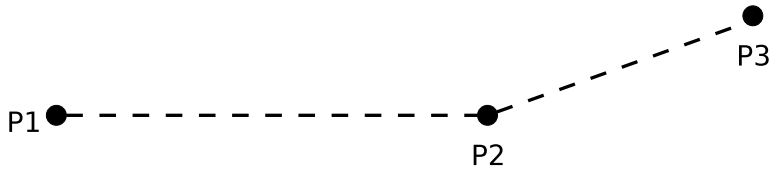


Abbildung 5.2: Lineare Interpolation: die Bewegung zwischen den gegebenen Punkten (P1, P2, P3) erfolgt jeweils entlang einer Geraden.

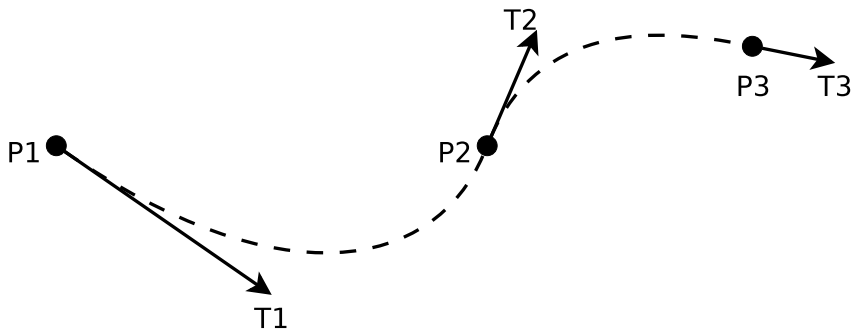


Abbildung 5.3: Spline-Interpolation: bei dieser Form der Interpolation wird eine glatte Kurve verwendet, die alle gegebenen Punkte miteinander verbindet. Dabei werden nicht nur die Positionen (P1, P2, P3) sondern auch die Tangenten (T1, T2, T3) zu den jeweiligen Punkten in die Berechnung der Zwischenstellen miteinbezogen. Dies hat eine flüssigere und realistischere Bewegung zur Folge, da der ursprüngliche Pfad auf diese Weise genauer angenähert wird als durch lineare Interpolation.

des konstruierten Pfades anzukommen, wo die Position nicht mehr bestimmt werden kann (vergleiche Abschnitt 3.2.2). Abb. 5.4 zeigt die Erweiterung der Interpolations-Methode (siehe Abb. 3.3) um die Extrapolations-Technik. Im Extremfall könnte der Wert für das Interpolations-Intervall sogar auf 0 gesetzt werden. Das entspricht dann reinem Dead Reckoning, wo die Darstellung wieder in Echtzeit erfolgt und nur mehr die extrapolierten Werte für die Ermittlung der aktuellen Positionen verwendet werden.

5.2.3 Glättungsfunktion zur Korrektur von Prediction-Fehlern

Wie in Abschnitt 3.1 bereits erwähnt, ist das Problem bei allen Prediction-Methoden das Auftreten von Fehlern. Das bedeutet, dass die extrapolierten Werte in fast allen Fällen von den tatsächlichen Werten abweichen, was dazu führt, dass diese Fehler in irgendeiner Form korrigiert werden müssen. Abb. 5.5 soll diesen Prozess im Falle der eben beschriebenen Verwendung von Spline-Interpolation veranschaulichen.

Schritt 1 zeigt den Fehler in der Berechnung des extrapolierten Wertes P2.

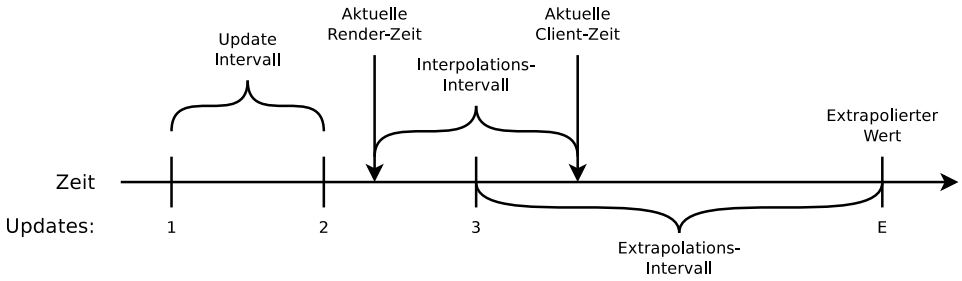


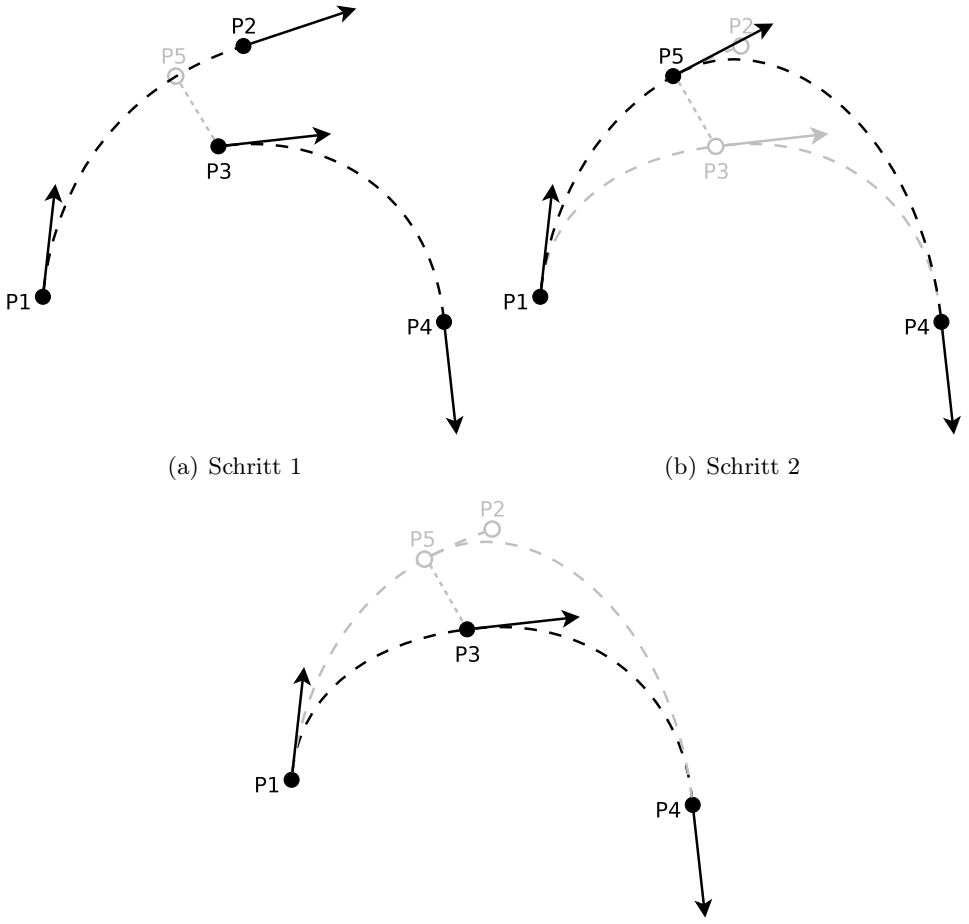
Abbildung 5.4: Interpolation + Extrapolation: zusätzlich zu den vom Server erhaltenen State Updates (1, 2, 3) wird am Ende des konstruierten Pfades noch ein extrapolierter Wert (E) angefügt, der ausgehend von dem letzten State Update (3) mit Hilfe einer Prediction-Methode berechnet wurde.

Die Position des neuen State Updates P3 weicht deutlich von dem Prediction-Ergebnis ab, da es nicht auf der Kurve liegt. P5 markiert die Position, wo sich das Fahrzeug zum Zeitpunkt von P3 befindet, wenn es dem vorrausberechneten Pfad folgt. P4 ist das Ergebnis der Extrapolation ausgehend von dem neuen Zustand P3. Das Fahrzeug soll ab nun somit dem konstruierten Pfad zwischen P3 und P4 folgen. Das heißt, P3 und P4 müssen in die Kurve eingefügt werden, während P2 entfernt werden muss, da es sich dabei um das fehlerhafte Ergebnis der vorigen Prediction handelt. Passiert dies jedoch ohne Glättung, kommt es zu einem Sprung in der Darstellung, da die Position des Fahrzeuges ruckartig geändert wird, was zu einer ungleichmäßigen Bewegung führt. Als Alternative sollte die Überführung des alten Pfades auf den neuen Pfad über einen bestimmten Zeitraum interpoliert werden.

Diese Glättung passiert zwischen Schritt 2 und Schritt 3. Zunächst wird P2 aus der Kurve entfernt, während P5 und P4 in die Kurve eingefügt werden. So lange sich das Fahrzeug zu diesem Zeitpunkt noch vor P5 befindet, hat dies keine sichtbaren Auswirkungen auf die Darstellung, da der Pfad, dem das Fahrzeug gerade folgt, nicht direkt verändert wird. Um das Fahrzeug nun auf den richtigen Pfad zu bringen, wird P5 schrittweise an P3 angenähert; zwischen Schritt 2 und 3 liegt also ein kurzes Zeitintervall. Auf diese Weise wird eine möglichst flüssige Überführung auf den neuen, korrigierten Pfad erreicht.

Der gesamte Extrapolations-Algorithmus besteht aus folgenden Schritten, die jedesmal ausgeführt werden, wenn der Client ein neues State Update vom Server erhält:

1. Client erhält ein State Update vom Server ($P3$), das mit dem Timestamp $t(P3)$ versehen ist.
2. Finde die letzten beiden State Updates aus der Liste der am Client bereits gespeicherten Werte ($P1$ und $P2$).
3. Wenn $t(P3) < t(P1)$, abbrechen. Das neue Update wird ignoriert, da



(a) Schritt 1

(b) Schritt 2

(c) Schritt 3

Abbildung 5.5: Geglättete Korrektur von Prediction-Fehlern am Client:

- P1 = zuletzt erhaltenes State Update
- P2 = auf Grund von P1 vorrausberechnete Position
- P3 = soeben erhaltenes State Update
- P4 = auf Grund von P3 vorrausberechnete Position
- P5 = Ergebnis der Interpolation zwischen P1 und P2 zum Zeitpunkt t(P3)

es sich um veraltete Daten handelt.

4. Berechne die Position $P5$ zum Zeitpunkt $t(P3)$ durch Interpolation zwischen den am Client bereits gespeicherten State Updates.
5. Entferne $P2$ aus der Liste gespeicherter State Updates und füge stattdessen $P5$ ein.
6. Starte Task, der über einen bestimmten Zeitraum $P5$ auf die Werte

von $P3$ interpoliert.

7. Berechne zukünftigen State ($P4$) durch Extrapolation um eine bestimmte Zeit ausgehend von $P3$.
8. Füge $P4$ an das Ende der Liste von gespeicherten State Updates ein.

5.3 Steuerung der Testläufe

Das eben beschriebene Synchronisierungs-Verfahren kann durch verschiedene Parameter beeinflusst werden. Werte wie das Interpolations-Intervall oder das Extrapolations-Intervall für das Time-Delay Verfahren bzw. die Prediction Methode sind variabel. Darüber hinaus können beide Techniken auch komplett deaktiviert werden, so dass sich der gesamte Mechanismus entweder wie reines Time Delay oder reines Dead Reckoning verhält.

Um die optimale Konfiguration für das Verfahren zu finden, bzw. die Vor- und Nachteile der jeweiligen Techniken zu untersuchen, wurden Funktionen zur Aufzeichnung von Benutzereingaben sowie der Pfade, entlang denen sich die Fahrzeuge bewegen, implementiert und mehrere Testläufe mit verschiedenen Einstellungen durchgeführt. Die Auswertung dieser Daten erfolgt in Kapitel 6.

5.3.1 Replay-Funktion für Benutzereingaben

Damit die einzelnen Testläufe bestmöglich miteinander vergleichbar sind, wurde eine Replay-Funktion für Input Events implementiert. Damit können sämtliche Benutzereingaben während der Programmausführung aufgezeichnet und anschließend im CSV Format gespeichert werden. Diese gespeicherten Input Events können beim nächsten Programmstart wieder geladen und ausgeführt werden. Dies ermöglicht die Durchführung mehrerer Testläufe mit verschiedenen Einstellungen, aber mit den selben Benutzereingaben.

5.3.2 Aufzeichnung der Bewegungs-Pfade

Bei den Daten, die schlussendlich miteinander verglichen und ausgewertet werden, handelt es sich um die Pfade, entlang denen sich die Fahrzeuge bewegen. Dazu werden zum Einen die Positionsdaten am Server gesammelt, und zum Anderen die Punkte, an denen die entsprechenden Fahrzeuge zum selben Zeitpunkt am Client dargestellt werden. Auf beiden Seiten wird 50 mal pro Sekunde die aktuelle Position gespeichert und die Liste aller Punkte am Ende in einer CSV Datei abgelegt. Dies ermöglicht die Bewertung der Qualität des verwendeten Synchronisierungs-Mechanismus, etwa durch Berechnung der durchschnittlichen Abweichung zwischen der tatsächlichen Position der Fahrzeuge am Server und dem Ergebnis der Interpolation bzw. Prediction am Client.

5.3.3 Simulierte Latenz

Ein weiterer zentraler Faktor für das Ergebnis der Synchronisierung ist, wie bereits mehrmals erwähnt wurde, die Netzwerklatenz. Um die Anwendung unter verschiedenen, aber klar definierten Netzwerkbedingungen testen zu können wurde eine Funktion umgesetzt, mit der beliebige Werte sowohl für die durchschnittliche Latenz als auch für die Varianz in der Latenz (=*Jitter*) simuliert werden können. Dadurch können auch solche Testsituationen simuliert werden, wo Client und Server weit voneinander entfernt sind, die Tests aber trotzdem lokal ausgeführt werden können. Die Implementierung dieser Funktion erfolgte in der Netzwerkschicht selbst, in der Klasse *Peer*, indem empfangene Netzwerkpakete erst in einem Buffer für die entsprechende Dauer zwischengespeichert werden, bevor sie an die Anwendngsschicht zur Verarbeitung weitergereicht werden.

5.3.4 Anpassung der Werte durch Parameterisierung

Alle Parameter der Test Applikation können über ein Properties File angegeben werden (*„racer.properties“*). Folgende Werte des Synchronisierungs-Verfahrens können durch Parameter beeinflusst werden:

- *networkSimulatedLatency*: Zu simulierende Latenz in Sekunden. Entspricht der Zeit, für die eingehende Netzwerkpakete zwischengespeichert werden, bevor sie verarbeitet werden. Da dies sowohl am Client als auch am Server geschieht, ergibt sich für die Paketumlaufzeit (Round Trip Time) die doppelte Verzögerung.
- *networkSimulatedVariance*: Zu simulierende Varianz der Latenz in Sekunden. Entspricht dem Wert um den die Latenz schwankt (= Jitter). Die Berechnung der Verzögerung für jedes Netzwerkpaket erfolgt nach folgender Gleichung:

$$delay = max(0, latency + (random() - 0.5) \cdot variance); \qquad (5.1)$$

  wobei random() eine zufällige Zahl zwischen 0 und 1 zurückliefert. Es ergibt sich eine Gleichverteilung in folgendem Intervall:
  $[latency - variance/2, latency + variance/2]$.
- *deadReckoningExtrapolationTime*: Extrapolations-Intervall, das für den Prediction Algorithmus verwendet wird. Gibt an, wie weit bei der Extrapolation neuer Werte vorausgerechnet werden soll (siehe Abb. 5.4, Extrapolations-Intervall). Ein Wert von 0 für diesen Parameter hat zur Folge, dass gar keine Prediction durchgeführt wird.
- *deadReckoningInterpolationTime*: Interpolations-Intervall, das für die Fehlerkorrektur des Prediction Algorithmus verwendet wird. Dieser Wert entspricht dem Zeitraum über den Prediction-Fehler korrigiert werden (vergleiche Abb. 5.5: Interpolation von $P5$ zu $P3$).

- *deadReckoningUseAcceleration*: Legt fest, welche Form von Dead Reckoning verwendet werden soll.
  - *true*: Extrapolation unter der Annahme einer konstanten Beschleunigung (siehe Gleichung 3.3).
  - *false*: Extrapolation unter der Annahme einer konstanten Geschwindigkeit ohne Miteinbeziehung der Beschleunigung (siehe Gleichung 3.2).
- *userControlsFile*: Pfad zu der Datei, welche zum Speichern bzw. Laden der Input Events verwendet werden soll. Diese Funktion ist nur im Netzwerkmodus bei genau einem lokalen Spieler wirksam. In allen anderen Fällen werden dieser und die nächsten beiden Parameter ignoriert.
- *readUserControls*: Wenn *true*, werden die gespeicherten Input Events aus der angegebenen Datei geladen und ausgeführt. Die manuelle Steuerung wird dabei deaktiviert.
- *writeUserControls*: Wenn *readUserControls* == *false* und *writeUserControls* == *true*, werden alle Benutzereingaben gesammelt und im CSV Format in die angegebene Datei geschrieben.

Des Weiteren können auch sämtliche physikalischen Eigenschaften der Fahrzeuge wie Reibung und Maximalgeschwindigkeit über Parameter angegeben werden. Dabei handelt es sich um alle Einträge der Properties Datei, die mit dem Präfix „car“ oder „wheel“ beginnen.

Außerdem verfügt das Synchronisierungsverfahren noch über eine weitere Variable, welche nicht über das Properties File geändert werden kann. Es handelt sich dabei um die Verzögerung, die für das Time Delay verwendet wird (siehe Abb. 5.4, Interpolations-Intervall). Da dies nicht immer ein konstanter Wert ist, sondern wie zuvor schon erwähnt auch dynamisch ermittelt werden kann (z.Bsp. in Abhängigkeit von der aktuellen Paketumlaufzeit), musste die Anpassung dieses Wertes für die verschiedenen Testläufe im Code vorgenommen werden.

5.3.5 Auswertung der Ergebnisse

Ziel der Tests war es, geeignete Parameter für das Synchronisierungs-Verfahren zu finden, sowie die Identifizierung der Vor- und Nachteile der jeweiligen Konfigurationen. Um die Ergebnisse miteinander vergleichen zu können, wurde versucht, die aus den Testläufen resultierenden Pfade der Fahrzeuge objektiv zu bewerten. Die gespeicherten Pfaddaten (siehe Abschnitt 5.3.2) lagen jeweils als Liste von (x, y) Werten vor; je eine Liste für die Bewegung eines Fahrzeuges am Server, sowie eine Liste für die Punkte, an denen das Fahrzeug zum selben Zeitpunkt am Client zu sehen war. Dazu wurde auf beiden Seiten jeweils 50 mal pro Sekunde die aktuelle Position gespeichert. Auf Basis dieser Daten wurden die folgenden beiden Eigenschaften berechnet:

Abweichung vom tatsächlichen Pfad

Die einfachste Art zur Bestimmung der Qualität der Synchronisierungs-Methode ist die Berechnung der durchschnittlichen Abweichung zwischen der Darstellung des Autos am Client und der tatsächlichen Position am Server zum selben Zeitpunkt. Die Berechnung dieses Wertes erfolgte nach folgender Gleichung:

$$Deviation_A = \frac{\sum_{i=0}^{n} Dist(server[i], client[i])}{n}, \tag{5.2}$$

wobei gilt:

$$Dist(p_1, p_2) = \sqrt{(p_2.x - p_1.x)^2 + (p_2.y - p_1.y)^2} \tag{5.3}$$

und n für die Anzahl der Elemente in jeder der beiden Listen steht.

Gleichmäßigkeit der Bewegung

Der zweite Wert, der in die Bewertung miteinbezogen wurde, ist die Gleichmäßigkeit der Bewegung der Fahrzeuge. Dazu wurde zunächst für jeden Punkt in den beiden Listen jeweils der Abstand zur vorigen Position berechnet. Für diese Abstände wurde anschließend die *Standardabweichung*[5] vom Durchschnitt ermittelt und das Ergebnis des Clientpfades mit dem Ergebnis der Serverdaten wie folgt verglichen:

$$Deviation_B = |Smoothness(server) - Smoothness(client)|, \tag{5.4}$$

wobei gilt:

$$Smoothness(list) = \sqrt{\sum_{i=1}^{n} \frac{(Dist(list[i], list[i-1]) - Mean(list))^2}{n}}, \tag{5.5}$$

$$Mean(list) = \sum_{i=1}^{n} \frac{(Dist(list[i], list[i-1])}{n}. \tag{5.6}$$

Für beide Werte ist ein möglichst geringes Ergebnis erwünscht, da dies eine geringere Abweichung zwischen den beiden Pfaden bedeutet.

[5] http://de.wikipedia.org/wiki/Standardabweichung

Kapitel 6

Auswertung der Test-Ergebnisse

In Kapitel 5 wurde die entwickelte Test-Applikation sowie die Durchführung der Testläufe beschrieben. In diesem Kapitel folgt die Präsentation und Auswertung der Ergebnisse. Dazu wurden mehrere Testläufe jeweils mit unterschiedlichen Parametern durchgeführt. Um die Ergebnisse besser miteinander vergleichen zu können, wurden jedoch immer die selben Benutzereingaben aus einem zuvor durchgeführten Beispiel-Durchlauf zur Steuerung der Fahrzeuge verwendet (siehe Abschnitt 5.3.1).

Jede Konfiguration wurde außerdem zwei mal getestet; einmal für eine simulierte Latenz von 40 Millisekunden, sowie ein zweites mal mit einem Latenzwert von 80 Millisekunden. Für die simulierte Varianz (Jitter) wurde dabei jeweils die Hälfte des Latenzwertes verwendet, d.h. 20 bzw. 40 Millisekunden (siehe Abschnitt 5.3.4, Parameter *networkSimulatedLatency* und *networkSimulatedVariance*).

6.1 Synchronisierungsverfahren im Vergleich

Die Ergebnisse aus den Testläufen sollen die Unterschiede zwischen den einzelnen Synchronisierungs-Methoden am Beispiel einer 2D Fahrsimulation aufzeigen. Dadurch soll zudem ermittelt werden, welche Technik sich zur Synchronisierung von Physiksimulationen dieser Art besonders gut eignet.

6.1.1 Point-to-Point Methode

Bei dieser Technik handelt es sich um die in Abschnitt 3.1.2 beschriebene Darstellungsform, wo ein Objekt immer erst dann an eine neue Position bewegt wird, wenn ein neues Update-Paket vom Server eintrifft (siehe Gleichung 3.1). Wie bereits erwähnt wurde, liefert diese Methode keine zufriedenstellenden Ergebnisse und ist daher für den Einsatz in Computerspielen

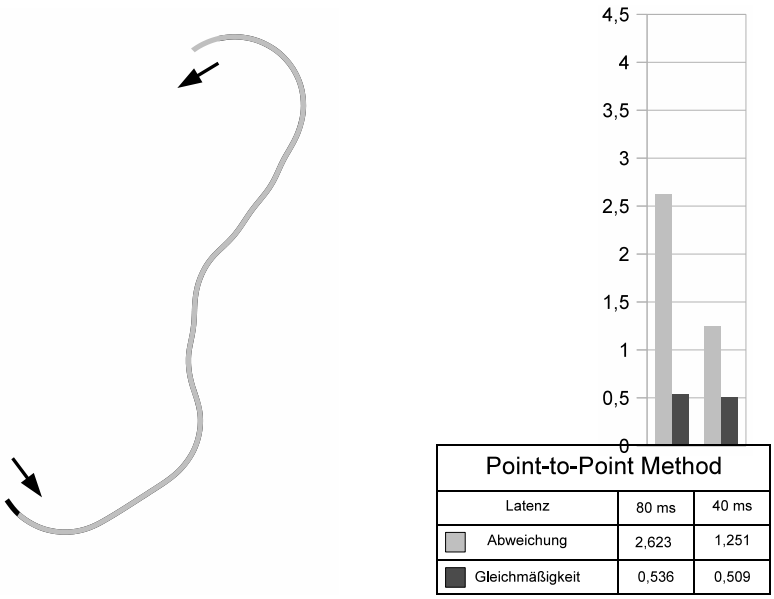
kaum geeignet. Die Miteinbeziehung dieser Methode in das Testverfahren dient rein dem Zweck, um einen Vergleich der anderen Ergebnisse zu einem anzunehmenden „Worst Case“ Szenario zu ermöglichen.

Parameter-Belegung

Um Time-Delay und Dead Reckoning vollständig zu deaktivieren, wurden für diesen Testlauf sowohl der Wert für das Time Delay als auch der Wert für die Extrapolation des Prediction Algorithmus auf 0 gesetzt (siehe Abb. 5.4: Interpolations-Intervall bzw. Extrapolations-Intervall).

Interpretation des Ergebnisses

Abb. 6.1(a) zeigt die Visualisierung der wie in Abschnitt 5.3.2 beschrieben erzeugten Positions-Daten bei einer simulierten Latenz von 80 Millisekunden. Zur Darstellung der Pfade wurden jeweils 150 Punkte aus den Listen von gespeicherten Positionen verwendet. Dies entspricht der Bewegung eines Fahrzeugs über einen Zeitraum von drei Sekunden. Dargestellt sind der Pfad am Server (graue Kurve) und am Client (schwarze Kurve). Die beiden schwarzen Pfeile markieren die Bewegungsrichtung. Die schwarze Kurve ist in der Abbildung kaum erkennbar, da sie beinahe perfekt von der grauen



(a) Ausgabe der Pfade für eine Latenz von 80 Millisekunden.

(b) Ergebnis der in 5.3.5 beschriebenen Berechnungen

Abbildung 6.1: Auswertung des Ergebnisses der Point-to-Point Technik.

überdeckt wird. Aus der Betrachtung dieser Grafik heraus allein ist es demnach nur schwer nachzuvollziehen, warum diese Technik für den Einsatz in Spielen ungeeignet sein soll.

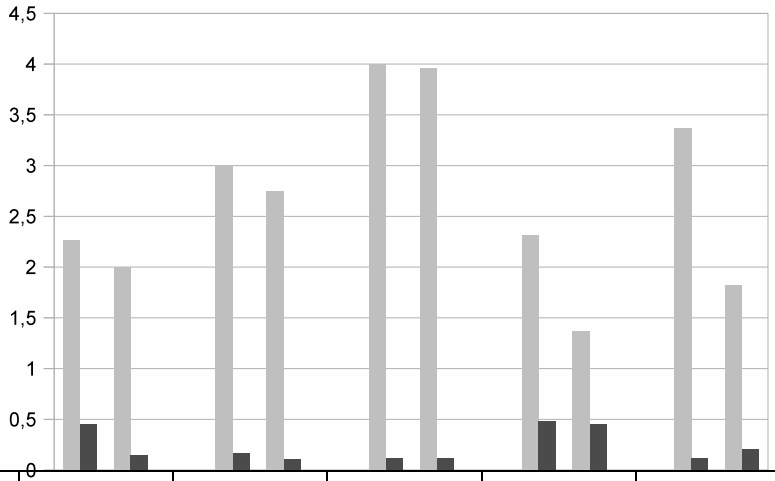
Das Hauptproblem bei Verwendung dieser Technik ist die extrem ungleichmäßige Bewegung, da die Positionen aller Objekte immer erst dann aktualisiert werden, wenn ein neues Positions-Update vom Server ankommt. Bei einer Update-Rate von 20 Paketen pro Sekunde bedeutet dies, dass sich die Fahrzeuge beim Client immer für 50 Millisekunden am selben Punkt befinden, bevor sie an die nächste Position springen, was zu einer sehr ruckeligen Bewegung führt. Hinzu kommt, dass Jitter Effekte nicht ausgeglichen werden können. Das bedeutet, dass jede Schwankung der Latenzwerte zusätzliche Unregelmäßigkeiten für die Bewegung zur Folge hat.

Was aus Abb. 6.1(a) allerdings abgelesen werden kann, ist die Tatsache, dass die beiden Pfade zeitlich etwas voneinander abweichen, da die Anfänge beider Kurven leicht versetzt sind. Dies liegt an der Verzögerung, die sich beim Versenden der Positions-Daten auf Grund der Netzwerk-Latenz ergibt. Diese zeitliche Abweichung, sowie die Unregelmäßigkeiten in der Bewegung zeigt auch die Berechnung der unter Abschnitt 5.3.5 angeführten Kurveneigenschaften (siehe Abb. 6.1(b)):

- Der Wert für die *Abweichung* sagt aus, wie weit die beiden Kurven zu einem bestimmten Zeitpunkt im Durchschnitt voneinander abweichen (siehe Abschnitt 5.3.5). Obwohl sich die beiden Pfade, wie aus Abb. 6.1(a) ersichtlich ist, fast perfekt überlagern, ergibt sich bei einer Latenz von 80 Millisekunden bereits eine Abweichung von mehr als zweieinhalb Metern, was fast einer ganzen Fahrzeuglänge (= drei Meter) entspricht. Der Grund dafür ist die zeitliche Verzögerung zwischen den beiden Pfaden. Die Fahrzeuge werden am Client also im Durchschnitt etwa eine Fahrzeuglänge hinter ihrer tatsächlichen Position dargestellt. Für die Hälfte der Latenz ergibt sich auch ca. der halbe Wert für die Abweichung.
- Der Wert für die *Gleichmäßigkeit* gibt an, wie flüssig die Bewegung der Fahrzeuge ist (siehe Abschnitt 5.3.5). Die Ergebnisse für die Point-to-Point Technik beschreiben dabei wie bereits erwähnt das Worst-Case Szenario, da eine unregelmäßigere Bewegung als bei Verwendung dieser Methode kaum möglich ist. Jede andere Synchronisierungsmethode sollte für diesen Wert daher deutlich bessere Ergebnisse liefern.

6.1.2 Reines Time Delay

Bei dem in diesem Abschnitt getesteten Verfahren handelt es sich um die unter Abschnitt 3.2.3 beschriebene Time-Delay Technik. Dazu wurde jede Form von Prediction deaktiviert, indem für das Extrapolations-Intervall ein Wert von 0 verwendet wurde. Die folgenden Testläufe unterscheiden sich rein in dem Wert, der für das Interpolations-Intervall des Time-Delay Algorithmus



Time Delay	100 ms		150 ms		200 ms		0,5 * RTT		0,75 * RTT	
Latenz	80 ms	40 ms	80 ms	40 ms	80 ms	40 ms	80 ms	40 ms	80 ms	40 ms
Abweichung	2,262	1,996	3,003	2,743	3,994	3,962	2,317	1,363	3,362	1,819
Gleichmäßigkeit	0,450	0,146	0,161	0,103	0,117	0,112	0,476	0,445	0,112	0,208

Abbildung 6.2: Auswertung der Ergebnisse für reines Time Delay.

verwendet wurde. Die Schwierigkeit bei dieser Technik besteht genau darin, nämlich in der Auswahl des richtigen Wertes für die Verzögerung. Abb. 6.2 zeigt die Auswertung der Ergebnisse jeweils bei Verwendung verschiedener konstanter Werte (100 ms, 150 ms und 200 ms) bzw. für solche Fälle, wo die Berechnung des Verzögerungswertes dynamisch erfolgte, und zwar abhängig von der aktuellen Paketumlaufzeit (0,5 * RTT und 0,75 * RTT). Die Visualisierung der Bewegungspfade gleicht in allen fünf Fällen dem Ergebnis der Point-to-Point Methode (siehe Abb. 6.1(a)), mit dem einzigen Unterschied, dass sich der Wert, der für das Time Delay gewählt wurde, in der zeitlichen Versetzung der beiden Pfade wiederspiegelt.

Aus der Statistik sehr gut ersichtlich ist der Zusammenhang zwischen der Verzögerung und dem *Abweichungs*-Ergebnis; je höher der Wert für das Time Delay gewählt wird, desto stärker weichen auch die Bewegungspfade voneinander ab. Dies liegt daran, dass durch die verzögerte Darstellung beim Client alle Fahrzeuge hinter ihrer tatsächlichen Position angezeigt werden. Bei einer Verwendung von 200 Millisekunden Verzögerung weicht die Darstellung zum Beispiel um knapp vier Meter von der eigentlichen Position ab, was mehr als einer ganzen Fahrzeuglänge entspricht, und zwar unabhängig von der aktuellen Netzwerklatenz.

Dass der gewählte Wert für das Time Delay entsprechend groß ist, ist jedoch essentiell für eine flüssige Bewegung, wie das Ergebnis für die Berechnung der *Gleichmäßigkeit* zeigt. Während 100 Millisekunden Verzögerung bei einer Latenz von 80 Millisekunden noch völlig unzureichend sind (dabei

ergibt sich für die Gleichmäßigkeit beinahe das selbe Ergebnis wie bei Verwendung der Point-to-Point Technik), kann durch höhere Delay Werte ein immer besseres Ergebnis erzielt werden. Der Grund für dieses Verhalten liegt darin, dass bei einer zu geringen Verzögerung genau die selben Situationen wie bei der Point-to-Point Methode auftreten, da für die Interpolation immer zwei State Updates benötigt werden. Ist dies einmal nicht mehr der Fall, da die Übertragung über das Netzwerk zu lange dauert oder wenn Daten verloren gehen, bleibt das Fahrzeug beim Client an der zuletzt bekannten Position stehen, bis wieder aktuellere Daten verfügbar sind, wobei es dann zu einem Sprung in der Darstellung kommt. Angewandt auf die in Abb. 5.4 dargestellte Situation tritt dieser Fall immer dann auf, wenn sich die aktuelle Renderzeit *nach* dem Zeitpunkt des zuletzt erhaltenen State Updates (Nummer 3 in der Abbildung) befindet. Die Kombination mit Dead Reckoning versucht genau dieses Problem durch Extrapolation zu lösen.

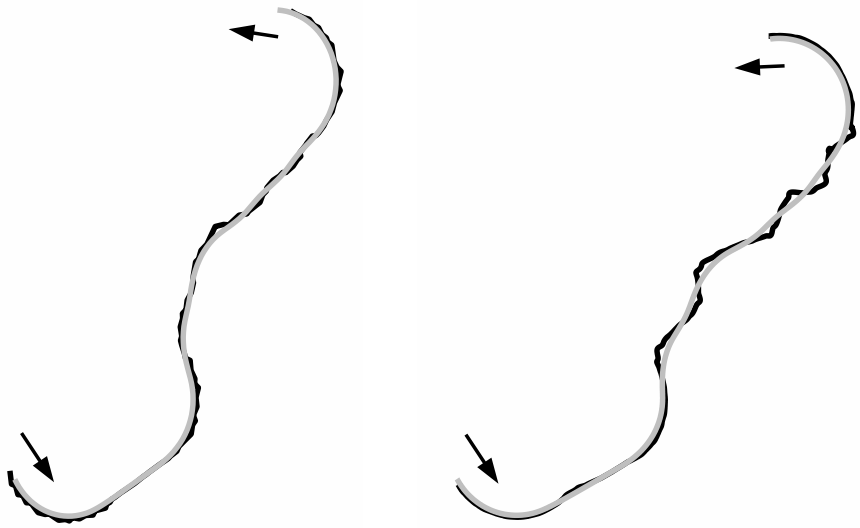
Von den Testläufen ohne Extrapolation lieferte nur der Fall mit 200 Millisekunden Verzögerung für beide Latenzwerte eine durchgehend flüssige Bewegung. Die Ergebnisse für den Gleichmäßigkeits-Wert dieses Durchlaufs können somit auch als ungefährer Referenzwert für ein optimales Ergebnis herangezogen werden. Ein vergleichbar gutes Ergebnis konnte bei 150 Millisekunden Verzögerung nur für 40 Millisekunden Netzwerklatenz bzw. bei Verwendung von drei Vierteln der Round Trip Time für einen Latenzwert von 80 Millisekunden erreicht werden. Das Ergebnis der Tests mit 100 ms Verzögerung bei 40 ms Latenz, bzw. 150 ms Verzögerung bei 80 ms Latenz scheinen auf den ersten Blick zwar beinahe gleich gut zu sein, in diesen Fällen waren jedoch bereits eindeutige Sprünge und „Ruckler“ in der Bewegung der Fahrzeuge wahrnehmbar. Um eine möglichst flüssige Bewegung zu erhalten, ist es also wichtig, dass das Ergebnis der Gleichmäßigkeits-Berechnung möglichst dem Optimalwert entspricht.

6.1.3 Reines Dead Reckoning

Bei der nächsten Technik, die im Zuge dieser Arbeit untersucht wurde, handelt es sich um das in Abschnitt 3.1.2 beschriebene Dead Reckoning; eine Form von Prediction, wo die aktuelle Position von Objekten mit Hilfe von Extrapolation berechnet wird, indem entweder die Geschwindigkeit oder die Beschleunigung der Objekte als konstant angenommen wird. Das Time Delay Verfahren wurde für die Durchführung dieser Tests durch Verwendung eines Verzögerungswertes von 0 vollständig ausgesetzt.

Abb. 6.3(a) und 6.3(b) zeigen die Visualisierung der aus den Testläufen resultierenden Bewegungspfade jeweils für einen Latenzwert von 80 Millisekunden. Im ersten Fall wurde die auf Gleichung 3.2 basierende Extrapolationsform verwendet, während für den zweiten Test die Extrapolation nach Gleichung 3.3 durchgeführt wurde.

Im Gegensatz du den Ergebnissen der Point-to-Point Methode sowie des

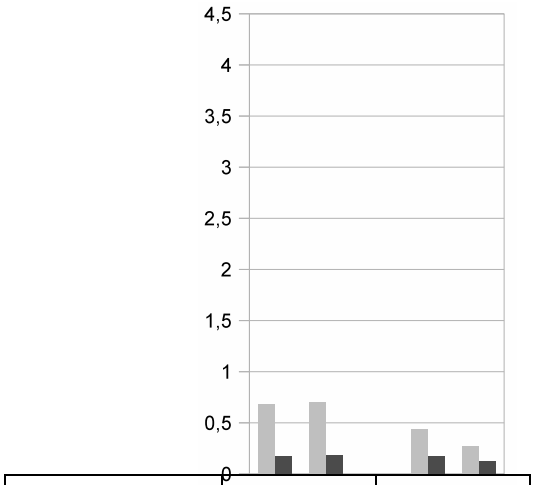


(a) V = konst. (siehe Gleichung 3.2) (b) A = konst. (siehe Gleichung 3.3)

Abbildung 6.3: Ausgabe der Pfade bei Verwendung von Dead Reckoning unter der Annahme einer konstanten (a) Geschwindigkeit, (b) Beschleunigung.

Time Delay Verfahrens, bei denen sich die Pfade von Server- und Clientpositionen nahezu perfekt überdeckten (vergleiche Abb. 6.1(a)), weichen die beiden Kurven bei Verwendung von Dead Reckoning jeweils klar voneinander ab. Dies liegt an den unvermeidlichen Prediction-Fehlern. Aus dem Vergleich der Bewegungskurven mit dem Ergebnis der Point-to-Point Methode ist jedoch auch die deutlich bessere zeitliche Übereinstimmung erkennbar, da die Anfänge des Client- und Server-Pfades jeweils sichtbar näher beisammen liegen. Dies ist auch der Grund für die geringen Werte bei der Berechnung der *Abweichung* (siehe Abb. 6.4). Obwohl Client- und Serverpfad bei Verwendung von Dead Reckoning nicht hundertprozentig übereinstimmen, ist aus der Statistik klar ersichtlich, dass die Abweichung zwischen den beiden Kurven dennoch geringer ist als bei Verwendung eines Time Delays, da die Darstellung dabei ohne zeitliche Verzögerung erfolgt.

Aus dem Vergleich der beiden Extrapolationstechniken bestätigt sich außerdem die in Abschnitt 3.1.2 getätigte Annahme, dass sich Prediction 3.3 besser für die Berechnung der Bewegung von Fahrzeugen eignet als Prediction 3.2. Durch Miteinbeziehung der Beschleunigung in den Extrapolations-Algorithmus können sowohl für den Wert der Abweichung als auch für den Gleichmäßigkeitswert bessere Ergebnisse erzielt werden. In beiden Fällen



Dead Reckoning	Const Vel.		Const Acc.	
Latenz	80 ms	40 ms	80 ms	40 ms
Abweichung	0,683	0,696	0,438	0,271
Gleichmäßigkeit	0,176	0,177	0,168	0,125

Abbildung 6.4: Auswertung der Ergebnisse für Dead Reckoning.

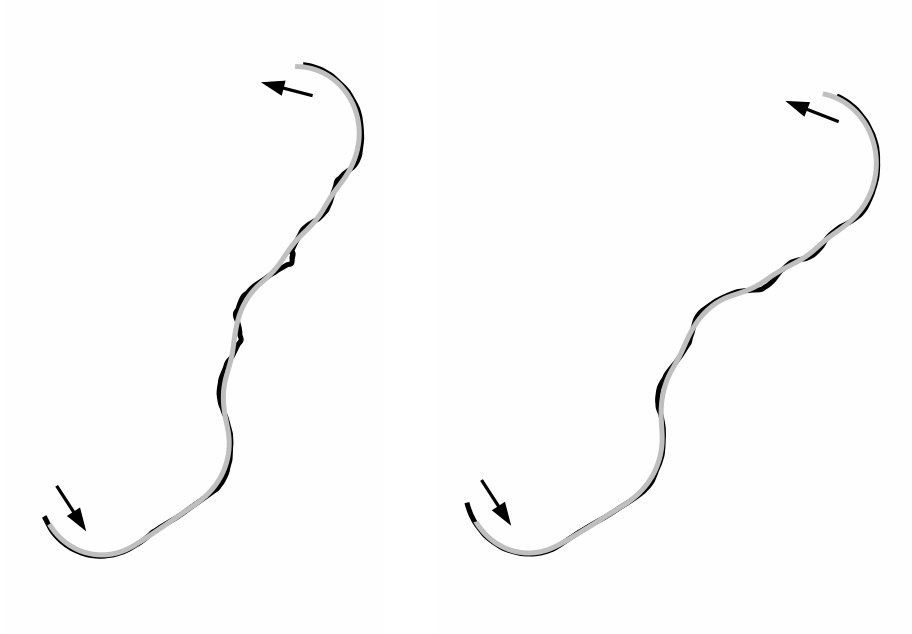
kommt das Ergebnis für die Berechnung der Gleichmäßigkeit aber nicht ganz an den Optimalwert heran (vergleiche Abb. 6.2 bei einem Time Delay von 200 Millisekunden).

Für die Extrapolationsform nach Gleichung 3.2 ist schon aus der Visualisierung der Pfade (Abb. 6.3(a)) die recht unregelmäßige Bewegung abzulesen. Im Gegensatz dazu ergibt sich bei Extrapolation nach Gleichung 3.3 eine sichtbar flüssigere Bewegung (Abb. 6.3(b)). Diese Extrapolationsvariante führt zwar bei jedem Richtungswechsel des Fahrzeugs zu erkennbaren Abweichungen vom tatsächlichen Pfad, in den Kurven ist die Übereinstimmung jedoch fast perfekt.

6.1.4 Kombination beider Techniken

Durch Kombination beider dieser soeben getesteten Techniken können sowohl die Auswirkungen von Prediction Fehlern durch Verwendung eines kleineren Extrapolations-Intervalls verringert, als auch das Hauptproblem der Time Delay Technik gelöst werden, da solche Fälle vermieden werden, wo keine zwei State Updates mehr für die Interpolation verfügbar sind (vergleiche Abb. 5.4).

Abb. 6.5 zeigt die Visualisierung der Bewegungsppfade, die sich aus den Testläufen unter Verwendung von Dead Reckoning in Kombination mit einem dynamisch ermittelten Time-Delay für eine simulierte Latenz von 80 Millisekunden ergaben. In beiden Fällen wurde die Extrapolationsvariante

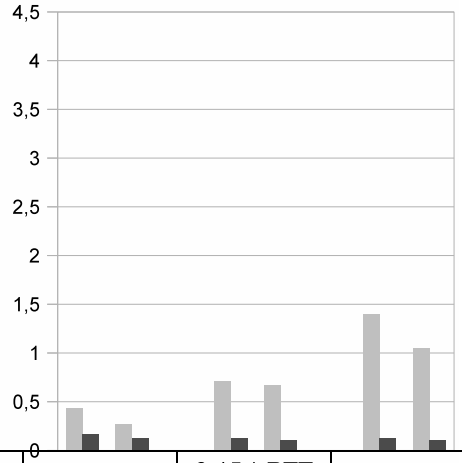


(a) Delay = 0,15 * RTT (b) Delay = 0,3 * RTT

Abbildung 6.5: Ausgabe der Pfade bei Verwendung von Dead Reckoning in Kombination mit einem dynamischen Time Delay abhängig von der aktuellen Round Trip Time für eine Latenz von 80 Millisekunden.

ausgehend von einer konstanten Beschleunigung verwendet (Gleichung 3.3).

Mit steigender Verzögerung kann sowohl die geringere Auswirkung durch Prediction-Fehler als auch die größere zeitliche Verschiebung jeweils zwischen den beiden Kurven beobachtet werden. Dies spiegelt sich auch in den berechneten Kurveneigenschaften wieder (siehe Abb. 6.6). Wie sich bereits bei der Auswertung der Ergebnisse für das Time Delay Verfahren zeigte, hängt der Wert, der sich für die Berechnung der *Abweichung* ergibt, direkt mit dem Wert zusammen, der für die Verzögerung des Time Delay Algorithmus gewählt wird. Durch die Kombination mit Dead Reckoning Prediction kann jedoch bereits bei einem Time Delay Wert von 0,15 mal der aktuellen Round Trip Time ein vergleichbar gutes Ergebnis für die Berechnung der *Gleichmäßigkeit* erzielt werden wie im Falle einer statischen Verzögerung um 200 Millisekunden ohne Verwendung von Prediction, während für den Wert der Abweichung eine Reduktion auf weniger als ein Fünftel erreicht wird (vergleiche Abb. 6.2, Time Delay mit 200 Millisekunden Verzögerung).



DR + Time Delay	0		0,15 * RTT		0,3 * RTT	
Latenz	80 ms	40 ms	80 ms	40 ms	80 ms	40 ms
Abweichung	0,348	0,271	0,716	0,670	1,402	1,046
Gleichmäßigkeit	0,168	0,125	0,126	0,105	0,123	0,105

Abbildung 6.6: Auswertung der Ergebnisse für Dead Reckoning kombiniert mit einem dynamischen Time Delay abhängig von der Round Trip Time.

Kapitel 7

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde versucht, dem Leser einen Eindruck zu vermitteln, worauf bei der Entwicklung von Online Spielen zu achten ist und welche bestehenden Architekturen und Techniken sich zur Implementierung eignen. Da die Wahl der richtigen Architektur und des Netzwerkprotokolls, sowie des passenden Synchronisierungsverfahrens sehr stark von der Art des entwickelten Spiels und den jeweiligen Anforderungen abhängt, ist es wichtig, sich einen Überblick über die Vor- und Nachteile der einzelnen Techniken zu verschaffen. Zu diesem Zweck wurden im Rahmen dieser Arbeit mehrere bestehende Ansätze am Beispiel einer 2D Fahrsimulation getestet und die Ergebnisse miteinander verglichen. Eine eigene Lösung wurde als Kombination aus verschiedenen bestehenden Methoden ausgearbeitet.

7.1 Evaluierung der entwickelten Synchronisierungs-Methode

Aus dem Vergleich der Testergebnisse in Kapitel 6 ist ersichtlich, dass durch die Kombination der Time Delay Technik mit Dead Reckoning Prediction ganz klar eine Verbesserung zu der Prediction-losen Variante erreicht werden kann. Die beiden Verfahren ergänzen sich gegenseitig, was insgesamt zu einem besseren Ergebnis führt.

Eine weitere Stärke des entwickelten Ansatzes liegt in der Steuerbarkeit des gesamten Algorithmus über Parameter. Die Technik kann dadurch leicht an die Anforderungen verschiedener Spiele angepasst werden.

7.1.1 Probleme des vorgestellten Ansatzes

Die Auswertung der Testergebnisse zeigt zwar, dass mit der verwendeten Methode eine gute Synchronisierung der Simulation auch bei wechselnden

Netzwerkbedingungen und in der Gegenwart von Netzwerklatenz erreicht werden kann. Der größte Kritikpunkt an dem getesteten Ansatz ist jedoch das Fehlen jeglicher Input Prediction bzw. Client Side Prediction (siehe Abschnitt 3.1.1). Das bedeutet, dass es unmöglich ist, sehr große Verzögerungen bei der Übertragung auszugleichen. Da das Spiel erst auf Benutzereingaben reagiert, nachdem diese vom Server verarbeitet wurden, können zu hohe Latenzwerte das Spiel trotz allem unspielbar machen.

Eine weitere Schwierigkeit liegt in der Behandlung von Kollisionen. Da die gesamte Berechnung der Spielphysik auf der Serverseite erfolgt, können Kollisionen am Client nicht vorhergesehen werden. In solchen Fällen kommt es daher zu besonders hohen Fehlern bei der Dead Reckoning Prediction.

7.2 Mögliche Verbesserungen und Erweiterungen

Als Fortführung dieser Arbeit könnte untersucht werden, ob sich die beschriebenen Probleme durch Client Side Prediction lösen lassen, oder ob es Sinn macht, sich gänzlich von dem herkömmlichen Client/Server Modell loszulösen. Durch Berechnung der gesamten Spielphysik sowohl am Client als auch am Server würde die strikte Trennung von Spiellogik und Darstellung aufgehoben. Dazu müsste jedoch irgendeine Form von Rollback-Mechanismus für die verwendete Physik-Engine entwickelt werden, um Inkonsistenzen, die sich bei einem solchen Ansatz zwangsläufig ergeben, ausgleichen zu können.

Da die in dieser Arbeit untersuchten Techniken zur Synchronisierung nur objektiven Tests durch Auswertung der Positionsdaten unterzogen wurden, wäre ein weiterer Punkt für mögliche Erweiterungen die Durchführung von Benutzerstudien, um auch einen Eindruck darüber zu erhalten, welche Effekte sich durch die Anwendung der verschiedenen Verfahren für den Spieler ergeben.

Abgesehen davon gibt es noch eine ganze Reihe von verschiedenen Aspekten der Netzwerkspiele-Programmierung, welche in dieser Arbeit zum Großteil zwar Erwähnung fanden, wo mit großer Wahrscheinlichkeit jedoch ebenso Potenzial für Optimierungen offen bleibt. Zum Beispiel könnte versucht werden, die Datenmenge, die über das Netzwerk übertragen wird, mit Hilfe von Techniken wie Interest Management oder Delta Kodierung noch weiter zu reduzieren.

Anhang A

Inhalt der DVD

A.1 Masterarbeit

Pfad: /

Bartsch_2012.pdf . . . Masterarbeit (Gesamtdokument)

A.2 Literatur

Pfad: /Literatur/

Aggarwal.pdf . . . . . Dokument zu [1]
Beigbeder.pdf . . . . . Dokument zu [4]
Bernier.pdf . . . . . Dokument zu [5]
Caldwell.pdf . . . . . Dokument zu [6]
Claypool.pdf . . . . . Dokument zu [7]
Cronin.pdf . . . . . Dokument zu [8]
Dick.pdf . . . . . Dokument zu [10]
Fiedler1.pdf . . . . . Dokument zu [11]
Fiedler2.pdf . . . . . Dokument zu [12]
Fiedler3.pdf . . . . . Dokument zu [13]
Fiedler4.pdf . . . . . Dokument zu [14]
Fiedler5.pdf . . . . . Dokument zu [15]
Frohnmayer.pdf . . . . . Dokument zu [16]
Hook1.pdf . . . . . Dokument zu [18]
Hook2.pdf . . . . . Dokument zu [19]
IEEE_Std_DIS.pdf . . Dokument zu [20]
Jardine.pdf . . . . . Dokument zu [21]
Jiang.pdf . . . . . Dokument zu [22]

Mauve.pdf	Dokument zu [23]
Nielsen.pdf	Dokument zu [27]
OECD.pdf	Dokument zu [28]
Palant.pdf	Dokument zu [29]
Pantel1.pdf	Dokument zu [30]
Pantel2.pdf	Dokument zu [31]
Pipenbrinck.pdf	Dokument zu [32]
Quax.pdf	Dokument zu [36]
Sawashima.pdf	Dokument zu [37]
Sheldon.pdf	Dokument zu [38]
Simpson.pdf	Dokument zu [39]
Smed.pdf	Dokument zu [41]
Terrano.pdf	Dokument zu [42]
Valve1.pdf	Dokument zu [43]
Valve2.pdf	Dokument zu [44]

A.3 Projekt

Pfad: /Projekt/

readme.txt	Textdatei mit Anleitungen zu Installation und Verwendung des Projekts
cogaen-core/	*Cogaen Core*: Kern der Game Engine *Cogaen*[1] (Copyright (c) Roman Divotkey)
cogean-plus/	*Cogaen Plus*: Implementierung der Netzwerkanbindung, sowie einige weitere Cogaen Services, welche im Rahmen dieser Arbeit entwickelt wurden.
graphx2d/	*graphx2d*: die verwendete Grafikbibliothek (entwickelt im Zuge eines Projekts während des Masterstudiums an der FH Hagenberg, Copyright (c) Matthias Bartsch)
graphx2d-lwjgl/	*graphx2d LWJGL*: graphx2d-Implementierung basierend auf der Grafikschnittstelle LWJGL[2]
test-application/	*Testapplikation*: Die im Rahmen dieser Arbeit entwickelte Testapplikation zur Evaluierung der vorgestellten Synchronisierungsmethoden

[1] https://github.com/divotkey/cogaen3-java
[2] http://www.lwjgl.org/

A.4 Testapplikation

Pfad: /Testapplikation/

NetworkedPhysics.jar . .	Ausführbares Java Archiv: startet die Testapplikation
NetworkedPhysics_lib/	Unterordner mit allen externen Abhängigkeiten
racer.properties	Properties-Datei mit allen einstellbaren Parametern
*.dll	LWJGL Windows Natives
*.so	LWJGL Linux Natives
.jnilib,.dylib	LWJGL Mac OSX Natives

A.5 Testdaten

Pfad: /Testdaten/

test-1/	Testergebnisse ohne Extrapolation und ohne Verzögerung
test-2/	Testergebnisse ohne Extrapolation mit 100 ms Verzögerung
test-3/	Testergebnisse ohne Extrapolation mit 150 ms Verzögerung
test-4/	Testergebnisse ohne Extrapolation mit 200 ms Verzögerung
test-5/	Testergebnisse ohne Extrapolation mit $0.5 \cdot$ RTT Verzögerung
test-6/	Testergebnisse ohne Extrapolation mit $0.75 \cdot$ RTT Verzögerung
test-7/	Testergebnisse mit Extrapolation (V = konst.) ohne Verzögerung
test-8/	Testergebnisse mit Extrapolation (A = konst.) ohne Verzögerung
test-9/	Testergebnisse mit Extrapolation (A = const.) und $0.15 \cdot$ RTT Verzögerung
test-10/	Testergebnisse mit Extrapolation (A = const.) und $0.3 \cdot$ RTT Verzögerung

Literaturverzeichnis

[1] Aggarwal, S., H. Banavar, A. Khandelwal, S. Mukherjee und S. Rangarajan: *Accuracy in dead-reckoning based distributed multi-player games*. In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, S. 161–165, Portland, OR, USA, 2004. ACM. http://doi.acm.org/10.1145/1016540.1016559.

[2] Allman, M., V. Paxson und E. Blanton: *TCP Congestion Control*. RFC 5681 (Draft Standard), Sep. 2009. http://tools.ietf.org/html/rfc5681.

[3] Armitage, G., M. Claypool und P. Branch: *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. John Wiley & Sons, Hoboken, NJ, USA, 2006.

[4] Beigbeder, T., R. Coughlan, C. Lusher, J. Plunkett, E. Agu und M. Claypool: *The effects of loss and latency on user performance in unreal tournament 2003®*. In: *Proceedings of the 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, S. 144–151, Portland, OR, USA, 2004. ACM. http://doi.acm.org/10.1145/1016540.1016556.

[5] Bernier, Y.: *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization*. In: *Proceedings of the 15th Game Developers Conference*, GDC 2001, San Jose, CA, USA, März 2001. https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization.

[6] Caldwell, N.: *Defeating Lag With Cubic Splines*. Gamedev.Net, Feb. 2000. http://www.gamedev.net/page/resources/_/technical/multiplayer-and-network-programming/defeating-lag-with-cubic-splines-r914.

[7] Claypool, M., D. LaPoint und J. Winslow: *Network Analysis of Counterstrike and Starcraft*. In: *Proceedings of the 22nd IEEE International Performance, Computing, and Communications Conference (IPCCC)*, S. 261–268, Phoenix, AZ, USA, Apr. 2003. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1203707.

[8] Cronin, E., B. Filstrup, A. R. Kurc und S. Jamin: *An Efficient Synchronization Mechanism for Mirrored Game Architectures*. In: *Proceedings of the 1st workshop on Network and system support for games*, NetGames '02, S. 67–73, Braunschweig, Germany, 2002. ACM. http://doi.acm.org/10.1145/566500.566510.

[9] Deering, S. und R. Hinden: *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard), Dez. 1998. http://tools.ietf.org/html/rfc2460, Updated by RFCs 5095, 5722, 5871, 6437, 6564.

[10] Dick, M., O. Wellnitz und L. Wolf: *Analysis of Factors Affecting Player's Performance and Perception in Multiplayer Games*. In: *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, NetGames '05, S. 1–7, Hawthorne, NY, USA, 2005. ACM. http://doi.acm.org/10.1145/1103599.1103624.

[11] Fiedler, G.: *Networked Physics*. Blog Artikel, Sep. 2006. http://gafferongames.com/game-physics/networked-physics/.

[12] Fiedler, G.: *Reliability and Flow Control*. Blog Artikel, Okt. 2008. http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/.

[13] Fiedler, G.: *UDP vs. TCP*. Blog Artikel, Okt. 2008. http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/.

[14] Fiedler, G.: *Virtual Connection over UDP*. Blog Artikel, Okt. 2008. http://gafferongames.com/networking-for-game-programmers/virtual-connection-over-udp/.

[15] Fiedler, G.: *What every programmer needs to know about game networking*. Blog Artikel, Jan. 2010. http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/.

[16] Frohnmayer, Mark and Gift, T.: *The TRIBES Engine Networking Model*. In: *Proceedings of the Game Developers Conference*, GDC 2000, S. 191–207, San Jose, CA, USA, März 2000. http://www710.univ-lyon1.fr/~jciehl/Public/educ/GAMA/2008/tribes_networking_model.pdf.

[17] Gamma, E., R. Helm, R. E. Johnson und J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[18] Hook, B.: *Introduction to Multiplayer Game Programming*. Blog Artikel. http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/IntroductionToMultiplayerGameProgramming.

[19] Hook, B.: *The Quake3 Networking Model*. Blog Artikel. http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking.

[20] IEEE: *Standard for Information Technology - Protocols for Distributed Interactive Simulations Applications. Entity Information and Interaction (IEEE Std 1278-1993)*. Institute of Electrical and Electronics Engineers, Inc., 1993. http://ieeexplore.ieee.org/servlet/opac?punumber=2826.

[21] Jardine, J. und D. Zappala: *A hybrid architecture for massively multiplayer online games*. In: *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '08, S. 60–65, Worcester, MA, USA, 2008. http://doi.acm.org/10.1145/1517494.1517507.

[22] Jiang, X., F. Safaei und P. Boustead: *Latency and Scalability: A Survey of Issues and Techniques for Supporting Networked Games*. In: *Proceedings of the 13th IEEE International Conference on Networks*, Bd. 1, S. 150–155, Kuala Lumpur, Malaysia, Nov. 2005. IEEE. http://works.bepress.com/pboustead/23.

[23] Mauve, M., J. Vogel, V. Hilt und W. Effelsberg: *Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications*. IEEE Transactions on Multimedia, 6(1):47–57, Feb. 2004. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1261886.

[24] Mills, D.: *Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*. RFC 4330 (Informational), Jan. 2006. http://tools.ietf.org/html/rfc4330, Obsoleted by RFC 5905.

[25] Mills, D., J. Martin, J. Burbank und W. Kasch: *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905 (Proposed Standard), Juni 2010. http://tools.ietf.org/html/rfc5905.

[26] Mulholland, A. und T. Hakala: *Programming Multiplayer Games*. Wordware Publishing Inc., Plano, TX, USA, 2004.

[27] Nielsen: *What Americans Do Online: Social Media And Games Dominate Activity*. Techn. Ber., Nielsen Company, Aug. 2010. http://blog.nielsen.com/nielsenwire/online_mobile/what-americans-do-online-social-media-and-games-dominate-activity/.

[28] OECD: *The Future of the Internet Economy: A Statistical Profile*. OECD Digital Economy Papers, OECD Publishing, Juni 2011. http://www.oecd.org/internet/interneteconomy/48255770.pdf.

[29] Palant, W., C. Griwodz und P. Halvorsen: *Evaluating dead reckoning variations with a multi-player game simulator*. In: *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '06, S. 20–25, Newport, RI, USA, Mai 2006. ACM. http://doi.acm.org/10.1145/1378191.1378196.

[30] Pantel, L. und L. C. Wolf: *On the impact of delay on real-time multiplayer games*. In: *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '02, S. 23–29, Miami, FL, USA, 2002. ACM. http://doi.acm.org/10.1145/507670.507674.

[31] Pantel, L. und L. C. Wolf: *On the suitability of dead reckoning schemes for games*. In: *Proceedings of the 1st workshop on Network and system support for games*, NetGames '02, S. 79–84, Braunschweig, Germany, 2002. ACM. http://doi.acm.org/10.1145/566500.566512.

[32] Pipenbrinck, N.: *Hermite Curve Interpolation*. Blog Artikel, März 1998. http://cubic.org/docs/hermite.htm.

[33] Postel, J.: *User Datagram Protocol*. RFC 768 (Standard), Aug. 1980. http://tools.ietf.org/html/rfc768.

[34] Postel, J.: *Internet Protocol*. RFC 791 (Standard), Sep. 1981. http://tools.ietf.org/html/rfc791, Updated by RFCs 1349, 2474.

[35] Postel, J.: *Transmission Control Protocol*. RFC 793 (Standard), Sep. 1981. http://tools.ietf.org/html/rfc793, Updated by RFCs 1122, 3168, 6093, 6528.

[36] Quax, P., P. Monsieurs, W. Lamotte, D. De Vleeschauwer und N. Degrande: *Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game*. In: *Proceedings of the 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, S. 152–156, Portland, OR, USA, 2004. ACM. http://doi.acm.org/10.1145/1016540.1016557.

[37] Sawashima, H., Y. Hori, H. Sunahara und Y. Oie: *Characteristics of UDP Packet Loss: Effect of TCP Traffic*. In: *Proceedings of the 7th Annual Conference of the Internet Society*, INET '97, Kuala Lumpur, Malaysia, Juni 1997. http://www.isoc.org/inet97/proceedings/F3/F3_1.HTM.

[38] Sheldon, N., E. Girard, S. Borg, M. Claypool und E. Agu: *The effect of latency on user performance in Warcraft III*. In: *Proceedings of the 2nd workshop on Network and system support for games*, NetGames '03, S. 3–14, Redwood City, CA, USA, 2003. ACM. http://doi.acm.org/10.1145/963900.963901.

[39] Simpson, Z. B.: *Time Synchronization*, Bd. 3 d. Reihe *Game Programming Gems*, Kap. 5.1. Charles River Media, 2002. http://www.mine-control.com/zack/timesync/timesync.html.

[40] Smed, J. und H. Hakonen: *Algorithms and Networking for Computer Games*. John Wiley & Sons, Hoboken, NJ, USA, 2006.

[41] Smed, J., T. Kaukoranta und H. Hakonen: *Aspects of networking in multiplayer computer games*. The Electronic Library, 20(2):87–97, 2002. http://dx.doi.org/10.1108/02640470210424392.

[42] Terrano, M. und P. Bettner: *1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond*. In: *Proceedings of the 15th Game Developers Conference*, GDC 2001, San Jose, CA, USA, März 2001. http://www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php.

[43] Valve Developer Community: *Source Engine*. Wiki Artikel. https://developer.valvesoftware.com/wiki/Source_Engine.

[44] Valve Developer Community: *Source Multiplayer Networking*. Wiki Artikel. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.