

Netzwerkprogrammierung - Fighting Lag and Latency

Diplomarbeit

von

**Stefan Hett
aus Neustadt (Hessen)**



Hochschule Aalen
Hochschule für Technik und Wirtschaft –
UNIVERSITY OF APPLIED SCIENCES

Betreuender Dozent:
Abgabetermin:

Prof. Dr. Lecon
07.02.2008

Ehrenwörtliche Erklärung

Ich versichern hiermit, dass ich die Diplomarbeit mit dem Thema „Fighting Lag and Latency“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Stefan Hett)

Inhaltsverzeichnis

1 Einleitung.....	7
1.1 Motivation.....	7
1.2 Problemstellung und -abgrenzung.....	7
1.3 Ziel der Diplomarbeit.....	7
1.4 Aufbau des Dokumentes.....	8
2 Grundlagen.....	8
2.1 Von X bis X4.....	8
2.2 Datenübertragung im Internet.....	10
2.2.1 Das OSI-Referenzmodell.....	10
2.2.1.1 ATM – Sicherungsschicht (Layer 2).....	12
2.2.1.2 IP – Vermittlungsschicht (Layer 3).....	13
2.2.1.3 TCP – Transportschicht (Layer 4).....	14
2.2.1.4 UDP - Transportschicht (Layer 4).....	14
2.2.2 Lag.....	15
2.2.2.1 Die Gesetze der Physik.....	15
2.2.2.2 Serialisierung.....	15
2.2.2.3 Warteschlangen.....	16
2.2.2.4 Round-Trip-Time.....	16
2.2.2.5 Lag, Latenz und Bandbreite.....	17
2.2.3 Lag ist nicht konstant (Jitter).....	17
2.2.4 Packetloss.....	18
2.2.5 Auswirkungen von Lag auf Spiele.....	19
2.3 Netzwerkframeworks.....	19
2.4 Die X4 Game Engine.....	19
3 Vorgehen.....	20
3.1 Evaluierung der Netzwerkframeworks.....	20
3.2 Integration der Netzwerkengine in die Spieleengine.....	20
3.3 Aufbau eines Testnetzwerkes.....	20
3.4 Analyse der Systeme zur Lagkompensierung.....	20
3.5 Entwurf der Netzwerkengine.....	21
3.6 Integration der Netzwerkengine.....	21
3.7 Abschließende Analyse.....	21
3.8 Verwendete Software.....	21
3.8.1 Visual Studio (VS).....	21
3.8.2 Virtual Assist X.....	22
3.8.3 Visual Source Safe (VSS).....	22
3.8.4 VSSConnect.....	22
3.8.5 Debian.....	22
3.8.6 Network Simulator.....	22
3.8.7 Network Animator (NAM).....	23
3.8.8 WireShark.....	23
3.8.9 Fraps.....	24
3.8.10 Adobe Premiere Pro.....	24
4 Techniken zur Reduzierung der Auswirkungen von Lag.....	24
5 Evaluation der Netzwerkframeworks.....	25
5.1 Phase 1: Basisevaluierung.....	25
5.1.1 Abgelehnte Netzwerkframeworks.....	25
5.1.1.1 HawkNL:.....	25

5.1.1.2 Net-Z:	25
5.1.1.3 TinCat	26
5.1.1.4 Torue Network Library	26
5.2 Phase 2: Featureevaluation	26
5.2.1 RakNet	26
5.2.2 ZoidCom	27
5.3 Die Wahl des Netzwerkframeworks	27
6 Netzwerkintegrationstest	27
6.1 Testlauf	28
6.2 Ergebnisse des Integrationstests	28
7 Entwurf und Realisierung	28
7.1 Anpassungen der Basissysteme	28
7.1.1 Singleton-System	28
7.1.1.1 Bestehendes System	28
7.1.1.2 Anforderungen	29
7.1.1.3 Implementierung	29
7.1.1.4 Einschränkungen	30
7.1.2 ID-System	30
7.1.2.1 Bestehendes System	30
7.1.2.2 Anforderungen	31
7.1.2.3 Implementierung	31
7.1.2.4 Einschränkungen	35
7.2 Netzwerkengine	36
7.2.1 Server/Client	37
7.2.2 BitStream	39
7.2.3 Replicator	39
7.2.4 NetworkObject	40
7.2.5 ObjectReference	41
7.3 Clientprediction für den PlayerController	42
8 Network Simulator	43
8.1 Aufbau des Testnetzwerkes	43
8.2 Network Simulator Skript	43
9 Abschließende Performancetests	45
9.1 Testmodul	45
9.2 Testauswertung	45
Anhang A - Quellenverzeichnis	49
Anhang B - Glossar	50

Abbildungsverzeichnis

Abbildung 1: Screenshot aus dem Spiel Elite.....	8
Abbildung 2: X1 bis X4.....	9
Abbildung 3: OSI-Referenzmodell.....	10
Abbildung 4: OSI-Referenzmodell (Router).....	11
Abbildung 5: ATM-Header.....	12
Abbildung 6: IP-Header.....	13
Abbildung 7: TCP-Header.....	14
Abbildung 8: UDP-Header.....	14
Abbildung 9: Network Animator Screenshot.....	23
Abbildung 10: WireShark Screenshot.....	23
Abbildung 11: Adobe Premiere Pro Screenshot.....	24
Abbildung 12: Evaluierung RakNet/ZoidCom.....	26
Abbildung 13: Klassendiagramm: Singleton.....	29
Abbildung 14: Klassendiagramm: IDMap.....	32
Abbildung 15: Klassendiagramm: IDBase.....	33
Abbildung 16: Netzwerkengine Übersicht.....	36
Abbildung 17: Klassendiagramm: Server.....	37
Abbildung 18: Klassendiagramm: Client.....	38
Abbildung 19: Klassendiagramm: Replicator.....	39
Abbildung 20: Klassendiagramm: Network Object.....	40
Abbildung 21: Klassendiagramm: ObjectReference.....	41
Abbildung 22: Schiffsposition Client/Server - P2P - kein Lag.....	46
Abbildung 23: Schiffsposition Client/Server - lineare Interpolation - kein Lag.....	47
Abbildung 24: Schiffsposition Client/Server - P2P - 1s Lag.....	47
Abbildung 25: Schiffsposition Client/Server - lineare Interpolation - 1s Lag.....	48

Tabellenverzeichnis

Tabelle 1: Netzwerkframework Evaluierung.....25

Anlagenverzeichnis

Anlage 1 -Network Integration - Background Information
Anlage 2 - Network Integration - SRS
Anlage 3 - Network Integration - Analysis
Anlage 4 - Network Integration - Integration Plan
Anlage 4 - Network Integration - Integration Test

1 Einleitung

1.1 *Motivation*

Multiplayerspiele haben eine lange Geschichte hinter sich. Vom ersten „Computerspiel“ für mehr als einen Spieler, Tennis for Two von William Hunter, welches auf einem Oszilloskop gespielt wurde bis hin zu heutigen Spielen wie World of Warcraft oder Tabula Rasa, welche ausschließlich mit aktiver Internetverbindung gespielt werden können, war es ein langer Weg. Die Faszination welche ein Mehrspielermodus mit sich bringt ist dabei jedoch ungebrochen. Die eigenen Fähigkeiten mit anderen Spielern zu messen und/oder zusammen mit Freunden zu Spielen ist der Grund für die anhaltende Erfolgsgeschichte vieler der beliebtesten mehrspielerfähigen Computerspiele.

Dabei haben Multiplayerspiele einen steinigen Weg hinter sich und müssen auch heute noch viele Hürden nehmen, bis es soweit ist, dass man zusammen mit anderen realen Personen ein Spiel erleben kann.

Dies trifft im Besonderen dann zu, wenn die teilnehmenden Spieler geographisch weit voneinander entfernt sind, wie es in der Regel bei Verbindungen über das Internet der Fall ist.

1.2 *Problemstellung und -abgrenzung*

Eine Multiplayerunterstützung, die über das Internet funktioniert, stellt spezielle Anforderungen an jede Netzwerkengine. Die Umgebungsbedingungen sind nicht mit denen zu vergleichen, die in einem lokalen Netzwerk vorherrschen und bedürfen spezieller Verfahren um ein flüssiges Spielerlebnis für die Anwender zu ermöglichen.

Darüber hinaus birgt die Erweiterung einer existierenden Spieleengine zusätzlichen Schwierigkeiten in sich, welche bei einer von Grund auf neu erstellten Engine nicht zu berücksichtigen wären. Insbesondere trifft dies deshalb zu, da die zu Grunde liegende Spieleengine nicht strikt auf eine spätere Integration einer Netzwerkengine ausgelegt ist.

Da die Umsetzung einer Netzwerkunterstützung Änderungen an nahezu jedem Element der Spieleengine zur Folge hat, ist auch eine enge Abstimmung mit anderen am Gesamtprojekt beteiligten Entwickler unerlässlich.

1.3 *Ziel der Diplomarbeit*

Ziel der Diplomarbeit ist die Analyse von Techniken und Verfahren, die notwendig sind, um einen Mehrspielermodus über das Internet zu realisieren, aufzuzeigen welche spezifischen Probleme dabei auftreten und wie diese zu lösen sind.

Dazu wird eine Netzwerkengine für eine existierende Spieleengine entworfen, implementiert und gezeigt, wie die verschiedenen Techniken dazu dienen, dass die Netzwerkengine auch mit den Umgebungsbedingungen des Internets zu Rande kommt.

1.4 Aufbau des Dokumentes

Kapitel 2 liefert die Grundlagen, die notwendig sind, um die Diplomarbeit und die damit verbundenen Problemstellungen verstehen zu können. Das Vorgehen der Diplomarbeit sowie verwendete Software wird in Kapitel 3 behandelt, gefolgt von Kapitel 4, in welchem Techniken erläutert werden, die zur Kompensation von Lag eingesetzt werden können.

Kapitel 5 beschäftigt sich mit der durchgeführten Evaluierung der Netzwerkframeworks, während Kapitel 6 den folgenden Integrationstest beschreibt. Kapitel 7 erläutert die über die in den Anlagen hinausgehenden Entwurf und Realisierung der Netzwerkengine sowie der dazu gehörigen Komponenten.

Im vorletzten Kapitel wird die Verwendung des Netzwerksimulators beschrieben, bevor zum Abschluss gezeigt wird, wie die angewandten Techniken zu einer Reduzierung des wahrgenommenen Lags führen.

2 Grundlagen

2.1 Von X bis X4

1988 gegründet, entwickelte Egosoft noch Spiele für den Amiga. Bekannt wurde die Firma vor allem durch ihre Spieleserie „X – Beyond the Frontier“, welche auf dem Spielprinzip des beliebten Elite aufbaut.



Abbildung 1: Screenshot aus dem Spiel Elite

Angesiedelt im Sci-Fic-Universum ist Elite eine Mischung aus Wirtschafts- und Weltraumsimulation. Der Spieler übernimmt die Kontrolle über ein Raumschiff, treibt Handel, erledigt Missionen, nimmt an Raumkämpfen teil und erkundet ein riesiges Universum.

X – Beyond the Frontier greift dieses Spieleprinzip auf, erweitert es und bringt die gesamte Technik (Grafik, KI, UI, etc.) auf den aktuellen Stand. Der Spieler hat außerdem die Möglichkeit eigene Raumstationen zu bauen, mehr als nur ein Schiff zu kommandieren und sich Ruf und Ehre bei anderen Rassen zu verdienen.



Abbildung 2: X1 bis X4

Diese Entwicklung setzte sich über die beiden Sequels hinaus fort. Schöner Grafik, neue Gameplay-Features, abwechslungsreichere KI, etc. Allerdings fehlte bis einschließlich zu X3 die Möglichkeit mit/gegen andere Spieler anzutreten; ein Feature, welches von den Fans der Serie immer wieder gewünscht wurde. Mit X4 soll sich dies ändern und es möglich werden mit anderen X-Fans zusammen über das Internet Schlachten zu schlagen, Handelsimperien aufzubauen und sein Können als Pilot unter Beweis zu stellen.

2.2 Datenübertragung im Internet

2.2.1 Das OSI-Referenzmodell

Um den Transport von Daten über das Internet zu ermöglichen, werden die von einer Anwendung versandten Datenpakete vor der Übertragung durch zusätzliche Informationen erweitert. Diese Zusatzinformationen werden dann an einem Knotenpunkt ausgewertet und sorgen letztendlich dafür, dass das Paket beim Empfänger ankommt (oder auch nicht, wie sich später zeigen wird).

Das OSI-Referenzmodell gibt eine abstrakte Darstellung die zeigt, wie Daten durch Zusatzinformationen erweitert.

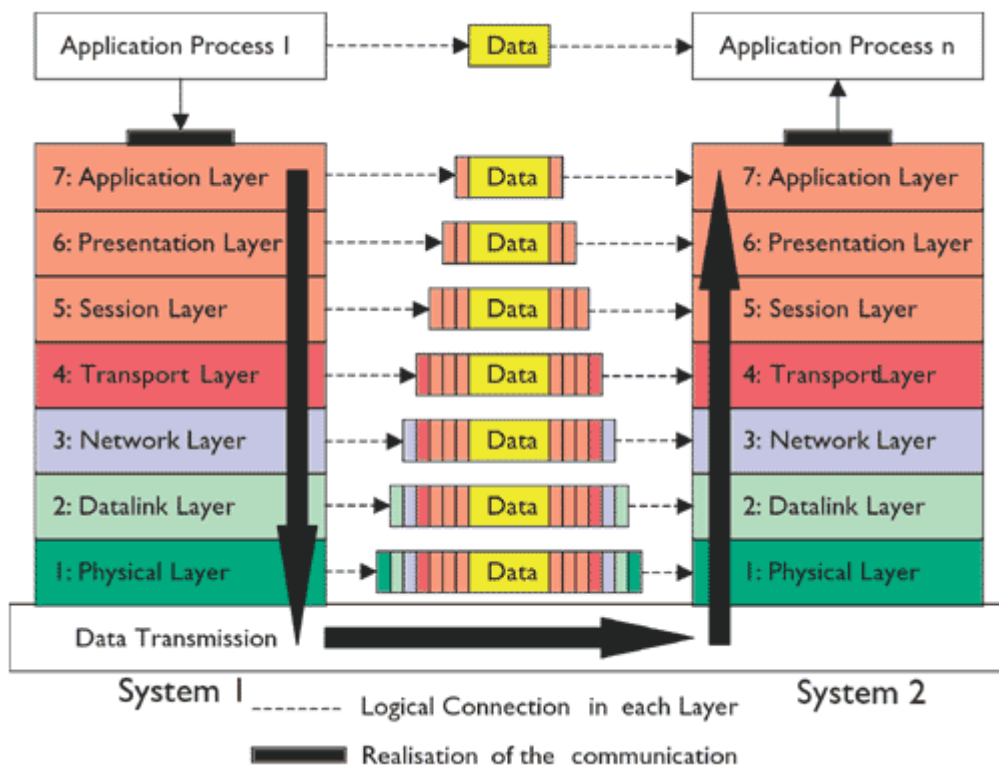


Abbildung 3: OSI-Referenzmodell

Daten, welche über ein Netzwerk an einen Empfänger geschickt werden sollen, durchlaufen dabei verschiedene Schichten. Jede Schicht fügt seine ganz spezifischen Headerinformationen hinzu und kapselt damit die Daten.

Beim Empfänger werden die Schichten in umgekehrter Reihenfolge wieder durchlaufen. Dabei verarbeitet jede Schicht lediglich die eigenen zuvor hinzugefügten Headerinformationen und reicht es gegebenenfalls an die übergeordnete Schicht weiter bzw., sendet die Daten an einen anderen Empfänger.

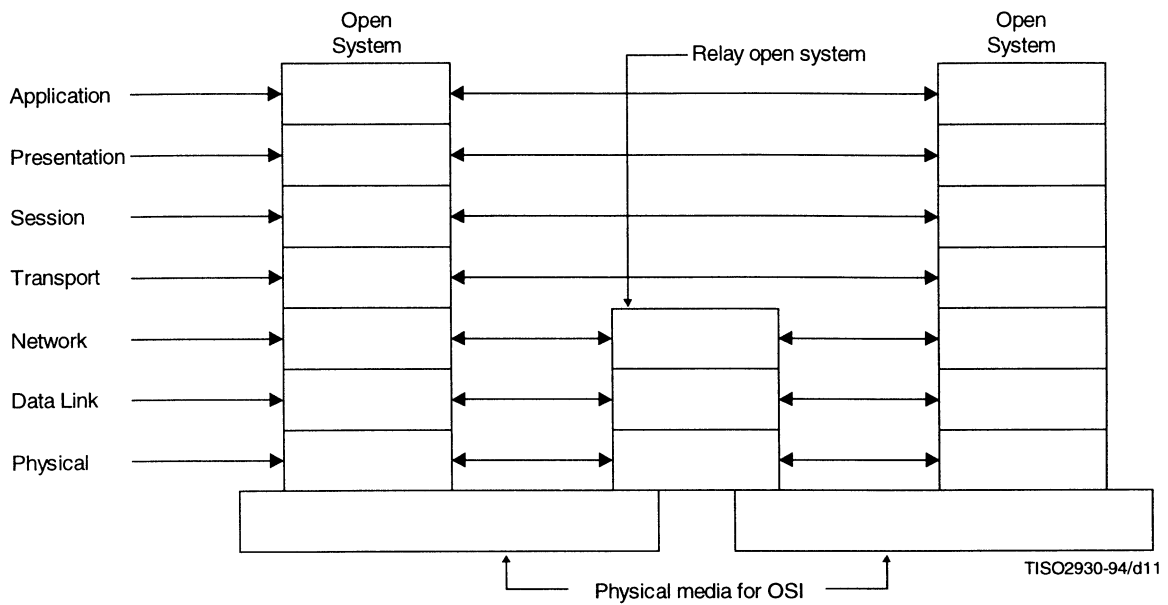


Abbildung 4: OSI-Referenzmodell (Router)

Angenommen wir haben ein Netzwerk bestehend aus 2 PCs, die über einen Router verbunden sind. Beim Senden durchläuft das Datenpaket zunächst sämtliche Schichten und wird an den Router übertragen. Dieser „entpackt“ nur die Headerinformationen bis zu der Schicht, die notwendig ist, um das Datenpaket weiterzureichen (im Fall eines Routers wäre dies Layer 1-3, wie in der obigen Abbildung). Die höheren Schichten werden vom Router nicht beachtet. Erst beim Empfänger werden wieder sämtliche Schichten durchlaufen, bis die Anwendung schließlich die Daten erhält.

Zur Umsetzung der verschiedenen Schichten kommen Protokolle zum Einsatz. Dabei hat jedes Protokoll eine ganz spezifische Aufgabe und kann einer Schicht im OSI-Referenzmodell zugeordnet werden.

Die wichtigsten Protokolle die bei der Entwicklung einer Netzwerkengine für ein Spiel werden im Folgenden genauer erläutert.

2.2.1.1 ATM – Sicherungsschicht (Layer 2)

Der Asynchronous Transfer Mode (ATM) wird heute bei den meisten ADSL-Breitbandanschlüssen verwendet. Ursprünglich wurde der ATM entwickelt, um eine Sprachübertragung über Links mit geringer Bandbreite übertragen zu können.

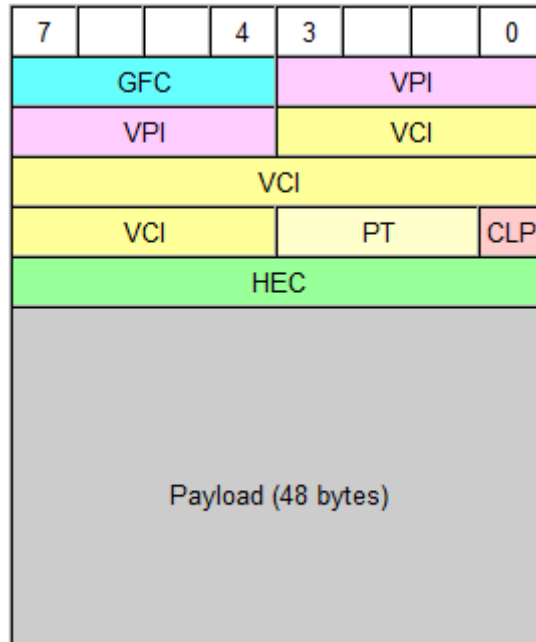


Abbildung 5: ATM-Header

Das ATM-Paket besteht aus einem 5 Byte großen Header und kapselt jeweils 48 Bytes an Daten.

Die wichtigsten Headerinformationen sind VPI (Virtual Path Identifier) und VCI (Virtual Channel Identifier). Zusammen identifizieren sie den nächsten Empfänger, an den das Datenpaket weitergereicht wird.

Darüber hinaus ist im HEC (Header Error Correction) ein CRC gespeichert um Übertragungsfehler der Headerinformationen identifizieren und einzelne Bit-Fehler korrigieren zu können.

2.2.1.2 IP – Vermittlungsschicht (Layer 3)

In der Vermittlungsschicht angesiedelt, befindet sich das Internet Protocol (IP). Das IP hat schon diverse Versionssprünge hinter sich. Aktuell ist die Version 4 (IPv4) noch die verbreitetste Variante. Der Nachfolger, Version 6 (IPv6), gewinnt jedoch immer mehr an Bedeutung.

Im Folgenden beschränken wir uns jedoch auf die Vorgängerversion, da diese momentan noch die häufiger anzutreffende ist.

+	Bits 0–3	4–7	8–15	16–18	19–31
0	Version	Header length	Type of Service (now DiffServ and ECN)	Total Length	
32	Identification			Flags	Fragment Offset
64	Time to Live		Protocol	Header Checksum	
96	Source Address				
128	Destination Address				
160	Options				
160 or 192+	Data				

Abbildung 6: IP-Header

Ein IP-Datenpaket besteht aus einem 20 oder 24 Byte großen Header und einem Datenpaket mit variabler Größe. Die maximale Größe eines gekapselten Datenpaketes beträgt dabei 65.535 Byte (2^{16} Bit = Total Length Headerinformationen).

In der Realität wird diese Größe jedoch eher selten erreicht. Insbesondere wird häufig auf das Fragmentierungssystem vom IP zurückgegriffen, um die Größe eines IP-Paketes an die Größe anzupassen, die das darunter liegende Protokoll versenden kann (z.B. 1.500 Byte beim Ethernet-Protokoll oder 9.180 Byte beim ATM-Protokoll).

Die wichtigsten Informationen, die der IP-Header enthält sind die IP Adressen (IPs) des Empfängers und Senders des Datenpakets.

2.2.1.3 TCP – Transportschicht (Layer 4)

Der vermutlich bekannteste Vertreter eines Protokolls der Transportschicht ist das Transmission Control Protocol (TCP).

Es bietet die zuverlässige sortierte Übertragung von Datenpaketen. Dies bedeutet, dass das Protokoll sicherstellt, dass gesendete Daten korrekt (d.h. fehlerfrei) und in der Reihenfolge, in der sie gesendet wurden, beim Empfänger ankommen. Dies kann unter Umständen dazu führen, dass Datenpakete, welche bei der Übertragung beschädigt wurden oder verloren gingen automatisch erneut übertragen werden.

Der TCP-Header ist 20 oder 24 Byte groß. Die Größe des gekapselten Datenpaketes ist variable.

Bit offset	Bits 0–3	4–7	8–15								16–31											
0	Source port										Destination port											
32	Sequence number																					
64	Acknowledgment number																					
96	Data offset	Reserved		CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window										
128	Checksum										Urgent pointer											
160	Options (optional)																					
160/192+	Data																					

Abbildung 7: TCP-Header

Durch die Angabe eines Ports, können verschiedene Anwendungen auf einem Host Daten empfangen und senden. Die Portnummer identifiziert dabei die Anwendung, die dem Datenpaket zugewiesen ist.

Neben den Angaben der Portnummer enthält der Header weitere Informationen, die vor allem dazu verwendet werden, um die sichere Übertragung der Daten zu gewährleisten.

2.2.1.4 UDP - Transportschicht (Layer 4)

+	Bits 0 - 15	16 - 31
0	Source Port	Destination Port
32	Length	Checksum
64	Data	

Abbildung 8: UDP-Header

Im Gegensatz zum TCP handelt es sich beim User Datagram Protocol (UDP) um ein unsicheres Übertragungsprotokoll. Im Unterschied zu TCP stellt UDP nicht sicher, dass versandte Daten auch beim Empfänger ankommen. Dies führt nicht nur zu einer wesentlich schnelleren Übertragung der Daten, da der gesamte Vorgang wie Empfangsbestätigungen, etc. entfällt, sondern auch zu einem kleineren Header von nur 8 Byte.

2.2.2 Lag

Als Lag bezeichnet man die Zeit, die ein Paket vom Sender bis zum Empfänger benötigt. Wie groß der Lag ist, wird durch verschiedene Faktoren beeinflusst.

2.2.2.1 Die Gesetze der Physik

„There is an old network saying: „Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed – you can't bribe God.“

David Clar, MIT

Der erste Faktor wird uns von der Physik diktiert: Die Geschwindigkeit mit der Daten übertragen werden können, ist durch die Lichtgeschwindigkeit beschränkt.

Dies ist in lokalen Netzwerken ein vernachlässigbarer Faktor, im Internet, bei dem ein Datenpaket durchaus um die ganze Welt versandt werden kann, jedoch relevant. Angenommen ein Paket würde einmal um die Erde geschickt, dann bräuchte es allein wegen der Einschränkung der Lichtgeschwindigkeit ca. 40 ms (12.000 km / 300.000 km/s).

Nun kann man damit argumentieren, dass selten eine Verbindung von Frankfurt nach Sydney aufgebaut wird, um ein Paket mehrere tausend Kilometer zurücklegen zu lassen, ist dies jedoch nicht immer nötig. Selbst beim Senden von Daten zwischen zwei geographisch nah beieinander liegende Hosts kommt es nicht selten vor, dass diese Daten erst über „Umwege“ ihr Ziel erreichen. Im Kleinen heißt das: Wenn man eine Verbindung zum Nachbarn aufbaut, wird das Paket unter Umständen erst vom eigenen Providernetz zu einem Knotenpunkt versandt und an das Providernetz des Nachbarn übergeben, bevor es das Ziel erreicht. Auch wenn geographisch nur einige Meter zwischen Sender und Empfänger liegen, so hat das Datenpaket bereits mehrere hundert km zurückgelegt.

Im Großen bedeutet dies, dass politische oder ökonomische Einflüsse dafür sorgen, dass selbst Verbindungen zu Hosts in einem Nachbarland große Umwege nehmen können.

Ein weiterer Grund, welcher dafür sorgen kann, dass Pakete längere Wege zurücklegen müssen, sind Unterbrechungen im Internet. Wenn ein Knotenpunkt ausfällt (bzw. überlastet ist), kann es dazu kommen, dass Pakete umgeleitet werden, was wiederum den Lag erhöht. Besonders fatal ist dies wenn der Ausfall gerade stattfindet und sich die Routingtabellen noch nicht auf den Ausfall „eingestellt“ haben. In diesem Fall gehen Router weiterhin davon aus, dass die Daten über den gerade ausgefallenen Router versandt werden können. Anstelle eine Ausweichroute schon früh zu wählen, wird erst sehr spät (oft erst direkt einen Hop vor dem ausgefallenen Router) eine andere Route gewählt.

2.2.2.2 Serialisierung

Der zweite Faktor, welcher sich auf den Lag auswirkt, ist in der Serialisierung von Daten zu finden. Egal ob die Daten über ein Ethernetkabel oder einen Satelliten gesendet werden, auf unterster Ebene werden diese in Bits aufgeteilt und Bit für Bit übertragen. Die Geschwindigkeit einer Verbindung (auch als Link bezeichnet) zwischen Sender und Empfänger wird dabei in Bit pro Sekunde (bps) angegeben. Ein Datenpaket von 1.500 Bytes (12.000 Bit) braucht beispielsweise 12 ms wenn es über eine Verbindung mit 1 Mbps übertragen wird.

Verglichen mit 56kbit Modemverbindungen ist der durch Serialisierung verursachte Lag etwas in den Hintergrund getreten. Bei einem Upstream von 33,6 kbps brauchten selbst 40 Bytes 9,5 ms, was u.a. für Positionsupdates in schnellen First Person Shootern relevant werden konnte.

2.2.2.3 Warteschlangen

Genauso wie es im richtigen Leben bei einer Serialisierung zu Warteschlangen kommen kann (z.B. im Supermarkt bei nur einer geöffneten Kasse reihen sich die Kunden hintereinander in einer Warteschlange an), so kann (und kommt) es auch bei der serialisierten Datenübertragung zu Warteschlangen, wann immer Multiplexing stattfindet. Multiplexing kommt in der Datenübertragung immer dann zum Einsatz wenn mehrere eingehende Links auf einen ausgehenden Link treffen. Zum Beispiel verwendet ein herkömmlicher Router, welcher in einem Haushalt mehreren PCs den Internetzugang zur Verfügung stellt, Multiplexing (eingehende Links = angeschlossene PCs, ausgehender Link = Verbindung zum Internetprovider). Hierbei kommt es zu Engpässen, wenn die benötigte Kapazität die verfügbare übersteigt. In dem Fall müssen Daten in einer Queue gepuffert werden und werden zeitverzögert übertragen, was den Lag um die Zeit erhöht, die die Daten in der Warteschlange verbringen.

Um zu vermeiden, dass dies zu einem Totalausfall einzelner Router führt, kommt sogenanntes statistisches Multiplexing zum Einsatz. Unter diesem Begriff verbirgt sich ein Verfahren, welches sicherstellt, dass im Mittel die benötigten Kapazitäten die verfügbaren nicht übersteigen. Dies verhindert jedoch nicht, dass es auch zu Spitzenauslastungen kommen kann, die kurzzeitig die verfügbare Kapazität übersteigt. In diesem Fall werden die Daten weiterhin in einer Queue gepuffert.

Neben dem oben beschriebenen Grund gibt es noch ein weiteres Szenario, welches zu durch Queueing verursachtem Lag führen kann. Kann über einen ausgehenden Link zu einem Zeitpunkt nur ein eingehender Host Daten übertragen, wie es z.B. beim IEEE802.11 b/g (WiFi Netzwerke) der Fall ist, müssen sendebereite Hosts warten, bis der Link wieder frei ist. In Experimenten verursachte diese Queueingwartezeit bei besonders hoch ausgelasteten Wireless-Netzwerken eine zusätzliche Verzögerung der Datenübertragung um 50-100 ms. [BOOK1]

2.2.2.4 Round-Trip-Time

Lag-Angaben entsprechen meist der sogenannten Round-Trip-Time (RTT). Dies ist die Zeit, die ein Datenpaket zum Empfänger und wieder zurück zum Sender benötigt. Somit ist der angezeigte Lag in etwa doppelt so groß wie der eigentliche Lag. Die Verdoppelung des Lags ist nur eine Annäherung an die RTT, da der Sende- und Empfangslag durchaus variieren kann. Gründe sind hierfür u.a. die Möglichkeit, dass die verwendete Route beim Senden und Empfangen nicht identisch sein muss, oder (wesentlich häufiger), dass der Link für den Up- und Downstream nicht gleich „groß“ ist. Dies war zu Zeiten analoger Modems der Fall (z.B. Downstream 56kBit/s, Upstream 33,6 kBit/s) und ist es auch heute noch mit ADSL (je nach Art z.B. 1 MBit/s Downstream und 128kBit/s Upstream). Dies hat zur Folge das gleichgroße Datenpakete einen unterschiedlichen Serialisierungslag beim Senden und Empfangen erfahren.

Der Grund, eher die Round-Trip-Time anzugeben, statt des eigentlichen Lags liegt darin, dass diese Information bei Spielen eher irrelevant ist. In der Regel ist es wichtig zu wissen, wie schnell ein Spieler vom Server auf eine Aktion eine Reaktion bekommt. Dies entspricht in etwa (abzüglich der Verarbeitungszeit der Daten) der RTT. Darüber hinaus bietet das Internet Control Message Protocol (ICMP) eine einfache Möglichkeit die RTT zu bestimmen. So liefern auch Hilfsanwendungen verschiedener Betriebssysteme wie z.B. der „ping“-Befehl unter Linux und Windows eine einfache Möglichkeit für den Endanwender die RTT zu einem Host anzuzeigen.

Hierbei ist jedoch anzumerken, dass die mit Hilfe der Echo-Funktion des ICMPs bestimmten RTT nicht unbedingt der RTT entspricht, welche im Spiel vorherrscht. Dies liegt daran, dass ICMP-Pakete anders im Internet behandelt werden, als Datenpakete von Spielen/Anwendungen (welche meist UDP oder TCP verwenden). Diese können einen durchaus höheren Lag erfahren.

Aus den oben genannten Gründen verwenden auch wir den Begriff „Lag“ als Zeit für den Hin- und Rückweg eines Datenpaketes.

2.2.2.5 Lag, Latenz und Bandbreite

Die Begriffe Lag, Latenz und Bandbreite werden in der Literatur und auch von Spielern wild durcheinander geworfen. So definiert z.B. die englische Seite von Wikipedia Lag als „symptom where result of an action appears later than expected“, während Latenz als „the time taken for a packet of data to be sent from one application, travel to, and be received by another application“ festgelegt wird. Andere Seiten verwenden Lag und Latenz oder Latenz und Bandbreite als Synonyme. Häufig wird auch lediglich einer der Begriffe verwendet.

Wir verwenden im weiteren Verlauf der Arbeit Latenz entsprechend der Definition von Dr. Henry S. Fortuna von der University of Abertay Dundee.

Demnach berechnet sich die Latenz wie folgt: Lag + Zeit für die Generierung des Paketes beim Sender + Zeit für die Verarbeitung des Pakets beim Empfänger und Generierung eines Antwortpaketes.

Als Bandbreite verstehen wir lediglich die Kapazität eines Links, welche in Bit/s angegeben wird.

2.2.3 Lag ist nicht konstant (Jitter)

Der Lag im Internet ist selbst dann nicht konstant, wenn die Größe von versandten Datenpaketen sowie sich Sender und Empfänger des Datenpaketes nicht ändern. Neben der offensichtlichen aber eher unüblichen Ursache, dass sich eine Datenroute verändert, was geschehen kann, wenn z.B. ein Knotenpunkt plötzlich ausfällt, sind vor allem Warteschlangen die Ursache für die Varianz des Lags.

Angenommen wir haben 2 Rechner, die sich über einen Router eine Internetverbindung teilen. Der Link ins Internet beträgt 128 kbps. Rechner 1 sendet nun in Intervallen von 40 ms 80 Byte große Pakete über das Internet. Diese benötigen 5 ms für die Übertragung.

$$\begin{aligned} 80 \text{ Bytes} * 8 \text{ Bit/Byte} &= 640 \text{ Bit} \\ 640 \text{ Bit} / 128 \text{ kbps} &= 5 \text{ ms} \end{aligned}$$

Da der Serialisierungslag von 5 ms kleiner ist als das Sendeintervall von 40 ms, ist die Leitung wieder „frei“, sobald das nächste Paket gesendet werden kann. Dies führt zu einem konstanten Lag von 5 ms.

Nun kommt Rechner 2 hinzu und sendet in 500 ms Intervallen 1.500 Byte große Datenpakete, welche für die Übertragung ca. 94 ms benötigen.

$$\begin{aligned} 1.500 \text{ Bytes} * 8 \text{ Bit/Byte} &= 12.000 \text{ Bit} \\ 12.000 \text{ Bit} / 128 \text{ kbps} &= 94 \text{ ms} \end{aligned}$$

Nun kann es im schlimmsten Fall dazu kommen, dass Rechner 1 und Rechner 2 gleichzeitig ihr Datenpaket auf den Weg schicken und das Datenpaket von Rechner 2 zuerst versandt wird. In diesem Fall werden 3 Datenpakete von Rechner 1 zunächst in der Warteschlange gespeichert. Wenn nach 94 ms der Link wieder frei ist, wird das erste der wartenden Datenpakete auf den Weg geschickt. Beim Empfänger kommt dieses nun mit einer Verzögerung von 99 ms an.

$$94 \text{ ms (Zeit in Warteschlange)} + 5 \text{ ms (Zeit für Serialisierung)} = 99 \text{ ms}$$

Das zweite Datenpaket hat einen Lag von 64 ms und das 3. einen Lag von 29 ms.

$$(99 \text{ ms} - 40 \text{ ms}) \text{ (Zeit in Warteschlange)} + 5 \text{ ms (Zeit für Serialisierung)} = 64 \text{ ms}$$

$$(64 \text{ ms} - 40 \text{ ms}) (\text{Zeit in Warteschlange}) + 5 \text{ ms} (\text{Zeit für Serialisierung}) = 29 \text{ ms}$$

Diese Art von Verzögerung kann überall auf dem Weg zwischen Sender und Empfänger auftreten und führt zu der Varianz im Lag, welche kurz als Jitter bezeichnet wird und in ms angegeben wird:

$$\text{Jitter} = \max \text{Lag} - \min \text{Lag}$$

Neben den oben beschriebenen Ursachen kann es außerdem zu Jitter kommen, wenn die Größe von ansonsten konstant großen Datenpaketen auf dem Weg zum Empfänger verändert werden, wie es beim IP over PPP/High-Level Data Link Control (HDLC) Verfahren der Fall ist.

Hierbei werden alle Vorkommen des Bytes 0x7E durch die beiden Bytes 0x7D und 0x5E maskiert. Dies führt dazu, dass ursprünglich gleichgroße Datenpakete mit unterschiedlichem Inhalt plötzlich unterschiedlich groß sind. Wird das veränderte Datenpaket nun über einen seriellen Link geschickt, variiert damit auch der Serialisierungslag.

Würde man den Lag nun ständig neu berechnen, führt eben Jitter dazu, dass sich der angezeigte Wert ständig ändert. Deshalb gibt man den Lag-Wert in der Regel als Mittelwert über eine bestimmte Zeit an.

2.2.4 Packetloss

Im schlimmsten Fall werden Pakete nicht nur stark verzögert empfangen, sondern können auch komplett verlorengehen und nie beim Empfänger ankommen. Dies wird als Packetloss bezeichnet.

Paketverluste können überall im Internet auftreten. Allerdings ist es heutzutage eher die Ausnahme, statt die Regel, dass Daten verlorengehen. Zu Zeiten von analogen Modems hingegen lag die häufigste Ursache für den Datenverlust darin, dass empfangene analoge Signale nicht mehr korrekt demoduliert werden konnten, da sich das Signal zu stark verschlechtert hatte. Paketverluste von über 2% waren somit keine Ausnahme.

Obwohl sich dies mit der Einführung von ISDN und Breitbandverbindungen sowie besseren Netzstrukturen wesentlich verbessert hat, gibt es auch heute noch Paketverluste, auch wenn sich diese nun eher im Bereich weit unter 2% bewegen.

Neben den Problem bei der Umwandlung der analogen in digitale Signale treten Paketverluste heutzutage vor allem dann auf, wenn ein Knotenpunkt im Netz ausfällt. Dies kann dazu führen, dass mehrere Sekunden oder sogar minutenlang Datenpakete am ausgefallenen Knotenpunkt verlorengehen, bis sich das Netz auf den ausgefallenen Knotenpunkt eingestellt hat und eine Ausweichroute gewählt wurde.

Die dritte Möglichkeit, welche zu Packetloss führt, tritt dann auf, wenn einzelne Router im Netz überlastet sind. Ist die Warteschlange eines Routers voll, werden alle weitere Daten die am Router ankommen einfach verworfen.

Um es gar nicht erst zu einem Überlauf der Queues in Routern kommen zu lassen, gibt es das sogenannte „Active Queue Management“. Unter diesem Begriff verbirgt sich ein System welches Datenpakete bereits verwirft, bevor die Queue eines Routers voll ist. Dies hat den Hintergrund, dass dadurch TCP-basierte Anwendungen die Datenrate mit der Pakete versandt werden, drosseln können, bevor es zu einem Datenstau kommt.

2.2.5 Auswirkungen von Lag auf Spiele

Welche Auswirkungen Lag auf ein Spiel hat, ist von dessen Netzwerkengine abhängig. Generell kann gesagt werden, je intoleranter eine Netzwerkengine für Lag ist, desto größer sind auch die spürbaren Auswirkungen, die der Spieler erfährt. Zwei kurze Videos demonstrieren die Auswirkung von Lag in zwei aktuellen MMORPGs.

Im WoW-Demovideo, dass die gewählten Aktionen (Angriff auf den Raptor mit der Waffe) genauso verzögert ausgeführt wird, wie die Angriffe der NPCs auf den Charakter. Die Animationen (sich bewegende Raptoren, sowie die Angriffsanimationen selbst) werden hingegen nicht von dem Lag beeinflusst, da sie nur auf dem Client ausgeführt werden und keine Datenübertragung erfordern.

Charakteristisch für den Lag in WoW ist ebenfalls, dass die Angriffsaktionen in Intervallen ausgeführt werden. Dies ist darauf zurückzuführen, dass jede einzelne Angriffsaktion eine Bestätigung vom Server benötigt, bevor sie der Client diese Aktion anzeigt.

In Tabula Rasa verhält es sich ähnlich. Auch hier werden die Auswirkungen eines Angriffs auf einen NPC nur schubweise umgesetzt, wie an der stockenden Rüstungs- und Lebensanzeige des NPCs zu sehen ist. Auch dies kann darauf zurückgeführt werden, dass Angriffsaktionen im Gegensatz zu Bewegungsaktionen (die sowohl in WoW als auch in Tabula Rasa bei vorhandenem Lag ohne spürbare Verzögerung umgesetzt werden) eine Bestätigung von Seiten des Servers abwarten.

2.3 Netzwerkframeworks

Bei der Integration einer Netzwerkunterstützung in eine beliebige Anwendung gibt es bestimmte Funktionen die unabhängig von der spezifischen Anwendung immer identisch sind.

Zum Beispiel muss eine Verbindung zwischen den beteiligten Hosts hergestellt werden und eine Datenübertragung ermöglicht werden. Hinzu kommt, dass wenn man statt TCP lieber UDP als Übertragungsprotokoll einsetzen will, Funktionalität hinzugefügt werden muss, will man nicht gänzlich auf eine Möglichkeit zur sicheren Datenübertragung verzichten.

Darüber hinaus gibt es viele speziellere Anwendungsfälle, die in verschiedenen Anwendungen (und im Speziellen auch in Spielen) immer wiederkehrend sind. Dies betrifft unter anderem eine Chatfunktionalität, auf welche in praktisch keinem Multiplayerspiel verzichtet werden kann oder die Möglichkeit Dateien zu übertragen.

Da dies auch von diversen anderen Programmieren und Firmen erkannt wurde, finden sich inzwischen diverse Anbieter von Netzwerkframeworks auf dem Markt. Diese stellen die grundlegenden Funktionen, die jede Netzwerkengine benötigt, zur Verfügung und ersparen somit dem Entwickler sich um die Implementierung auf Basis der vom Betriebssystem zur Verfügung gestellten Netzwerk API zu kümmern.

Abhängig vom Framework werden auch erweiterte Features wie die Übertragung von Sprache unterstützt. Besonders nützlich für die objektorientierte Programmierung sind auch integrierte Systeme, um auf Objektbasis Daten zu synchronisieren.

Eine Netzwerkengine eines Computerspiels kann außerdem darauf aufbauen, dass UDP basierte Netzwerkframeworks Möglichkeiten anbieten, um den Datentransfer detaillierter steuern zu können und dennoch nicht auf eine sichere Datenübertragung verzichten zu müssen.

2.4 Die X4 Game Engine

Die für den Entwurf und die Implementierung relevanten Systeme der Spieleengine sind in Anlage 1 Kapitel 2.4 beschrieben.

3 Vorgehen

Im Folgenden wird das Vorgehen im Projekt näher erläutert sowie eine Übersicht über die verwendete Software gegeben

3.1 *Evaluierung der Netzwerkframeworks*

Es ist klar, dass Netzwerkframeworks die Integration eines Netzwerksystems in eine Anwendung wesentlich erleichtern können. Viele Kleinigkeiten mit denen sich der Entwickler herumschlagen muss, wenn er eine Netzwerkengine direkt auf dem vom Betriebssystem zur Verfügung gestellten APIs (z.B. WinSock bei Windows) aufbauen möchte, sind sehr zeitaufwendig und Fehler in diesem Bereich sehr schwer zu entdecken. Darüber hinaus ist eine plattformunabhängige Netzwerkengine mit zusätzlichem Aufwand verbunden, da die Netzwerkimplementierung Betriebssystem abhängig ist und sich somit eine Implementierung unter Linux oder für Konsolen stark von der unter Windows unterscheiden kann.

Aus diesem Grund wurden verschiedene Netzwerkframeworks evaluiert, und für die Diplomarbeit das Framework verwendet, welches bei der Evaluierung am besten abgeschnitten hat (siehe Kapitel 5).

3.2 *Integration der Netzwerkengine in die Spieleengine*

Um die Machbarkeit der Integration des gewählten Netzwerkframeworks in die Spieleengine zu demonstrieren und Probleme, die während der Integration des Frameworks auftreten können zu identifizieren wurde ein erster Prototyp implementiert. Dieser Prototyp bestand aus einer rudimentären Implementierung eines Server- und eines Clientmoduls, welche die grundlegenden Funktionen des Frameworks testeten und demonstrierten, in welcher Art die Implementierung des Netzwerkes durchgeführt werden kann (siehe Kapitel 7).

3.3 *Aufbau eines Testnetzwerkes*

Ein großes Problem bei der Entwicklung von Netzwerkprogrammen ist, dass Testen dieser unter realistischen Bedingungen. Würde man auf 2 PCs, welche sich den gleichen Internetzugang teilen einen Test durchführen, würde man den Test allenfalls im Netz des eigenen Providers, nicht aber im Internet durchführen.

Aus diesem Grund wurde ein Weg gesucht in einem lokalen Netz die relevanten Probleme, die das Internet mit sich bringt, zu simulieren. Die gewählte Methode besteht aus einem Debian Server auf welchem mit Hilfe der Anwendung „Network Simulator 2“ der Lag im Internet simuliert wird (siehe Kapitel 9).

3.4 *Analyse der Systeme zur Lagkompensierung*

Diverse Quellen wurden untersucht (vor allem frei verfügbare Artikel im Internet), die von Netzwerkframeworks eingesetzten Methoden zur Lagkompensierung analysiert diverse Quellen zu Rate gezogen um Methoden aufzuzeigen, wie der Lag im Internet kompensiert werden kann. Im Zuge dieser Analyse wurde ein Dokument erstellt, welches die grundlegenden Problematiken sowie die relevanten Systeme der X4-Engine aufzeigt. Dieses Dokument diente als Basis für den Entwurf der Netzwerkengine und wurde den anderen Entwicklern zur Verfügung gestellt (siehe Anlage 1).

3.5 Entwurf der Netzwerkengine

Basierend auf den durchgeführten Recherchen wurde eine Software Anforderungsanalyse durchgeführt, um die Rahmenbedingungen der Netzwerkengine abzustecken (siehe Anlage 2).

Gleichzeitig wurde eine Umfrage im offiziellen Forum durchgeführt, in der Spieler der bisherigen X1-X3 Spiele gefragt wurden, wie sie sich eine Multiplayerunterstützung eines möglichen X3 Nachfolgers vorstellen würden. Die Ergebnisse dieser Umfrage flossen in das SRS-Dokument sowie die weitere Planungsphase ein (siehe Kapitel 10).

Anhand der SRS sowie der durchgeführten Analyse wurde eine Entwurf für die Netzwerkengine entwickelt. Dieser beschreibt die grundlegende Integration der Netzwerkengine sowie notwendige Anpassungen existierender Systeme (siehe Anlage 3).

3.6 Integration der Netzwerkengine

Basierend auf dem Entwurf wurde eine schrittweise Anpassung der Spieleengine durchgeführt, um die grundlegenden Systeme der Netzwerkengine zu integrieren, ohne andere Entwickler bei ihrer Arbeit zu behindern. Dafür wurde ein Integrationsplan entworfen, der für jede Iteration im Detail die durchzuführenden Änderungen beschreibt (siehe Anlage 4).

3.7 Abschließende Analyse

Um zu demonstrieren, dass die Netzwerkengine funktionsfähig ist und die integrierten Verfahren erfolgreich den Lag kompensieren, wird eine abschließende Analyse durchgeführt. Dazu werden Testmodule entwickelt, die ein einfaches Spieleuniversum generieren und dieses von einem Server auf einen Client überträgt und synchronisiert (siehe Kapitel 11).

3.8 Verwendete Software

Dieses Kapitel gibt eine Übersicht über die Software, die während der Diplomarbeit eingesetzt wurde.

3.8.1 Visual Studio (VS)

Für die Entwicklung von X4 wird Visual Studio 2005 SP1 eingesetzt. Diese IDE dient als Plattform für das Coden, Debuggen und Compilieren/Linken der Spieleengine. Darüber hinaus bietet es eine direkte Integration von VSS (siehe unten).

Der Nachfolger, Visual Studio 2008, erschien am 19. November 2007. Eine Portierung des Projektes auf die neue Version wurde jedoch nicht durchgeführt, da keine für die Entwicklung relevanten Features hinzugekommen sind und die aktuelle 2008er Version noch unter einigen kleineren Fehlern leidet.

Eines dieser neuen Features ist der „Class Viewer“. Dieser erlaubt es aus dem Quellcode UML-Klassendiagramme zu generieren. Zwar ist dieses Feature kein ausreichendes Argument um die komplette Entwicklung auf VS 2008 umzustellen, für die Dokumentation (im Besonderen für die Diplomarbeit) erspart der Class Viewer jedoch sehr viel Arbeit, weswegen zur Generierung dieser Diagramme VS 2008 eingesetzt wurde.

3.8.2 Virtual Assist X

Ein Problem mit dem Visual Studio zu kämpfen hat, sind Unzulänglichkeiten des integrierten IntelliSense-Systems. Dieses System soll dem Anwender dabei helfen effizienter zu Programmieren, indem es ihm z.B. die Parameterliste einer verwendeten Funktion anzeigt, die Möglichkeit bietet direkt zu einer Funktionsdefinition bzw. -deklaration zu springen, eine Liste aller Methoden und Variablen eines verwendeten Objektes anzeigt, etc.

Diese Funktionalität erspart dem Programmierer viel Zeit, da Tippfehler vermieden werden und das Nachschlagen eines Funktionsnamens nahezu überflüssig wird.

Das Problem dieses Systems ist jedoch, dass es auch in VS 2008 noch instabil läuft und Fehler enthält.

Virtual Assist X greift hier an, ersetzt als AddIn für Visual Studio weitestgehend die Funktionalität von Intellisense und bietet darüber hinaus erweiterte Features wie Highlighting von Bezeichnern, Refactoring, eine Rechtschreibprüfung, etc.

3.8.3 Visual Source Safe (VSS)

Als Versionsverwaltungssystem kommt während der Entwicklung VSS zum Einsatz. VSS bietet eine gute Integration in Visual Studio, ist aber bei der Arbeit über den VPN-Zugang sehr langsam und macht ein Arbeiten von außerhalb des Büros nahezu unmöglich. Dieses Manko wird durch die Verwendung von VSSConnect behoben.

3.8.4 VSSConnect

VSS ist um das Filesharing-System von Windows herum designt. Dies verursacht, dass entfernte Zugriffe über das Internet sehr langsam sind und dadurch das Arbeiten mit VS sehr erschwert wird.

VSSConnect integriert sich direkt in VS und ersetzt das langsame Übertragungsverfahren von VSS durch die Verwendung eines auf TCP aufbauenden Verfahrens. Dadurch wird die mangelnde Performance von VSS nahezu komplett eliminiert.

3.8.5 Debian

Die Linuxdistribution Debian wird verwendet, um einen Server aufzubauen, welcher zur Simulation von Lag im lokalen Netzwerk eingesetzt wird.

3.8.6 Network Simulator

Mit Hilfe des Network Simulator 2 (NS2) kann man verschiedene Netzwerke simulieren. Dadurch lassen sich reproduzierbare Tests erstellen, was in einem realen Netz nicht möglich ist. Insbesondere trifft dies auf das Internet zu, dass sich die Einflussfaktoren ständig ändern.

NS2 erlaubt es in TCL geschriebene Skripte auszuführen, die dann ein Netzwerk und damit verbundene konfigurierbare Umgebungsbedingungen simuliert. Im Normalfall werden dafür simulierte Datenquellen und Datensinken erstellt. Allerdings bietet NS2 auch die Möglichkeit Daten von/an ein(em) Netzwerkinterface zu empfangen/sendern.

3.8.7 Network Animator (NAM)

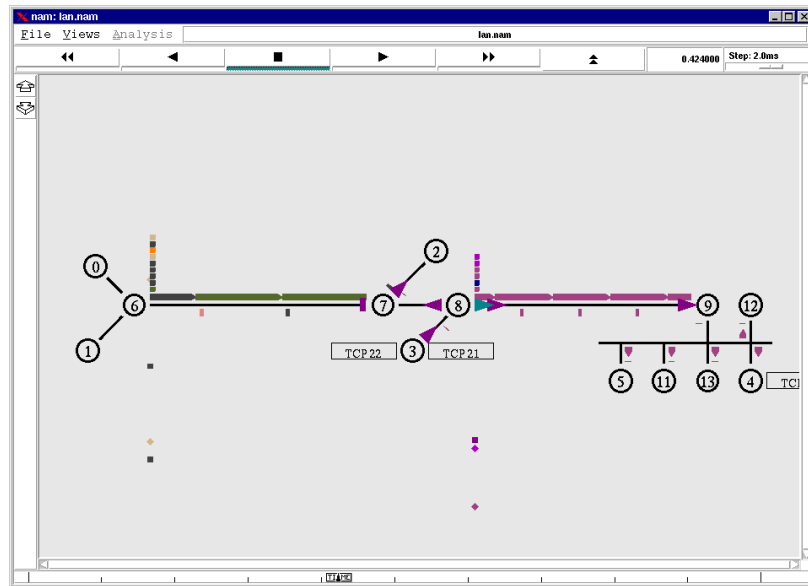


Abbildung 9: Network Animator Screenshot

Mit NS2 durchgeführte Simulationen lassen sich in Trace-Files ausgeben. Diese Dateien können nach dem Test dann von NAM eingelesen und angezeigt werden. Dadurch lassen sich nicht nur die Tests visualisieren sondern Fehler im Testskript nachvollziehen können.

3.8.8 Wireshark

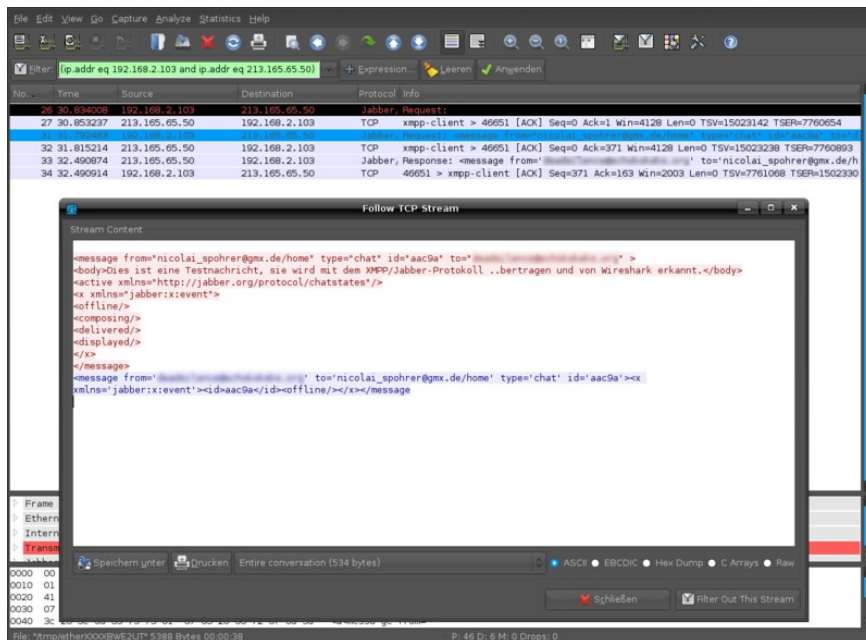


Abbildung 10: Wireshark Screenshot

Anhand von Wireshark lässt sich der Datentransfer der durch einen Server geleitet wird auf unterster Ebene loggen und analysieren. Die Pakete werden einzeln angezeigt und können dadurch ausgewertet werden. Primär wurde Wireshark verwendet, um Fehlern im NS2-Skript auf die Spur zu kommen.

3.8.9 Fraps

Für die Erstellung der Demonstrationsvideos wurde Fraps in Version 2.9.4 eingesetzt. Mit Hilfe dieser kleinen Anwendung kann man Videos von auf DirectX oder OpenGL basierten Spielen erfassen und dadurch die dargestellten Szenen in ein Videofile umwandeln.

Das Ergebnis ist eine AVI-Datei, welches die erfasste Szene im unkomprimierten AVI-Format enthält.

3.8.10 Adobe Premiere Pro



Abbildung 11: Adobe Premiere Pro Screenshot

Um die mit Fraps gemachten client- und serverseitigen Aufnahmen gegenüberzustellen und so die Auswirkungen des Lags zeigen zu können, ist ein Videoschnittprogramm nötig. Für diesen Zweck wurde Adobe Premiere Pro in der Version 3.0.0 eingesetzt, mit dessen Hilfe jeweils zwei Videos (Client- und Serveraufnahmen) zu einem zusammenzufassen wurden. Dabei war zu beachten dass der Timestamp beider Videos übereinstimmt, da eine Abweichung andernfalls als Lag interpretiert werden könnte.

4 Techniken zur Reduzierung der Auswirkungen von Lag

Die Techniken zur Reduzierung von Lag und deren Auswirkungen sind in Anlage 1 Kapitel 2.3 erläutert.

5 Evaluation der Netzwerkframeworks

Inzwischen gibt es eine ganze Hand voll Anbieter verschiedener Netzwerkframeworks, welche alle unterschiedliche Features bieten, sich in ihren Lizenzvereinbarungen unterscheiden und unterschiedlich gut supportet werden.

Um zu untersuchen, welche der verfügbaren Netzwerkframeworks sich für die X4-Netzwerkengine eignen, wurde eine Evaluierung in 2 Phasen durchgeführt.

5.1 Phase 1: Basisevaluierung

Die verfügbaren Netzwerkframeworks wurden zunächst dahingehend untersucht, wie hoch die zu erwartenden Kosten sind, welche relevanten Features sie bieten und ob sichergestellt werden kann, dass mögliche Fehler innerhalb der Engine bzw. fehlende Features nachträglich behoben/integriert werden können.

Name	Author / Firma	Lizenz	Features	Support
Hawk NL	Hawk Software	GPL2		inaktiv seit 2004
Net-Z	Quazal	pro Einheit	Client/Server und P2P eigene Skriptsprache zur Beschreibung von Netzwerkobjekten	aktiv, kommerzieller Support
RakNet	Jenkins Software	\$100	siehe unten	aktiv
TinCat	instance four	kommerziell	basiert auf TCP, mit rudimentärer UDP- Unterstützung	aktive, kommerzieller Support
Torque Network Library	Garage Games	GPL	UDP basiertes Framework	inaktiv seit 2005
ZoidCom	Jörg Rüppel	kommerziell	siehe unten	aktiv

Tabelle 1: Netzwerkframework Evaluierung

5.1.1 Abgelehnte Netzwerkframeworks

5.1.1.1 HawkNL:

Da seit 2004 keine neue Version erschienen und der Support praktisch nicht vorhanden ist, wurde HawkNL aussortiert. Es wurde davon ausgegangen, dass, obwohl der Sourcecode frei verfügbar ist, eine notwendige Einarbeitungszeit in keinem Verhältnis zu dem erwarteten Nutzen stünde.

5.1.1.2 Net-Z:

Da für dieses Produkt Lizenzgebühren pro verkaufter Einheit anfallen und es nach ersten Recherchen nicht abzusehen ist ob und in wie fern die integrierte Skriptsprache mit der Integration einer Netzwerkengine in X4 kompatibel ist, wurde entschieden auch dieses Framework nicht in die nähere Auswahl zu nehmen.

5.1.1.3 TinCat

Da TinCat primär auf TCP aufbaut und dies von einen Nachteil gegenüber UDP-basierten Netzwerkframeworks darstellt, sowie die Tatsache, dass für TinCat relevante Lizenzgebühren anfallen würden, wurde es zunächst aussortiert.

5.1.1.4 Torue Network Library

Obleich die Torue Network Library unter der GPL vertrieben wird und auf UDP aufbaut, müsste man sich direkt mit dem Sourcecode vertraut machen, um mögliche Fehler zu beheben und fehlende Features selbst einzubauen, da der Support des Frameworks seit 2005 praktisch nicht mehr existent ist. Aus diesem Grund fiel auch TorueNet durch das Raster.

5.2 Phase 2: Featureevaluierung

Nach der ersten Phase sind 2 Netzwerkframeworks übrig geblieben: ZoidCom und RakNet. Beide Frameworks werden im Folgenden detailliert untersucht und die verschiedenen Features, die diese unterstützen gegenübergestellt.

	RakNet	ZoidCom
Integrated Voice Communication	✓	X
Automated Endian Swapping	✓	✓
NAT punch-through	✓	X
Object Replication	✓ (semi-automatically)	✓
Object Synchronisation	✓ (semi-automatically)	✓
Auto Patcher	✓	X
Data Encryption	✓	X
DeadReconing / Prediction Algorithms	X	✓
Compression	✓	X
Supports RPCs	✓	X
Timestamp support	✓	✓
Priority System	simple (3 linear states)	dynamic
Reliability System	(un-)reliable, sequenced (32 streams), ordered (32 streams)	✓
Packet Creation	developer creates the packet individually	object oriented
Garbage Collection	X	✓ (with some exceptions)
Network IDs	✓	✓
Password support	✓	X
Supports large packages	✓	X
File transfer support	✓	✓
Multicast support	X	X
Host lookup	✓	✓
Language level	low level layer	high level coding

Abbildung 12: Evaluierung RakNet/ZoidCom

5.2.1 RakNet

RakNet ist vor allem für die Integration in kleinere Spiele, wie sie zum Beispiel von Hoobyprogrammieren und/oder der OpenSource-Community betrieben werden, ausgelegt. Aus diesem Grund bietet es unzählige Features, auf die die Entwickler solcher Spiele zurückgreifen können. Einige der Beispiele sind eine integrierte Unterstützung zur Sprachkommunikation, eine Patch-Funktion um Updates für das Spiel automatisch zu übertragen und die Unterstützung für die Übertragung ganzer Dateien.

Allerdings wird momentan angenommen, dass keines dieser Features von X4 benötigt wird.

Dennoch bietet RakNet einige Features an, die für X4 durchaus eingesetzt werden könnten. Dazu zählt die Möglichkeit Daten zu verschlüsseln, die Unterstützung von NAT punch-through sowie die Funktionalität, die übertragenen Daten zu komprimieren.

Nachteile gegenüber ZoidCom liegen jedoch im der Integration des Objekt orientierten Bereiches (Object Replication und Object Synchronisation), die etwas kompliziertere Generierung von Datenpaketen, sowie eine fehlende Garbage Collection, weshalb sich der Entwickler selbst um die Freigabe von nicht mehr benötigtem Speicher kümmern muss.

5.2.2 ZoidCom

ZoidCom kann vor allem durch eine sehr gute Dokumentation sowie eine sehr gute Unterstützung im objekt-orientierten Bereich punkten. Der Ansatz, den ZoidCom verwendet verspricht, dass die Umsetzung einer Netzwerkengine auf Basis von ZoidCom wesentlich einfach sein sollte als es mit RakNet zu erwarten ist.

Nachteile gegenüber RakNet sind vor allem darin zu finden, dass ZoidCom einige interessante Features vermissen lässt. So bietet das Framework von sich aus keine Möglichkeit Daten zu komprimieren oder zu verschlüsseln und auch NAT punch-through, welches für das Umgehen einiger Firewall einschränkungen nützlich ist, wird nicht direkt unterstützt.

Der wohl größte Nachteil von ZoidCom liegt jedoch darin, dass der Sourcecode nicht open source ist. Damit ist das Auffinden von Fehlern teilweise enorm umständlich und zeitaufwendig. Die Verwendung von ZoidCom im endgültigen Produkt ist demnach nur zu empfehlen, falls eine Lizenzvereinbarung auch den Zugriff auf den Sourcecode beinhaltet.

5.3 Die Wahl des Netzwerkframeworks

Eine Entscheidung für eines der beiden Netzwerkframeworks fällt insbesondere deshalb schwer, weil nicht abzusehen ist, welche Funktionen im Einzelnen von der Netzwerkengine benötigt werden. So kann es zum Beispiel sein, dass eine Verschlüsselung der Daten überflüssig ist oder größere Pakete, mit denen ZoidCom Probleme hat, gar nicht versandt werden müssen.

Auch ist zum jetzigen Zeitpunkt nicht abzuschätzen, auf welche Probleme man bei der Verwendung eines der beiden Frameworks stoßen wird.

Im schlimmsten Fall könnte dies bedeuten, dass nachdem man eine Netzwerkengine auf Basis einer der Netzwerkframeworks umgesetzt hat, neue Anforderungen auftauchen, die mit der Engine nur schwer umzusetzen sind.

Aus diesem Grund wurde festgelegt, dass eine unabhängige Netzwerkengine zu entwickeln ist, welche ein beliebiges Netzwerkframework einsetzen kann.

Für die erste Implementierung entschied man sich ZoidCom einzusetzen, da erwartet wurde, dass die Implementierung mit diesem Framework schneller durchzuführen ist.

6 Netzwerkintegrationstest

Um Annahmen der Evaluierung zu verifizieren, zu untersuchen, wie das Design einer Netzwerkengine am sinnvollsten aussehen sollte und herauszufinden, ob bei der Verwendung von ZoidCom relevante Performanceprobleme zu erwarten sind, wird ein erster Integrationstest durchgeführt.

Dieser Integrationstest besteht aus einem Verbindungsaufbau zwischen einem Server und einem Client, dem Versenden und Empfangen einfacher Nachrichten sowie Instanzen einer Testklasse unter Verwendung der Object Replication und Object Synchronization Systeme.

6.1 Testlauf

Der durchgeführte Testlauf ist in Anlage 5 beschrieben.

6.2 Ergebnisse des Integrationstests

Der durchgeführte Test hat gezeigt, dass die Integration der Netzwerkengine unter Verwendung von ZoidCom durchgeführt werden kann. Performance-Probleme oder sonstige unvorhergesehene Schwierigkeiten traten während des Testlaufs nicht auf.

Die während des Testlaufs gewonnenen Erkenntnisse (ins Besondere die Verwendung von ZoidCom) fließen in den Entwurf der Netzwerkengine ein.

7 Entwurf und Realisierung

Im Folgenden wird eine Übersicht über die Integration der Netzwerkengine in die Spieleengine, in 3 Kapiteln unterteilt, gegeben. Das erste Unterkapitel beschreibt den Entwurf und die Implementierung notwendiger Basissysteme, welche unabhängig von der Netzwerkengine sind, aber dennoch von dieser benötigt werden.

Das zweite Unterkapitel zeigt die wichtigsten Klassen, auf welche die Netzwerkengine aufbaut.

Abschließend wird im dritten Kapitel die Umsetzung des Dead-Reckoning/Clientprediction-Codes anhand der Anpassungen des PlayerControllers gezeigt.

7.1 Anpassungen der Basissysteme

Bereits während der frühen Integrationstests des Netzwerkframeworks (siehe Kapitel 6) wurden einige Schwachstellen der aktuellen X4-Engine aufgedeckt, die einer erfolgreichen Integration der Netzwerkengine im Wege standen. Diese Probleme lagen an Einschränkungen im ID- sowie Singleton-System.

Beide Systeme wurden komplett neu entworfen und in den X4-Code integriert.

7.1.1 Singleton-System

Unter dem Singleton-System verbirgt sich die Umsetzung des Singleton-Patterns, welches eine Möglichkeit beschreibt, sicherzustellen, dass nur eine bestimmte Anzahl Instanzen einer Klasse erzeugt werden kann. Im Fall der X4-Engine ist die Anzahl auf eine einzige Instanz festgelegt.

7.1.1.1 Bestehendes System

Das existierende Verfahren, um sicherzustellen, dass ein Objekt nur einmal vorhanden ist besteht darin, statische Instanzen von Klassen im globalen Namensraum anzulegen:

```
U::GameClass GameBase;
```

Anderen CPP-Modulen wird diese globale Instanz dann mit Hilfe von

```
extern U::GameClass GameBase;
```

bekannt gegeben, wodurch sie auf das Singleton zugreifen können.

Dieses Verfahren hat den Vorteil das es einfach zu verwenden und implementieren ist und keinen Performance-Overhead produziert.

Allerdings bringt dieses System auch den Nachteil einher, dass die Reihenfolge, in der die Klasseninstanzen erzeugt werden weder gesteuert, noch vorherbestimmt werden kann. Außerdem

werden auch alle Singletons instantiiert, selbst wenn sie nicht verwendet werden. Damit dies zu keinem relevanten Speicher- und Performanceoverhead führt, muss beim Erstellen des Konstruktors eines Singletons darauf geachtet werden, dass keine zeit- und ressourcenaufwändigen Operationen durchgeführt werden.

Darüberhinaus stellt das angewandte Verfahren nicht sicher, dass mehrere Instanzen eines Singletons erzeugt werden könnten, was u.U. zu unvorhersagbarem Verhalten der Engine führen kann.

Bereits ohne ein fertiges Design der Netzwerkengine ist abzusehen, dass auch der Netzwerkcode das Singleton-Pattern einsetzen wird. Nun ist es allerdings nicht gewollt, dass, falls der Spieler nur ein Singleplayerspiel startet, Teile der Netzwerkengine initialisiert werden. Dies ist mit der aktuellen Umsetzung des Singleton-Patterns nicht realisierbar. Deshalb muss das System angepasst werden.

7.1.1.2 Anforderungen

Die Anforderungen entsprechen denen des ursprünglichen System mit drei Ergänzungen (blau markiert):

- kein relevanter Performanceverlust
- Möglichkeit das Singleton-Pattern auf jede Klasse anwenden zu können
- **sicherstellen das von jedem Singleton auch nur eine Instanz erzeugt werden kann**
- **kontrollierbare Konstruktionsreihenfolge**
- **Singletons müssen nur dann konstruiert werden, wenn sie auch verwendet werden**

7.1.1.3 Implementierung

Für die Implementierung wurde ein auf Templates basierter Ansatz gewählt.

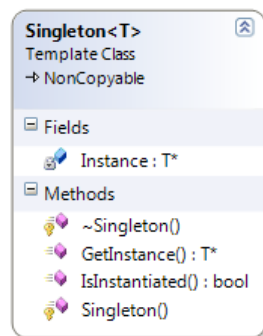


Abbildung 13: Klassendiagramm:
Singleton

Der Clue hinter der Implementierung ist die verwenden von Klassenvariablen und Klassenmethoden (welche in CPP mit Hilfe des Schlüsselwortes static umgesetzt werden).

Die Klassenvariable "Instance" ist initial auf NULL gesetzt und zeigt dadurch an, dass noch keine Instanz des Singletons erzeugt wurde. Wird nun GetInstance() aufgerufen, erzeugt diese Klassenmethode eine neue Instanz des Objektes, speichert dessen Adresse in der Klassenvariablen und liefert den Pointer zurück. Jeder nun folgende Aufruf von GetInstance() gibt lediglich den gespeicherten Pointer zurück.

Mit Hilfe von IsInstantiated() lässt sich überprüfen, ob bereits eine Instanz des Singletons erzeugt wurde.

7.1.1.4 Einschränkungen

Einige Grenzen hat diese Implementierung jedoch:

Um zu verhindern, dass über keinen anderen Weg als über die `GetInstance()`-Methode ein Singleton erzeugt werden kann, muss die ableitende Klasse den Zugriff auf ihren eigenen Konstruktor auf `private` setzen. Nur so kann einigermaßen sicher verhindert werden, dass eine weitere Instanz des Singletons durch einen Aufruf von

```
new MySingletonClass();
```

erzeugt wird.

Durch das setzen des Konstruktors auf `private` ist es jedoch auch nicht mehr für das Singletontemplate möglich in `GetInstance()` auf diesen Konstruktor zuzugreifen. Um dies wieder zu ermöglichen, muss die Singleton-Klasse als `friend` von der ableitenden Klasse mit Hilfe von

```
friend Xlib::Singleton<MySingletonClass>;
```

deklariert werden.

Eine weitere Einschränkung ist, dass die `GetInstance()`-Methode ausschliesslich den Standardkonstruktor der ableitenden Klasse aufruft. Somit ist es ohne weiteres nicht möglich Parameter an den Konstruktoraufruf zu übergeben. Ein verwendeter Ansatz um diese Einschränkung zu umgehen ist es in ableitenden Klassen eigene `GetInstance()`-Methoden zu implementieren, denen Parameter übergeben werden, welche an den Konstruktor der eigenen Klasse weitergereicht werden.

Eine gravierendere Änderung gegenüber der alten Umsetzung ist, dass der Destruktor einer Klasse nicht mehr aufgerufen wird. Dies liegt daran, dass das Singleton-System nicht vorsieht, dass eine einmal angelegte Instanz auch wieder freigegeben wird.

Somit muss entsprechender Code angepasst werden; beispielsweise indem Code aus dem Destruktor in eine Membermethode verschoben wird, welche explizit beim Beenden des Spiels aufzurufen ist.

7.1.2 ID-System

Das ID-System beschreibt ist die Implementierung eines Verfahrens, um Objekte in der X4-Engine eindeutig identifizieren zu können.

7.1.2.1 Bestehendes System

Für eine eindeutige Identifizierung von Objekten im Spiel wird die Adresse des Objektes verwendet. Dieses im Einzelspielermodus durchaus ausreichende Verfahren ist nicht nur extrem performant, sondern erzeugt auch keinen zusätzlichen Speicheroverhead.

Umständlich ist hingegen das Speichern und Laden von Spielständen, wenn mit diesem System Referenzen gespeichert werden müssen. Ein Beispiel eines solchen Eintrags im Savegame sieht folgendermaßen aus:

```
[...]
<listener listener="[0x03DFF040]" event=""/>
[...]
<component connection="galaxy" id="[0x03DFF040]" class="cluster"
macro="testcluster">
[...]
```

Dieses Savegame referenziert im Listener-Element die Testcluster-Komponente. Die ID `0x03DFF040` entspricht der Speicheradresse des Testcluster-Komponenten-Objektes zum Zeitpunkt,

zu dem das Spiel gespeichert wurde.

Wenn das Savegame nun wieder geladen wird, wird zunächst das Testcluster-Komponenten-Objekt erzeugt. Das Problem ist nun, dass das Objekt (mit sehr großer Wahrscheinlichkeit) an einer anderen Stelle im Speicher erzeugt wurde, als an der, an der es sich bei der Erzeugung des Savegames befunden hatte. Damit unterscheiden sich nun auch die IDs von denen, die im Savegame verwendet wurden.

Um dies zu kompensieren wird eine temporäre Tabelle erzeugt, die die IDs im Savegame auf die neu zugewiesenen IDs, d.h. Speicheradressen, abbildet.

Dieses Verfahren ist durchaus ausreichend für das Laden von Savegames, da es keinen Einfluss auf das laufende Spiel hat.

Auch die Netzwerkengine benötigt die Möglichkeit Objekte über ihre IDs zu referenzieren. So muss es z.B. möglich sein die Positionsänderungen eines Schiffes auf dem Server an alle Clients weiterzureichen. Dies erfordert, dass das Schiff auf dem Server eindeutig mit seinem Pendant auf den Clients in Verbindung gebracht werden kann.

Würde man nun das gleiche Verfahren wie beim Speichern von Savegames anwenden, gäbe es diverse Probleme zu lösen. Zum einen stellt sich die Frage, welche Speicheradresse verwendet werden soll: Die des Objektes auf dem Server oder die des Objektes auf dem Host, welcher das Objekt erzeugt hat? Zum anderen muss sichergestellt werden, dass eine ID nicht schon von einem anderen Objekt auf einem der verbundenen Rechner verwendet wird?

Dies führt zu veränderten Anforderungen, die an das ID-System gestellt werden.

7.1.2.2 Anforderungen

Die neuen Anforderungen die die Verwendung des ID-Systems für den Multiplayermodus mit sich bringt, sind lediglich Erweiterungen der Punkte, die bereits durch die Verwendung des Systems für das Speichern/Laden von Savegames gegeben wurden. Somit ist es nicht notwendig zwei unabhängige ID-Systeme (eines für das Laden/Speichern und eins für die Verwendung im Multiplayermodus) zu designen. Vielmehr kann das bestehende System durch das neue ersetzt werden.

Die Anforderungen im Einzelnen sind (neu hinzugekommene sind blau markiert):

- eindeutige Identifizierung von Objekten (auch während des Spiels)
- minimale Auswirkungen auf die Spielperformance und den Speicherbedarf
- das ID-System soll keinen wesentlichen Mehraufwand für Entwickler verursachen

7.1.2.3 Implementierung

Um die oben aufgezeigten Probleme zu lösen werden einige über die Anforderungen hinausreichende Eckpunkte für das neue ID-System festgelegt.

Jedes Objekt bekommt eine eindeutige laufende Nummer zugewiesen, welche als ID verwendet wird.

Die Verwendung von Zahlen (d.h. int bzw. int64) als ID stellt sicher, dass ein effizientes Arbeiten mit IDs möglich ist. Die Verwendung von Zeichenketten als IDs würde sich zu stark auf die Performance auswirken, auch wenn Strings den Vorteil bieten, eine für den Entwickler verständlichere ID verwenden zu können.

IDs werden nicht wiederverwendet.

Dadurch werden diverse Probleme umgangen die bei einer Wiederverwendung alter IDs auftreten. So wäre zum einen die Frage zu klären, wie man IDs, welche noch gar nicht verwendet wurden von solchen differenziert, die zwar einem Objekt zugewiesen wurden, das aber bereits zerstört ist.

Einem Objekt wird bei der Erzeugung automatisch eine ID zugewiesen.

Hiermit wird sichergestellt, dass kein Mehraufwand bei der Objekterzeugung für Entwickler gegenüber dem alten System verursacht wird. Müsse man erst eine ID für ein neu erzeugtes Objekt anfordern, würde dies auch eine Fehlerquelle für im Nachhinein nur schwer auffindbare Bugs darstellen.

Das nachträgliche Ändern der ID ist nicht möglich.

Der obige Zusatz stellt sicher, dass Objekte nicht über mehrere IDs referenziert werden können, was die Performance des ID-Systems unnötig belasten würde.

Anhand der Anforderungen und den zusätzlichen Punkten wurde ein aus zwei Templateklassen bestehendes ID-System entwickelt.

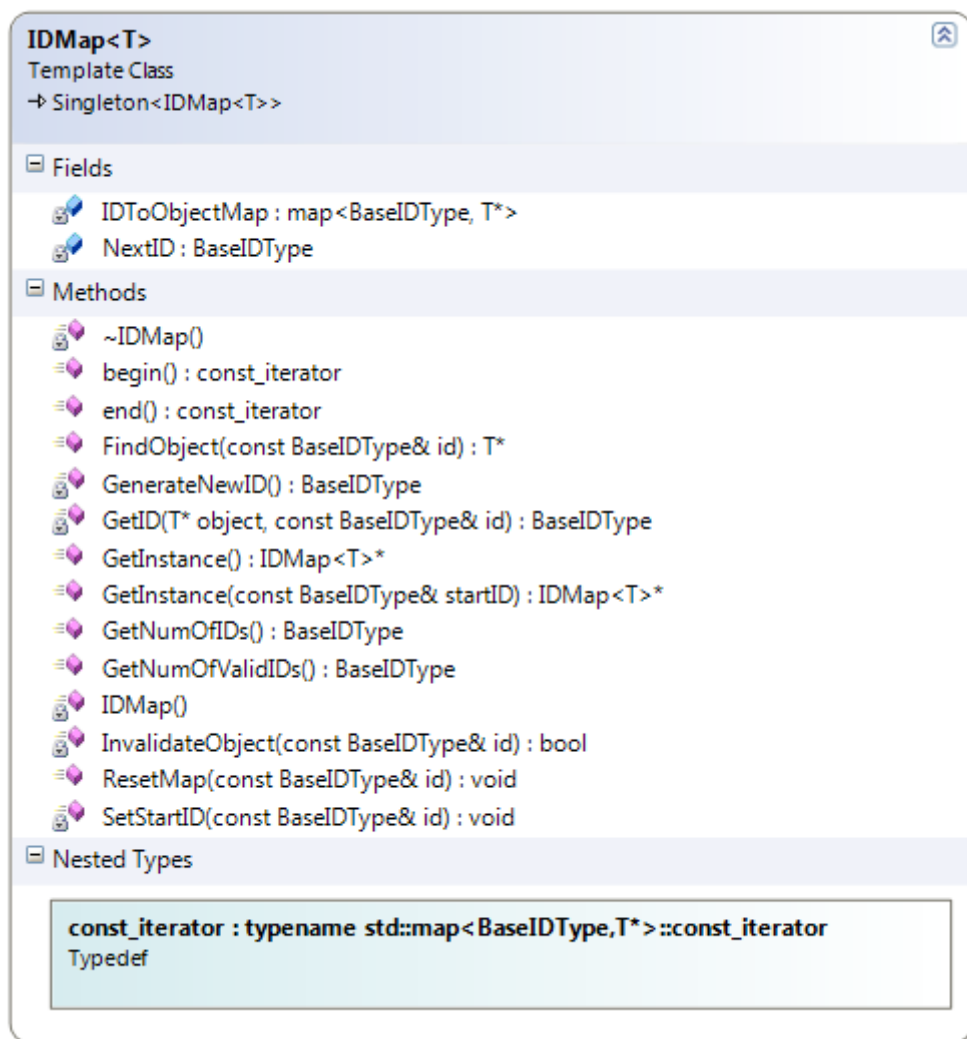


Abbildung 14: Klassendiagramm: IDMap

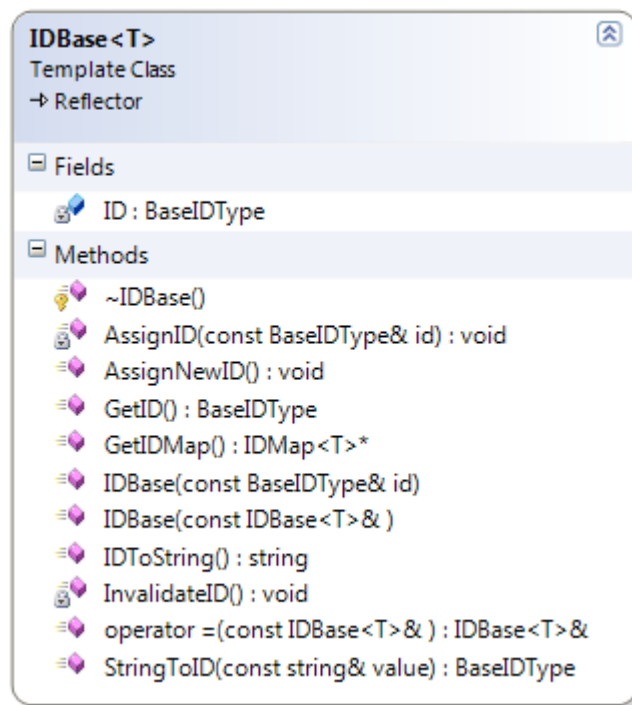


Abbildung 15: Klassendiagramm: IDBase

Dieses System funktioniert wie folgt:

Zunächst wird eine Basisklasse gewählt bzw. erstellt und von IDBase abgeleitet. Als Template-Parameter wird die eigene Basisklasse übergeben wie z.B.:

```
class UniverseID : public XLib::IDBase<UniverseID> { };
```

Damit wird ein neuer ID-Pool mit dem Namen “UniverseID” erstellt. Objekte aller Klassen, die von UniverseID abgeleitet werden, bekommen eine ID aus diesem Pool zugewiesen.

Dies geschieht automatisch während der Konstruktion des Objektes:

```
IDBase<T>::IDBase(const BaseIDType& id = InvalidID)
{
    AssignID(id);
}
```

AssignID() greift auf die zur IDBase zugehörigen IDMap zu und assoziiert das eigene Objekt mit der neuen ID. Wurde keine ID angegeben, generiert GetID() eine neue, noch nicht benutzte ID.

```
void IDBase<T>::AssignID(const BaseIDType &id = InvalidID)
{
    ID = IDMap<T>::GetInstance()->GetID(static_cast<T*>(this), id);
}
```

Wird das Objekt später zerstört, so wird auch die assoziierte ID wieder als ungültig markiert:

```
virtual IDBase<T>::~~IDBase()
{
    InvalidateID();
}
```

InvalidateID() ruft dazu die entsprechende Methode der zugehörigen IDMap auf, welche die ID wieder aus der Liste der assoziierten IDs entfernt.

```
void IDBase<T>::InvalidateID()
{
    IDMap<T>::GetInstance()->InvalidateObject(ID);
    ID = InvalidID;
}
```

Diese Automatisierung ist u.a. deshalb möglich, da IDMap ein Singleton ist, und somit die Methoden in IDBase direkt auf die IDMap zugreifen können, ohne dass diese explizit vom Aufrufer angegeben werden muss.

Beim ersten Zugriff auf die IDMap über IDMap<T>::GetInstance() wird die Instanz der IDMap erstellt. Jeder folgende Aufruf liefert lediglich einen Pointer auf die bereits erstellte Instanz zurück.

IDMap verwendet intern einen ID-Counter, der einer laufenden Nummer entspricht. Dieser Counter kann nur bei der Erstellung der IDMap gesetzt werden und legt die StartID fest, die dem ersten Objekt, welches eine ID anfordert, zugewiesen wird.

IDMaps GetID()-Methode generiert eine neue ID und fügt einen neuen Eintrag in der Assoziierungsliste hinzu:

```
BaseIDType IDMap<T>::GetID(T* object, const BaseIDType& id)
{
    ++NextID;
    BaseIDType newid = NextID - 1;
    IDToObjectMap.insert(pair<BaseIDType, T*>(newid, object));
    return newid;
}
```

Etwas komplexer wird die Methode durch die Möglichkeit, über den zweiten Parameter eine ID anzugeben, welche dem Objekt zugewiesen werden soll. Hier muss nun der Fehler abgefangen werden, dass die ID eventuell schon vergeben wurde. Auch ist darauf zu achten, dass der NextID-Counter angepasst werden muss.

InvalidateObject() hat es dagegen etwas leichter:

```
bool IDMap<T>::InvalidateObject(const BaseIDType &id)
{
    return !!IDToObjectMap.erase(id);
}
```

Soviel zur Zuweisung von IDs an Objekten.

Um ein zu einer ID gehöriges Objekt zu finden, wird die FindObject()-Methode verwendet, die in der Assoziierungsliste nach der ID sucht und den Pointer des zugehörigen Objektes zurück gibt.

```
T* IDMap<T>::FindObject(const BaseIDType &id) const
{
    std::map<BaseIDType, T*>::const_iterator position =
        IDToObjectMap.find(id);
    return position == IDToObjectMap.end() ? NULL : position->second;
}
```

Die weiteren Methoden der IDBase- und IDMap-Klasse dienen dazu, das Arbeiten mit dem ID-System zu erleichtern und sind im Quellcode genauer dokumentiert.

7.1.2.4 Einschränkungen

Beim Design des ID-Systems mussten mehrere Kompromisse eingegangen werden. Häufig stand eine performante Lösung einer fehlerunanfälligeren und sichereren Variante entgegen.

Eine dieser Entscheidungen betrifft die Auswahl des Datentyps der IDs. Die Verwendung eines Standarddatentyps wie z.B. den Integer-Datentyp hat zur Folge, dass die Anzahl der IDs, die innerhalb eines ID-Pools vergeben können, von der maximalen Grösse, die ein Wert dieses Datentyps (im Fall eines 32-bit Integers ohne Vorzeichen etwa 4,3 Milliarden) annehmen kann, begrenzt wird.

4,3 Milliarden hört sich zunächst durchaus ausreichend an. Allerdings wurde für das ID-System entschieden IDs nur einmal zu vergeben, um ein möglichst performantes System designen zu können. Dies hat zur Folge, dass bei jeder Erstellung/Zerstörung eines Objektes eine neue ID angefordert wird. Dies schliesst auch temporäre Objekte mit ein.

Das führt dazu, dass selbst wenn zu keinem Zeitpunkt im System gleichzeitig 4,3 Milliarden Objekte referenziert werden, dennoch ein Überlauf des ID-Counters denkbar ist.

Hierzu gesellt sich die Tatsache, dass beim Multiplayer-Modus Objekte auf allen verbundenen Rechnern untereinander referenziert werden müssen und falls die Entscheidung zu Gunsten eines Multiplayer-Spielmodus gefällt wird, der zur Folge haben würde, dass von Spielern zur Verfügung gestellte Server ununterbrochen laufen, ist abzusehen, dass der ID-Pool schon nach wenigen Tagen (bzw. Stunden) Spielzeit erschöpft ist.

Aus diesem Grund wurde das System so aufgebaut, dass es relativ einfach ist, den Datentyp der IDs zu ändern, um ihn, falls es notwendig werden sollte, von einem 32-bit Integer auf einen 64-bit Integer zu ändern. Theoretisch wäre es sogar möglich ihn durch einen eigenen Datentyp zu ersetzen, welcher dann nur noch durch den vorhandenen Speicher des PCs eingeschränkt ist.

Ein weiterer Kompromiss, der getroffen werden musste, betrifft die Zustände, die eine Assoziation zwischen ID und Objekt annehmen kann. Generell sind drei Zustände möglich:

- ID ist mit einem Objekt assoziiert
- ID ist mit keinem Objekt assoziiert
- ID war mit einem Objekt assoziiert, ist es nun aber nicht mehr

Zugunsten der Gesamtperformance unterscheidet das ID-System nicht zwischen dem zweiten und dritten Zustand. Dies hat weitreichende Auswirkungen auf die Fehleranfälligkeit des gesamten Designs, da man Funktionen die mit dem Arbeiten des Systems notwendig sind, verbieten müsste. Ein Beispiel ist die Möglichkeit Objekten bei ihrer Konstruktion, IDs zuzuweisen.

Dies ist zum Beispiel nötig, damit ein Schiff auf allen verbundenen Hosts die gleiche ID annehmen kann.

Fehleranfällig ist dies zumindest wie das folgende Szenario zeigt:

Ein Raumschiff speichert sich die ID einer Station ab, an die es andocken möchte. Nun wird die Station zerstört und eine neue mit der nun freien ID erstellt. Folglich versucht das Raumschiff jetzt an der anderen, d.h. "falschen" Station anzudocken.

Um diese Art von Fehlern zu vermeiden sind bestimmte Funktionen des ID-Systems mit Vorsicht zu geniessen.

7.2 Netzwerkeingine

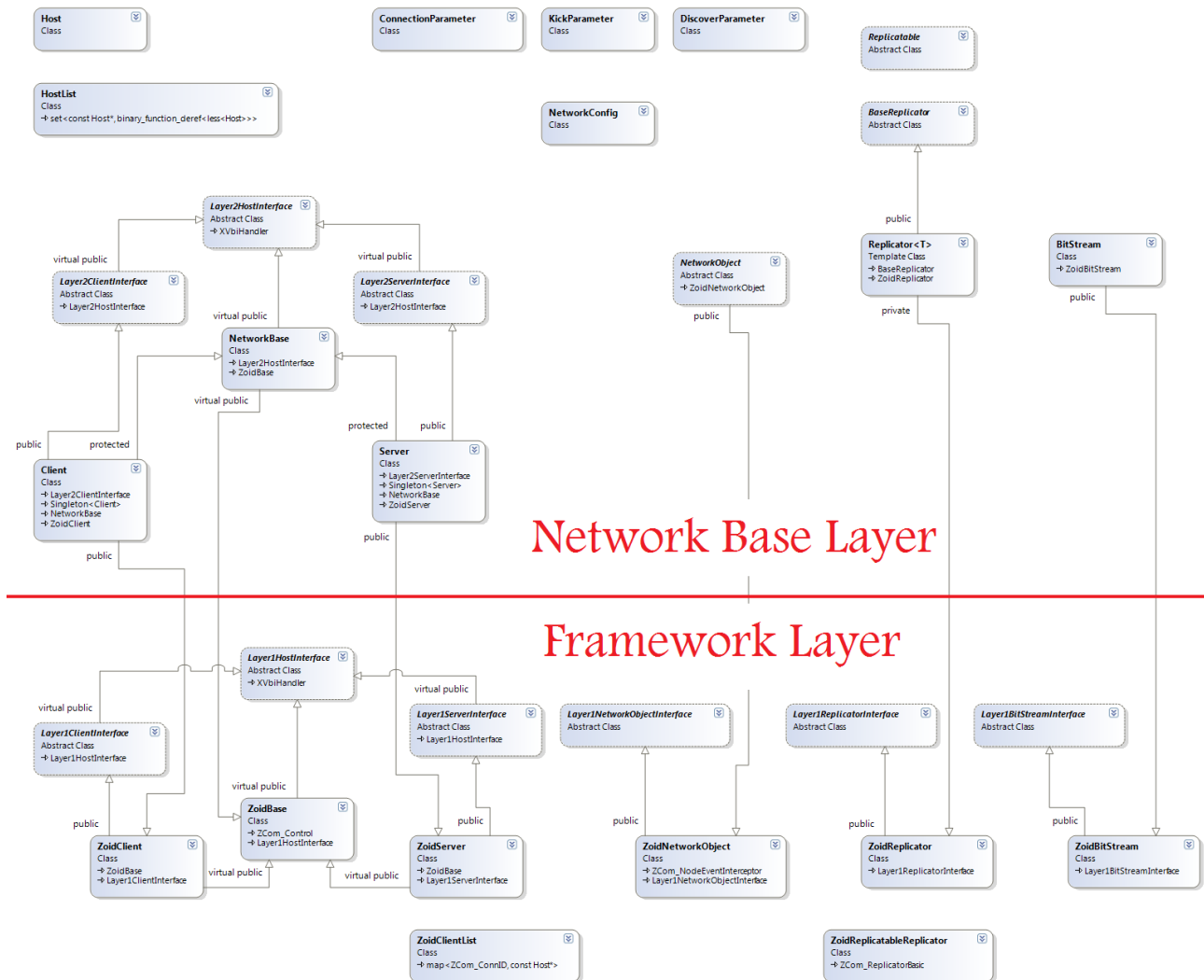


Abbildung 16: Netzwerkeingine Übersicht

Basierend auf dem Integration Plan (siehe Anlage 4) wurde die hier dargestellte Netzwerkeingine entwickelt. Die gesamte Engine kann dabei nicht nur in ihre unterschiedlichen Layer aufgeteilt werden, sondern auch in ihre unterschiedlichen Teilbereiche, deren grundsätzliche Funktionalität im Folgenden erläutert wird.

7.2.1 Server/Client

Der Kern der Netzwerkengine findet sich in Form zweier Klassen wieder; der Server- und der Client-Klasse.

Die Hauptaufgaben beider ist die Initialisierung der Netzwerkengine, sowie den Verbindungsaufbau zwischen den Hosts zu ermöglichen.

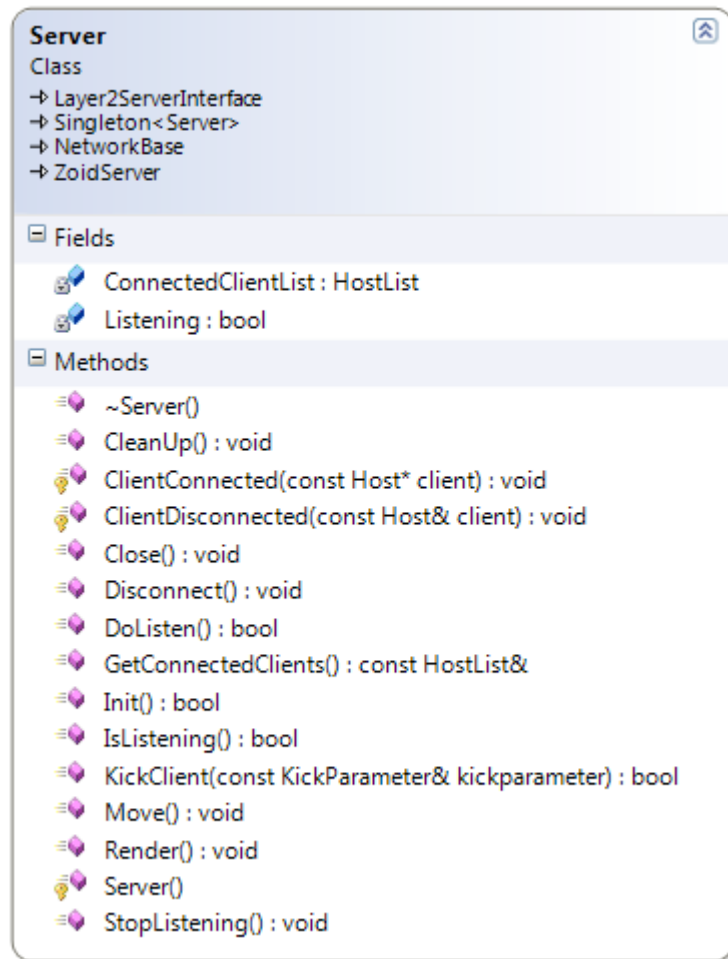


Abbildung 17: Klassendiagramm: Server

Hierzu dienen auf Seiten der Serverklasse zwei Methoden:

`DoListen()` versetzt den Server in einen „Lauschzustand“. In diesem wartet er auf eingehende Verbindungsanfragen von Clients und entscheidet, ob er diese annimmt. Wird eine Verbindung zu einem Client etabliert, fügt der Server diesen der Liste der verbundenen Clients hinzu.

Mit Hilfe der `StopListening()`-Methode kann der „Lauschzustand“ wieder verlassen werden.

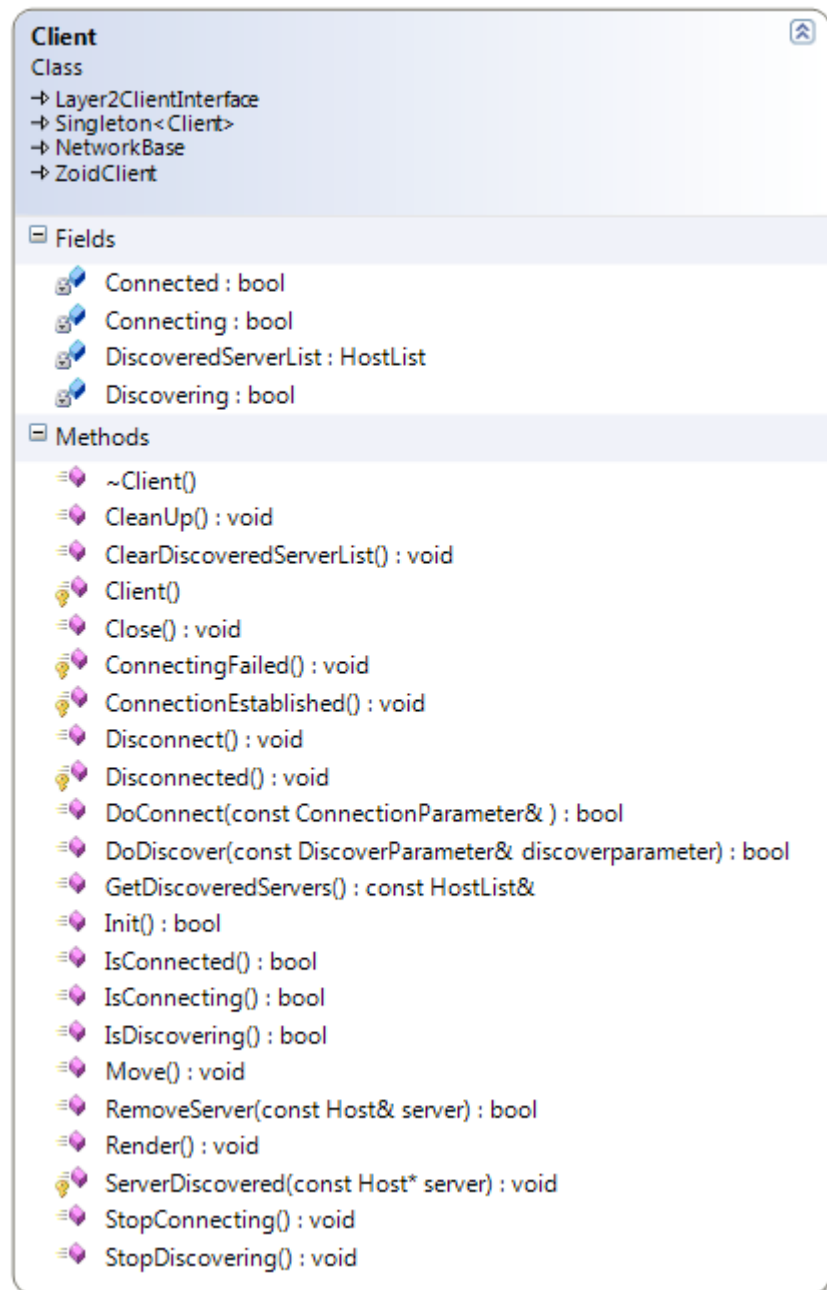


Abbildung 18: Klassendiagramm: Client

Die Clientklasse verfügt über insgesamt 4 Methoden die beim Verbindungsaufbau eingesetzt werden können.

Mit Hilfe von DoDiscover() und StopDiscover() kann ein Client nach Servern im Netzwerk suchen, ohne dass er die IP des Servers kennen muss. Die DoConnect()-Methode erlaubt eine Verbindung mit einem bekannten Server zu etablieren. Falls der Verbindungsaufbau abgebrochen werden soll, kann dies über den Aufruf der StopConnecting()-Methode geschehen.

Ähnlich wie die ConnectedClient-Liste des Servers speichert die Client-Klasse eine Liste der Server, die während des Discover-Vorgangs gefunden wurden.

Zur Initialisierung des Netzwerksystems besitzen beide Klassen eine Init()-Methode, welche dafür sorgt, dass die verschiedenen Subsysteme initialisiert werden.

7.2.2 BitStream

Für die Übertragung von Daten kommt die BitStream-Klasse zum Einsatz. Dank ihrer überladenen Read()- und Write()-Methoden kann aus nahezu jedem beliebigen Datum ein BitStream erzeugt bzw. das Datum wieder aus einem BitStream gelesen werden. Statt die Daten direkt zu übertragen wird lediglich der BitStream gesendet/empfangen, was die notwendige Bandbreite für die Übertragung verringert.

7.2.3 Replicator

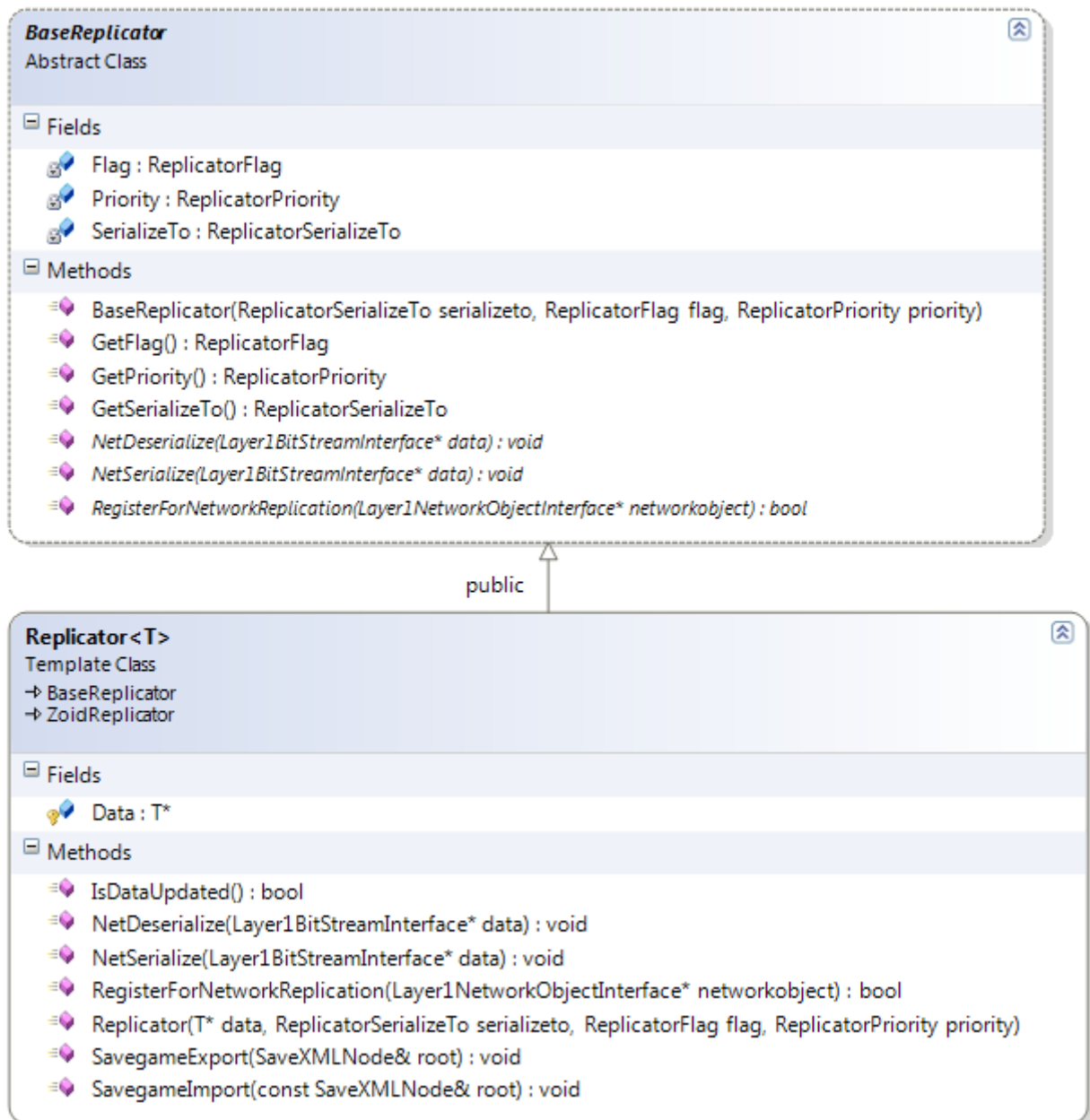


Abbildung 19: Klassendiagramm: Replicator

Die Umsetzung des Replikationssystems wurde in 2 Klassen unterteilt. Die BaseReplicator-Klasse enthält die grundlegenden Funktionen die zur Objektsynchronisierung bzw. -replikation nötig sind, während die Templateklasse, deren Templateparameter eine replizierbare (d.h. eine von Replicable abgeleitete) Klasse erwartet, die eigentliche Arbeit mit dem zu replizierenden Objekt tätigt.

7.2.4 NetworkObject

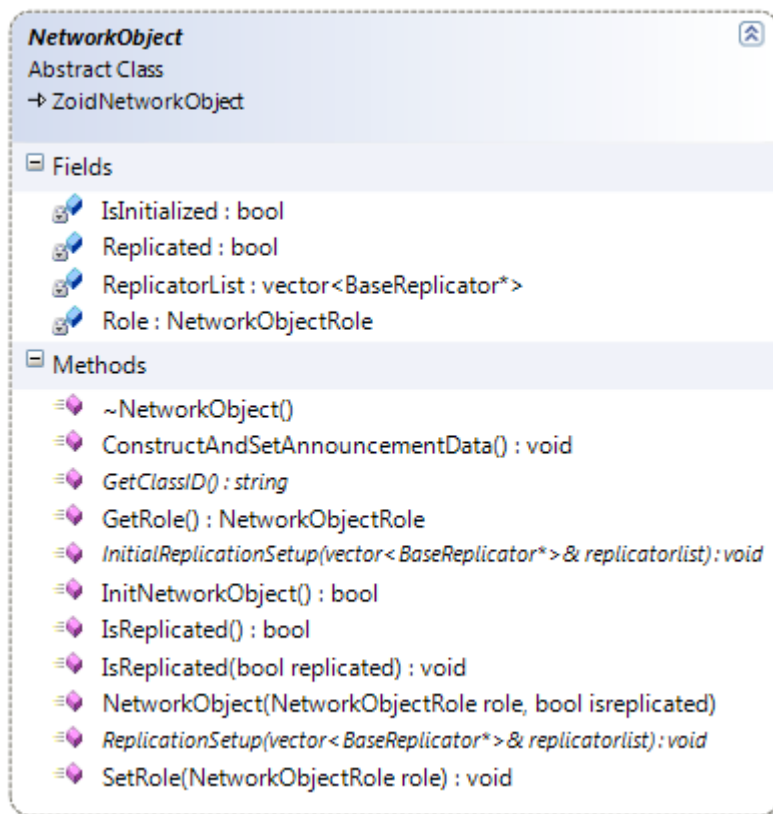


Abbildung 20: Klassendiagramm: Network Object

Die NetworkObject-Klasse ist die Schnittstelle zwischen Objekten der Spiele- und der Netzwerkengine. Jede Klasse, die über das Netzwerk synchronisiert werden kann, wird hiervon abgeleitet. Dadurch stellt diese für die Objektsynchronisierung notwendige Funktionen zur Verfügung und bekommt eine Netzwerkrolle zugewiesen.

7.2.5 ObjectReference

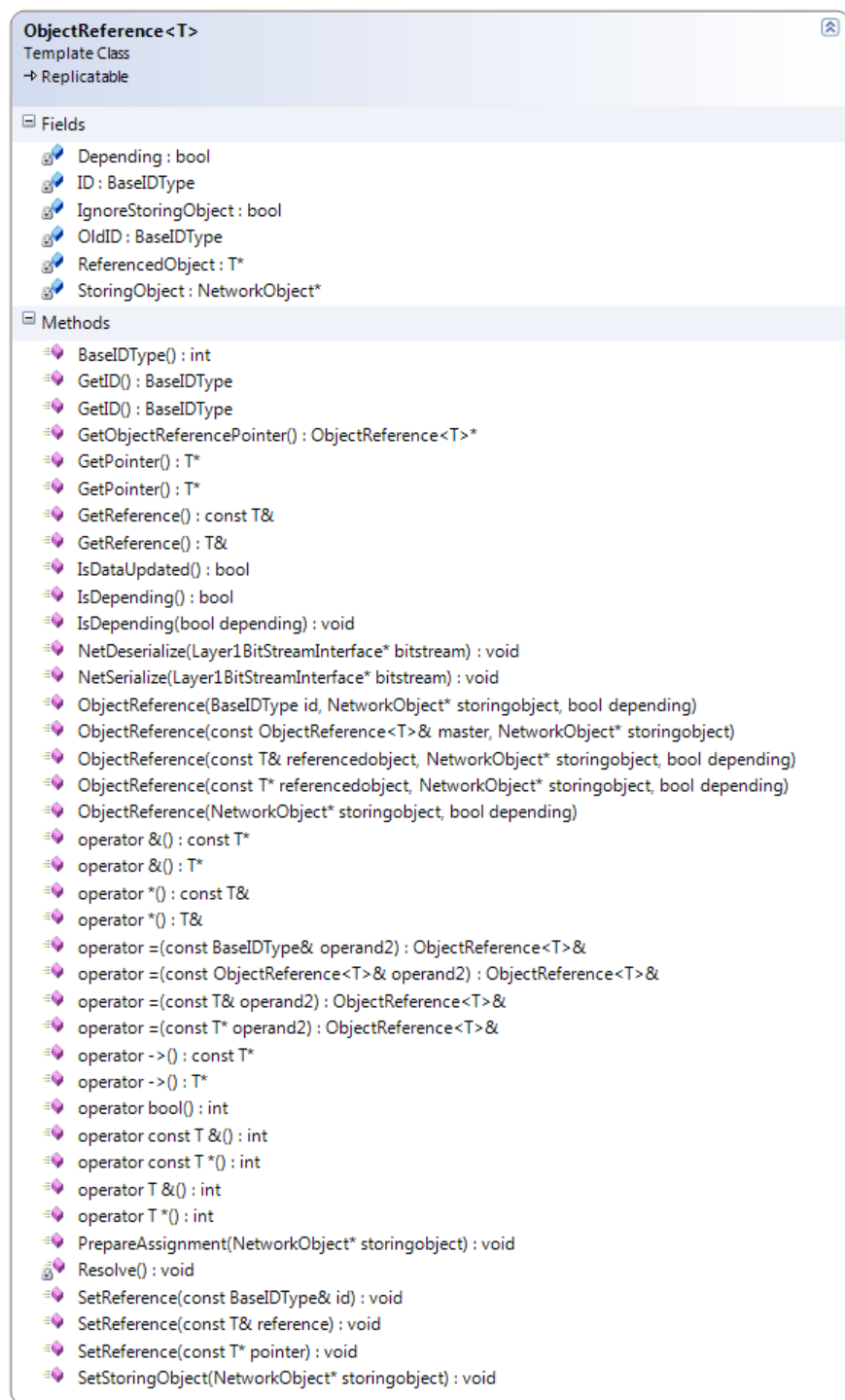


Abbildung 21: Klassendiagramm: *ObjectReference*

Die `ObjectReference`-Klasse entspricht der Umsetzung des in der Design Analyse geplanten Object Referenz Systems (Anlage 3 Kapitel 2.4).

7.3 Clientprediction für den PlayerController

Die Playercontroller-Klasse wird in der Spieleengine eingesetzt, um es dem Spieler zu ermöglichen die Kontrolle über ein Objekt (in der Regel sein eigenes Raumschiff) zu übernehmen. Dafür werden direkt die Tastatur- bzw. Joystick-Eingaben verarbeitet wodurch die Geschwindigkeit, Richtung und Beschleunigung des Raumschiffes geändert wird.

Damit diese Richtungsänderungen auch auf dem Client umgesetzt werden, wird der PlayerController angepasst.

Der Code, der die Spielereingaben verarbeitet wird auf Clientseite durch Code für die Clientprediction ersetzt. Beispielfhaft wird hier die lineare Interpolation gewählt.

Dazu wird die Move()-Methode, die jeden Frame ausgeführt wird, angepasst.

Zunächst überprüft der folgende Code, ob seit dem letzten Move()-Aufruf ein neues PDU angekommen ist. Ist dies der Fall werden die neuen Daten in LastRemoteVel, LastRemoteAcc und LastStartValue gespeichert und UpdateReceived auf den Zeitpunkt gesetzt, zu dem das Update angekommen ist.

```
if (RemoteVel != LastRemoteVel || RemoteAcc != LastRemoteAcc ||
    RemoteStartValue != LastStartValue) {
    UpdateReceived = CurTime;
    LastRemoteVel = RemoteVel;
    LastRemoteAcc = RemoteAcc;
    LastStartValue = RemoteStartValue;
}
```

Dabei entspricht LastRemoteVel der Geschwindigkeit, die das Raumschiff auf dem Server hatte, LastRemoteAcc dessen Beschleunigung und LastStartValue der Position und Rotation.

Der nächste Schritt ist die Berechnung des Lags, was durch

```
float nextupdate = Node->getEstimatedTimeUntilPossibleUpdate();
```

geschieht.

Wurde zuvor ein neues PDU empfangen, so wird dieses als nächstes verarbeitet.

Da das Position des Schiffes aus dem empfangenen Datenpaket bereits veraltet ist, gilt es zunächst die aktuelle Position des Schiffes auf dem Server zu interpolieren:

```
float curvalueonserver = LastStartValue.GetPos().X() +
    LastRemoteVel.GetLinear().X() * nextupdate;
```

Unser angestrebtes Ziel ist es, dass das Schiff in halber Lagzeit wieder wieder mit dem Server synchronisiert ist (vorausgesetzt die Geschwindigkeit des Schiffes hat sich nicht geändert).

```
float x1 = curvalueonserver + LastRemoteVel.GetLinear().X() * nextupdate/2;
```

Als letztes berechnen wir die Geschwindigkeit, die das Schiff auf dem Client haben muss, damit es sich nach halber Lagzeit auch an der angestrebten Position befindet.

```
float velx = (x1 - GetValueAtTime(CurTime).GetPos().X()) / (nextupdate/2);
```

Die y und z Werte sowie die Rotationsmatrix wird entsprechend berechnet und die Geschwindigkeit gesetzt.

Nachdem das Schiff synchronisiert wurde, (d.h. in halber Lagzeit), soll das Schiff weiter interpoliert werden. Dazu werden die Variablen des DeadReckoning-Systems gesetzt.

8 Network Simulator

8.1 Aufbau des Testnetzwerkes

Das Testnetzwerk ist aus 3 PCs aufgebaut.

Auf PC 1 läuft unter Windows die Serveranwendung. Die IP des Rechners ist 192.168.0.29.

Auf PC 2 läuft ebenfalls unter Windows die Clientanwendung. Die IP dieses Rechners sei 192.168.0.28.

PC 3 fungiert als Router. Als Betriebssystem kommt hier Debian zum Einsatz. Der Netzwerkkarte dieses Rechners sind 2 IPs zugewiesen: 192.168.0.23 und 192.168.0.227.

Um dafür zu sorgen, dass die Daten von PC1 nicht direkt zu PC2 sondern über den Debian Rechner versandt werden, wird die Routingtabelle des Windows-PCs verändert. Dazu dient der Kommandozeilenbefehl:

```
route ADD 192.168.0.28 MASK 255.255.255.255 192.168.0.23 METRIC 1 IF
0x10003
```

Dies fügt der Routingtabelle des Server einen neuen Eintrag hinzu, der soviel bedeutet wie: „Sende alle Pakete mit der Empfänger-IP des Clients (192.168.0.28 MASK 255.255.255.255) über den Router (192.168.0.23) und verwende dafür die entsprechende Netzwerkkarte (IF 0x10003).“

Die Angabe „METRIC 1“ bewirkt, dass diese Route bevorzugt verwendet werden soll.

Entsprechend wird auch die Routingtabelle auf dem Client angepasst:

```
route ADD 192.168.0.29 MASK 255.255.255.255 192.168.0.227 METRIC 1 IF
0x10003
```

8.2 Network Simulator Skript

Um dafür zu sorgen, dass die beim Router ankommenden Server-/Client-Daten mit einem konfigurierbaren Lag versehen werden, wird auf dem Router folgendes TCL-Skript gestartet.

```
# this script requires root privileges since it needs to execute the sysctl
command

##### Network Simulator setup #####
set ns [new Simulator]
$ns use-scheduler RealTime
```

Der obige Code erstellt ein neues Network Simulator Objekt und sorgt dafür, dass dieses für den Emulatormodus vorbereitet wird.

```
##### ensure that IP-Forward is disabled #####
set ipforw [exec sysctl -n net.ipv4.ip_forward]
if $ipforw {
    puts "can not run with ip forwarding enabled"
    exit 1
}
```

Um zu verhindern, dass der Linuxkernel selbst das Routing der ankommenden Pakete übernimmt, wird mit dem hier gezeigten Code das IP-Forwarding-System des Kernels deaktiviert.

```
##### set used variables #####
set me [exec hostname]
```

```

set notme "(not ip host $me) "
set notbcast "(not ether broadcast) "
set ServerToClient "(ip src 192.168.0.29) "
set ClientToServer "(ip dst 192.168.0.29) "

```

Um die folgenden Codeteile übersichtlich zu halten, werden einige Variablen erzeugt, die für die Filterregeln (siehe unten) eingesetzt werden.

```

##### Network topology #####
set serverNode [$ns node]
set clientNode [$ns node]
set outgoingNode [$ns node]
$ns simplex-link $serverNode $outgoingNode 10Mb 500ms SFQ
$ns simplex-link $clientNode $outgoingNode 10Mb 500ms SFQ

```

Diese Befehle erstellen eine einfache Netzwerktopologie, die vom Network Simulator verwendet wird. Dabei werden 3 Knoten erzeugt; jeweils einer für eingehende Daten vom Server bzw. Client sowie einer, der dazu dient, die ausgehenden Daten an den Empfänger weiterzureichen.

Die Knoten werden untereinander mit einem einfachen Link verbunden. Über einen solchen Link können Daten nur in eine Richtung versandt werden. Da in unserem Fall der Datenfluss nur unidirektional abläuft (vom Server-/Clientknoten zum ausgehenden Knoten), ist dieser Linktyp für uns ausreichend.

Die Angabe 10 Mb gibt die Bandbreite des Links an (10 Mbit) während „500 ms“ dafür sorgt, dass die Daten 500 ms benötigen, bis sie beim ausgehenden Knoten ankommen. Dies ist somit die Stelle, an der der simulierte Lag eingestellt werden kann. Soll dieser z.B. 500 ms betragen, wird für beide Links jeweils 250 ms eingetragen ($250 \text{ ms} * 2 = 500 \text{ ms}$ Lag bzw. RTT).

Die Angabe SFQ steht für „Stochastic Fairness Queuing“. Es gibt das eingesetzte Queuingverfahren an, welches eingesetzt werden würde, sollte es zu einer „Überlastung“ des Links kommen, was in unserer Simulation jedoch nicht der Fall ist.

Die folgenden Zeilen erstellen drei Netzwerke, welche später über Agenten mit den oben angelegten Knoten verbunden werden. Dies sorgt dafür, dass die Pakete, die an der Netzwerkkarte des Routers ankommen in die Netzwerksimulation gespeist werden.

Weiterhin werden Filterregeln angelegt um zu verhindern, dass Datenpakete, die bereits durch das simulierte Netzwerk geschickt wurden, erneut verlangsamt werden.

```

##### Network setup #####
### server network ###
set serverNetwork [new Network/Pcap/Live]
$serverNetwork set promisc_ true
$serverNetwork open readonly eth0
set serverMAC [$serverNetwork linkaddr]
set notServerNIC "(not ether src $serverMAC) "
$serverNetwork filter "$ServerToClient and $notServerNIC"
### client network ###
set clientNetwork [new Network/Pcap/Live]
$clientNetwork set promisc_ true
$clientNetwork open readonly eth0:1
set clientMAC [$clientNetwork linkaddr]
set notClientNIC "not ether src $clientMAC"
$clientNetwork filter "$ClientToServer and $notClientNIC"
### outgoing network ###

```

```

set ipnet [new Network/IP]
$ipnet open writeonly

##### Agent setup #####
### server agent ###
set serverAgent [new Agent/Tap]
$serverAgent network $serverNetwork
$ns attach-agent $serverNode $serverAgent
### client agent ###
set clientAgent [new Agent/Tap]
$clientAgent network $clientNetwork
$ns attach-agent $clientNode $clientAgent
### outgoing agent ###
set outgoingAgent [new Agent/Tap]
$outgoingAgent network $ipnet
$ns attach-agent $outgoingNode $outgoingAgent
$ns connect $serverAgent $outgoingAgent
$ns connect $clientAgent $outgoingAgent

```

Der Befehl

```
$ns run
```

bewirkt schlussendlich, dass die Netzwerksimulation gestartet wird.

9 Abschließende Performancetests

9.1 Testmodul

Um die Netzwerkengine abschließend zu testen und die Verbesserungen, die Clientprediction sowie Dead Reckoning mit sich bringt zu demonstrieren, wurden zwei Testmodule erstellt.

Beide Testmodule aktivieren die Netzwerkengine. Im Fall des Servermoduls wird jedoch ein Server-, im Fall des Clientmoduls ein Clientobjekt erstellt.

Auf dem Servermodul wird beim Starten ein kleines Testuniversum, bestehend aus einem Spielerschiff sowie einer Raumstation generiert und das UI aktiviert, welches das Cockpit anzeigt.

Sobald der Server läuft, kann der Client gestartet werden. Nach erfolgreichem Verbindungsaufbau mit dem Server überträgt dieser das Universum über das Netzwerk, welches vom Client dargestellt wird.

Nach der initialen Übertragung kann auf dem Server die Steuerung des Raumschiffes übernommen werden. Die Änderungen der Schiffsposition werden dabei zum Client geschickt, der diese auf seiner Seite umsetzt.

9.2 Testauswertung

Insgesamt wurden 4 Testdurchläufe durchgeführt. Jeweils mit unterschiedlichen DeadReckoning-Algorithmen sowie mit und ohne simuliertem Lag. Zur Dokumentation wurde ein Videomitschnitt von zwei der durchgeführten Tests erstellt.

In beiden Videos wird im oberen linken Bereich die Sicht des Clients angezeigt, während der untere rechte Bereich die Bildschirmausgabe auf dem Server entspricht.

Wie in den Videos zu sehen, führt ein großer Lag nicht nur dazu, dass die Schiffsposition aus Clientperspektive zeitverzögert ist, sondern insbesondere sind die Positionsänderungen ohne aktivierter Interpolation sehr ruckartig. Würde sich die Kamera auf dem Client nicht im Cockpit sondern z.B. in einem anderen Raumschiff befinden, würde man ein sich ruckartig bewegendes Schiff sehen.

In einer Umgebung, die nahezu keinen Lag zu verkraften hat, ist die Darstellung auf dem Client selbst ohne Interpolation flüssig, wie das zweite Video zeigt.

Um die Verbesserungen, die die Clientprediction mit sich bringt genauer aufzuzeigen wurden die Schiffspositionen während der Tests geloggt und im Anschluss ausgewertet. Die folgenden Graphen zeigen das Ergebnis dieser Auswertung:

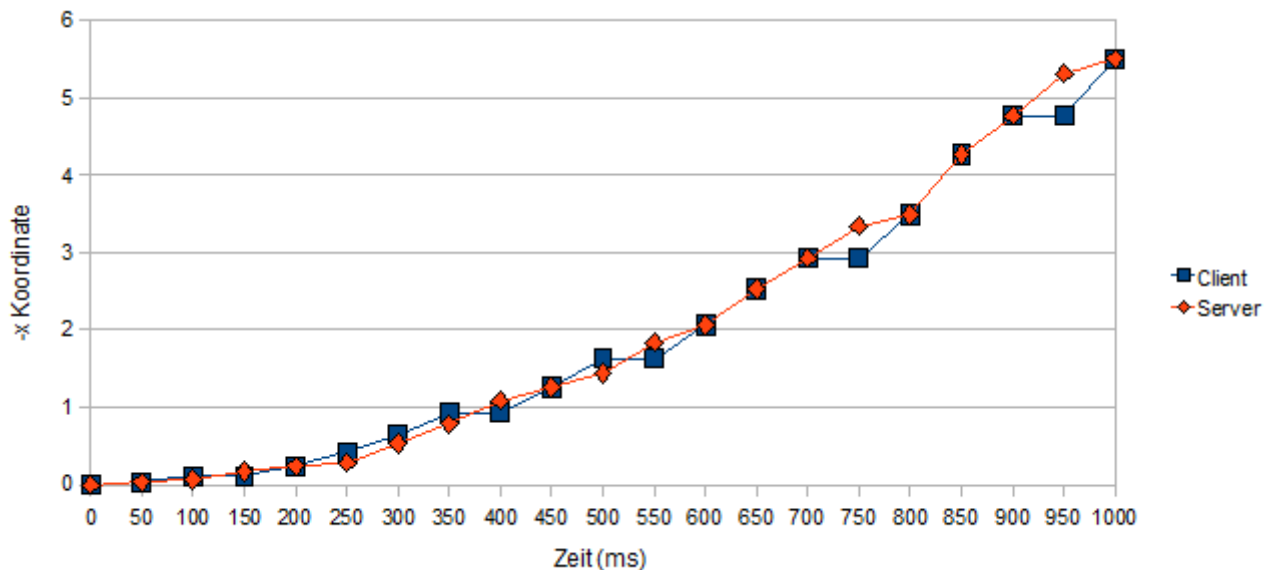


Abbildung 22: Schiffsposition Client/Server - P2P - kein Lag

Die Auswertung des Testlaufs ohne Lag und ohne Interpolation der Spielerposition bestätigt den aus dem Video gewonnenen Eindruck. Client- und Server-Position unterscheiden sich nur marginal voneinander.

Auffällig sind lediglich zwei Abweichungen bei 750 ms und 950 ms. Diese lassen sich dadurch erklären, dass dem Client noch kein Update der Schiffsposition geschickt wurde und er das Schiff auf der vorherigen Position belässt.

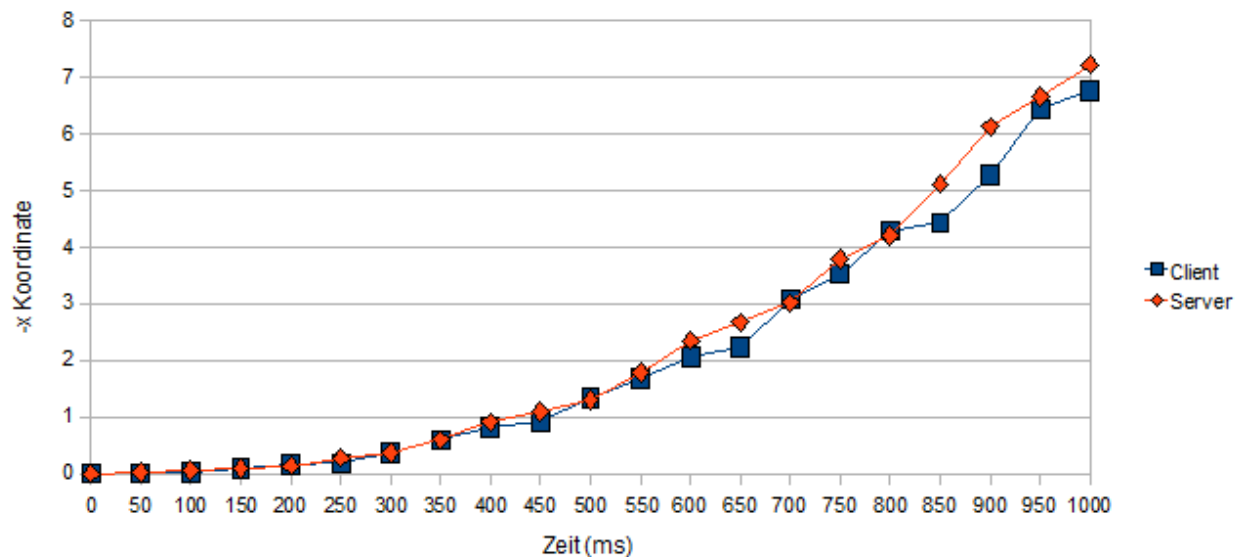


Abbildung 23: Schiffposition Client/Server - lineare Interpolation - kein Lag

Bei aktivierter linearer Interpolation sieht der resultierende Graph hingegen ein wenig anders aus. Am Anfang, wenn sich die X-Position des Schiffes nur sehr langsam ändert, ist die Abweichung zwischen Client und Server ebenso wie bei der Punkt-Zu-Punkt-Übertragung nicht erkennbar.

Je schneller sich der x-Wert des Schiffes jedoch ändert, desto größer ist auch die resultierende Abweichung auf dem Client. Dieser „hinkt“ dem Server durchgehend hinterher, wobei die Abweichungen jedoch so gering sind, dass sie keine Auswirkungen auf das Spielerlebnis haben.

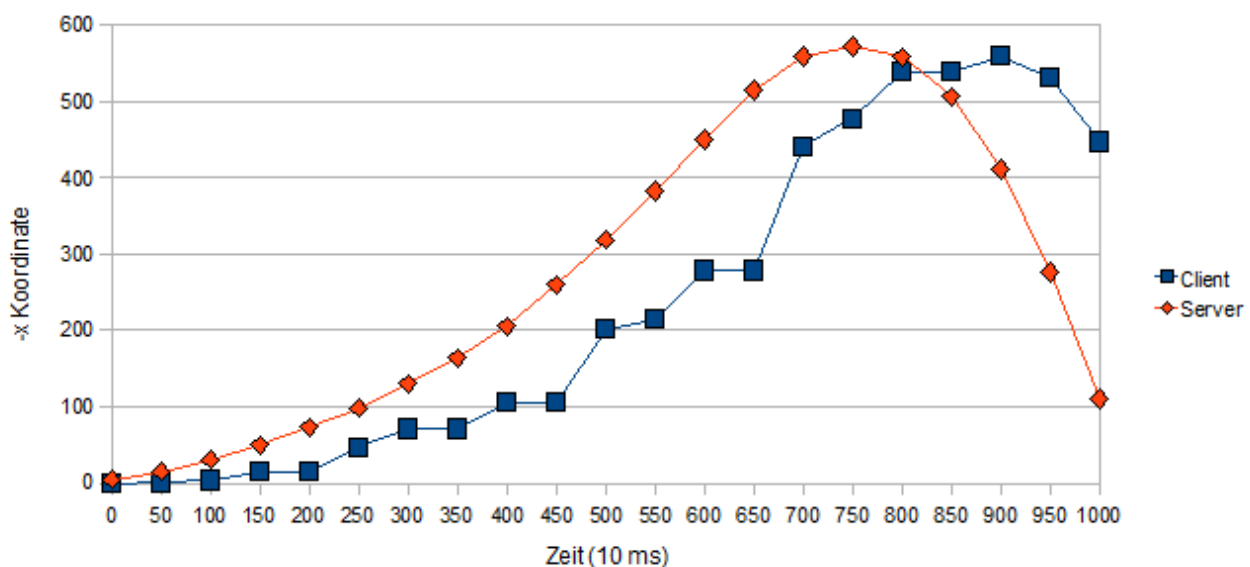


Abbildung 24: Schiffposition Client/Server - P2P - 1s Lag

Ganz anders verhält es sich hingegen sobald Lag ins Spiel kommt. Eine Punkt-Zu-Punkt-Übertragung ohne Interpolation hat zur Folge, dass der Client dem Server ununterbrochen hinterherhinkt. Die Verzögerung entspricht dabei der des Lags (im oberen Fall 1s).

Hinzu kommt, dass ZoidCom den hohen Lag erkennt und die Updatehäufigkeit mit der Positionsupdates an den Client geschickt werden herunter regelt. Dies führt im Test dazu, dass die Darstellung auf dem Client extrem ruckartig erscheint.

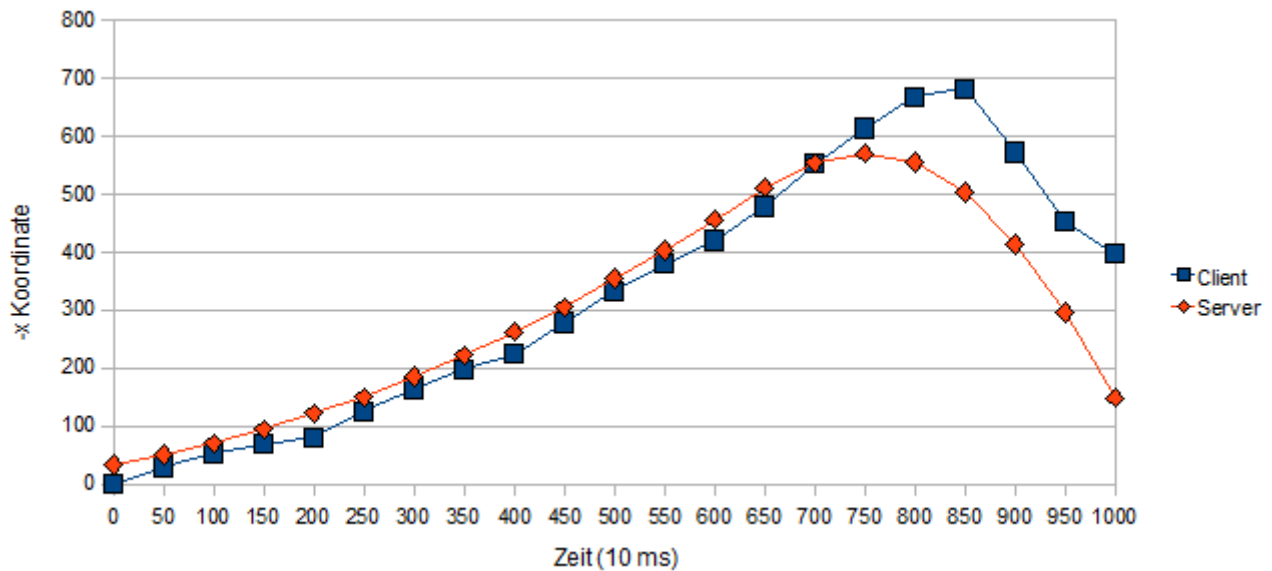


Abbildung 25: Schiffposition Client/Server - lineare Interpolation - 1s Lag

Führt man nun den gleichen Test mit aktivierter linearer Interpolation durch, werden die Vorteile dieses Systems sofort klar. Obwohl die Pakete mit einer Verzögerung von einer Sekunde beim Client auftauchen, wird die Position des Schiffes nahezu perfekt auf dem Client angezeigt.

Hinzu kommt, dass trotz des Lags keinerlei ruckartige Bewegungen beim Client auftreten. Selbst Kursänderungen wie etwa beim Zeitindex von 8s zu sehen, werden in flüssige Bewegungen umgesetzt.

Anhang A - Quellenverzeichnis

Literatur:

Networking and Online Games - Grenville Armitage, Mark Claypool, Philip Branch – Wiley Verlag
– ISBN: 0-470-01857-7

[Co@ch](#) Netzwerktechnik – Carsten Harnisch, Bärbel Ruschitzka, Stephan Cochius, Joachim-Friedrich Geisler – bhv Verlag – ISBN: 3-89360-230-5

Massive Multiplayer Game Development – Thor Alexander – Charles River Media – ISBN: 1-58450-243-6

Massive Multiplayer Game Development 2 – Thor Alexander – Charles River Media – ISBN: 1-58450-390-4

Artikel:

Advanced WinSock Multiplayer Game Programming: Multicasting

Dead Reckoning – Latency Hiding for Network Games

Dealing with Lag and Latency in Network Games

Defeating Lag with Cubic Splines

Designing Fast-Action Games For the Internet

Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization

The Internet Sucks

Unreal Networking Architecture

RFCs:

RFC 768 - User Datagram Protocol

RFC 791 – Internet Protocol

RFC 793 - Transmission Control Protocol

RFC 1577 - Classical IP and ARP over ATM

RFC 1662 - PPP in HDLC-like Framing

RFC 2309 - Recommendations on Queue Management and Congestion Avoidance in the Internet

ISOs:

ISO 7498-1 - Information technology - Open Systems Interconnection - Basic Reference Model - The Basic Model

Wikipedia Artikel:

Asynchronous Transfer Mode

ICMP

Internet

Internet Protocol

IPv4

Lag

OSI model

Transmission Control Protocol

User Datagram Protocol

Webseiten:

<http://www.internetpulse.net/>

<http://www.internettrafficreport.com/>

<http://www.wikihow.com>

Anhang B - Glossar

ADSL	Asymmetric Digital Subscriber Line – Zur Zeit die verbreitetste Technik die für Breitbandverbindungen eingesetzt wird.
bps	bits per second – Einheit mit der die Datenrate angegeben wird.
Downstream	Bezeichnet den Datenfluss von einem Sender zum empfangenden Host.
Hop	Als Hop wird ein der „Sprung“ von einem Knoten zu einem anderen innerhalb eines Netzwerkes bezeichnet.
IDE	Integrated Development Environment – Anwendungen zur Entwicklung von Software, die in der Regel einen oder mehrere Editoren (graphisch und textbasiert), Compiler, Linker, Debugger, etc. vereinen.
Link	Unter einem Link versteht man eine Verbindung zwischen zwei Knoten innerhalb eines Netzwerkes.
MMOG	Massive Multiplayer Online Game – Ein Spiele, welches eine persistente Umgebung enthält, mit welcher der Spieler interagieren kann. Dabei ist während des Spielens eine aktive Verbindung zum Server notwendig.
MMORPG	Massive Multiplayer Online Role Playing Game – Ein MMOG, mit Elementen eines Rollenspiels.
Multiplexing	Ein Verfahren um mehrere Signalquellen über eine Verbindung zu übertragen.
Upstream	Bezeichnet den Datenfluss von einem Empfänger zum sendenden Host.

Anlage 1 – Network Integration – Background Information

Anlage 2 – Network Integration - SRS

Anlage 3 – Network Integration - Analysis

Anlage 4 – Network Integration – Integration Plan

Anlage 5 – Network Integration – Integration Test