UNIVERSITÄT DUISBURG-ESSEN

FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN

ABTEILUNG INFORMATIK

Bachelorarbeit

Design, Implementation and Evaluation of a Reliable UDP Framework

Lorenz Schwittmann

Fakultät für Ingenieurwissenschaften
Abteilung Informatik
Universität Duisburg-Essen

29. September 2010

Betreuer:
Dr. Arno Wacker
Dipl.-Inform. Sebastian Holzapfel

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Massively Multiplayer Online Games (MMOGs) haben in den letzten Jahren stetig an Popularität gewonnen. Weltweit geben Spieler mehrere Milliarden US-Dollar aus[1], um an solchen Spielen teilzunehmen. Diese Spiele zeichnen sich dadurch aus, dass mehrere tausend Spieler in der gleichen Welt zusammen interagieren. Dadurch unterscheiden sie sich von herkömmlichen Multiplayer Spielen, in denen die Anzahl der Teilnehmer meist weit unter 100 liegt.

Zu einem großen Teil wird für diese MMOGs das Client-Server Modell verwendet. Die Spieler starten dabei auf ihren Rechnern Clients, welche zu einem Server verbinden. Dieser koordiniert die Clients und leitet das Spiel. Mit steigender Spieleranzahl verbraucht dieses Modell jedoch zunehmend Ressourcen auf diesem Server. Als Betreiber eines Spiels muss man demnach sicherstellen, dass genügend Ressourcen (z.B. Rechenzeit und Speicher) vorhanden ist, um den Nutzern eine hohe Interaktivität zu ermöglichen.

Eine Alternative bietet hier ein Peer-To-Peer basiertes Modell. Bei diesem verbinden sich die Clients der Spieler nur untereinander und teilen sich anfallende Berechnungen. Eine kostenintensive Unterhaltung von Servern entfällt somit. Dieser Ansatz wirft jedoch, im Gegensatz zu Server basierten Spielen, diverse Probleme auf, welche derzeit Gegenstand der Forschung sind. Diese reichen von Konsistenz über Betrugsmöglichkeiten bis hin zum Verbindungsaufbau.

Die Universitäten Duisburg-Essen, Hannover und Mannheim arbeiten im peers@play Projekt an der Entwicklung einer Middleware für verteilte Massively Multiplayer Online Games, welche Peer-to-Peer basiert arbeitet.

[1] Laut einer 2009 von Newzoo BV durchgeführten Umfrage gaben alleine die Amerikaner 3,8 Milliarden USD dafür aus. http://www.gamesindustry.com/company/542/service/2434

1.1 Motivation

Bei einem Peer-to-Peer basierten Modell läuft auf den Rechnern der Nutzer jeweils ein Client, welcher sich zu anderen Teilnehmern des Spiels verbindet. Hier besteht ein großer Unterschied zum Client-Server Ansatz, bei dem die Clients immer nur nach außen Verbindungen aufbauen, nicht jedoch eingehende Verbindungen haben. Heutzutage ist es durchaus geläufig, dass hinter einem DSL Anschluss mit einer IP-Adresse mehrere Computer mit dem Internet verbunden sind. Dies widerspricht der Annahme[Pos81a], dass einer IP-Adresse jeweils maximal ein Host zugeordnet ist.

Damit dennoch alle internen Rechner bidirektional Pakete mit entfernten Rechnern über das Internet austauschen können, werden Techniken eingesetzt, die in Kapitel 2 genauer erläutert werden. Diese haben jedoch den Nachteil, dass sie einkommende Verbindungen zum Teil wesentlich erschweren. Dies ist kein Problem beim Server-Client-Ansatz, da die Clients dabei immer nur Verbindungen nach außen herstellen.

Wie sehr das Aufbauen einer Verbindung zu einem Client erschwert wird, hängt maßgeblich vom verwendete Transportprotokoll ab. Mit dem Transmission Control Protocol (TCP) ist es in weniger Fällen möglich, eine Verbindung aufzubauen, als mit dem User Datagram Protocol (UDP). Genauer wird dies in Kapitel 2 beschrieben. Im Gegensatz zu UDP bietet TCP jedoch mehr Funktionalität. Es garantiert unter anderem, dass die Daten in der richtigen Reihenfolge ankommen und bei Paketverlust eine erneute Übertragung durchgeführt wird. Für Entwickler wäre es von Vorteil, wenn sie eine Schnittstelle hätten, die diese Eigenschaften anbietet und entsprechend der jeweiligen Möglichkeiten intern TCP oder UDP zum Verbindungsaufbau einsetzt. Wird UDP gewählt, so müssen die fehlenden Eigenschaften von dieser Schicht implementiert werden.

Der Entwickler hat somit immer die Zusicherung, nach dem Verbindungsaufbau einen verlässlichen Datenstrom zu haben, ohne sich um den internen Ablauf kümmern zu müssen.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist es, ein Framework (im Folgenden als RUDP-NG bezeichnet) zu entwickeln, welches aufbauend auf UDP verlässliche Verbindungen implementiert. Dabei soll es die von TCP bekannten Eigenschaften wie Verlässlichkeit (erneute Übertragung bei Paketverlust) und Flusskontrolle implementieren.

Bei der Implementierung wird das Gears4Net Framework verwendet, welches zum Entwickeln von verteilten Protokollen entworfen wurde. [SWHW09]

Die Arbeit umfasst drei Bereiche. Zunächst sollen die Spezifikation erarbeitet werden, die dieses RUDP-NG Framework haben muss. Aufbauend auf diesen Spezifikation wird das Framework dann im Rahmen des peers@play Projektes implementiert. Im letzten Schritt wird die Implementierung unter verschiedenen Bedingungen mit bereits existierenden Implementierungen verglichen.

Dies sind zum einen Compound TCP, welches als eine relativ neue TCP Variante einen optimalen Wert vorlegen soll, und lidgren-network-gen3[2], was ein ebenfalls auf UDP basierendes verlässliches Transportprotokoll ist.

1.3 Strukturierung der Arbeit

Zunächst wird in Kapitel 2 die Motivation dieser Arbeit ausführlicher dargelegt. Es werden die verschiedene NAT-Systeme beschreiben und Verfahren erläutert, wie durch diese Verbindungen aufgebaut werden können.

In Kapitel 3 werden die Spezifikationen des neuen Protokolles vorgestellt und diverse Designentscheidungen beschrieben. Da RUDP-NG das Gleiche leisten soll wie TCP, wird dabei auch auf die TCP Spezifikationen zurückgegriffen.

Darauf aufbauend wird in Kapitel 4 auf die Implementierung im Rahmen des peers@play Projektes eingegangen. Es werden die Interfaces und die allgemeine Architektur erläutert sowie einzelne Implementierungsdetails gezeigt.

Abschließend wird in Kapitel 5 mit Messungen die Implementierung mit Anderen verglichen. Es wurden verschiedene Parameter verändert, um unterschiedliche Netzwerkumgebungen zu simulieren.

[2] http://code.google.com/p/lidgren-network-gen3/

2 Network Address Translation

Im Regelfall vergeben Internet Service Provider (ISP) ihren Privatkunden nur eine IP-Adresse. Da jedoch mit der steigenden Bedeutung des Internets immer häufiger mehr als ein Computer pro Haushalt verwendet wurde, wurden "DSL-Router" immer häufiger eingesetzt. Diese Geräte vereinen mehrere Funktionen. Zum einen enthalten sie einen Switch, welcher mehrere PCs zu einem LAN zusammenschließen kann, zum anderen ein DSL-Modem, welches für die Verbindung ins Internet verantwortlich ist.

In der Kombination dieser beiden Komponenten liegt die Motivation für Network Address Translation (NAT). Die internen Rechner haben eine private IP-Adresse, die nicht ins Internet geroutet wird [RMK+96]. Der Router muss demnach bei ausgehenden Paketen die Quelladresse im Header durch die eigene externe IP-Adresse ersetzen. Umgekehrt müssen aber auch einkommende Pakete zurückübersetzt werden, um den gewünschten internen Rechner zu erreichen.

Würde jeder Rechner nur jeweils mit einem Host im Internet kommunizieren, so wäre dies auf der Netzwerkebene lösbar. Auf dieser findet mittels des Internet Protokolls (IP) eine Host zu Host Kommunikation statt [Pos81a]. Der Router hätte eine Tabelle, die eine externe in eine interne IP-Adresse umwandeln könnte. Dies ist jedoch eine Annahme, die man nicht treffen kann. Insbesondere bei beliebten Seiten ist es wahrscheinlich, dass mehrere interne Rechner mit der gleichen externen IP-Adresse Pakete austauschen.

Aus diesem Grund wird die über der Netzwerkschicht liegende Transportebene verwendet, um Pakte richtig zuordnen zu können. Auf dieser findet eine Kommunikation von Applikation zu Applikation statt. Dazu fügt diese Schicht den Paketen mindestens einen Quell- und einen Zielport hinzu, der jeweils, zusammen mit der IP-Adresse, genau eine Applikation auf einem Host adressiert.

Bei einem ausgehenden Paket ändert der NAT implementierende Router die Quelladresse auf die eigene externe Adresse. Der Quellport kann ebenfalls verändert werden. Gleichzeitig wird in einer Tabelle ein Eintrag angelegt, welcher aus Quellport, Zielport, Zieladresse und interner Quelladresse besteht. Erreicht ein Paket aus dem Internet den DSL-Router, so wird es entsprechend der Tabelle übersetzt und an den richtigen internen Rechner weitergeleitet.

2.1 Beispiel

Es gibt einen Rechner mit der IP-Adresse 192.168.1.2, welcher sich zusammen mit einem DSL-Router in einem LAN befindet. Der DSL-Router hat dabei die externe IP-Adresse 192.0.2.42.

Der Rechner schickt ein Paket (Quellport 5789, Zielport 80, Quelladresse 192.168.1.2) an 192.0.2.23. Der DSL-Router ändert in diesem sowohl Quellport als auch Quelladresse auf 8741 bzw. 192.0.2.42 und legt in seiner Übersetzungstabelle den Eintrag (5789, 80, 192.0.2.23, 192.168.1.2) an.

Trifft nun eine Antwort von 192.0.2.23 auf dem Port 8741 ein, so wird mittels des Eintrages festgestellt, dass dieses an 192.168.1.2 auf Port 5789 weitergeleitet werden muss. Die entsprechenden Felder werden im Paket modifiziert und es wird an den Rechner geschickt.

2.2 NAT-Varianten

Von diesem grundlegenden Ansatz ausgehend gibt es mehrere Variationen, die sich in der Art und Weise der Portzuordnung und Weiterleitung von Paketen unterscheiden. Die folgende Typen wurden in [AJ07] beschrieben.

2.2.1 Portzuordnung

Die Portzuordnung beschreibt, welcher externe Port für ein ausgehendes Paket verwendet wird.

Endpunktunabhängige Zuordnung: Von einem internen Rechner ausgehende Pakete werden werden immer zu dem gleichen externen Port übersetzt. Dabei ist sowohl die Zieladresse als auch der Zielport unerheblich.

Adressenabhängige Zuordnung: Von einem internen Rechner ausgehende Pakete werden für die gleiche Zieladresse immer an den gleichen externen Port übersetzt. Dies bedeutet, dass, verglichen mit endpunktunabhängiger Zuordnung, nur noch der Zielport variieren darf. Weicht die Zieladresse ab, so wird ein anderer externer Port gewählt.

Adressen- und portabhängige Zuordnung: Bei diesem letzten Typ wird für jeden Endpunkt (bestehend aus Zielport und Zieladresse) ein neuer externer Port gewählt.

2.2.2 Filterverhalten

Getrennt von der Zuordnung der externen Ports zu internen Rechnern wird das Filterverhalten betrachtet. Dabei wird eingeschränkt, welche bei einem externen Port ankommende Pakete zu dem dazugehörigen internen Rechner weitergeleitet werden. Damit diese Filterung jedoch überhaupt erst in Kraft tritt, muss zuvor durch ein ausgehendes Paket ein externer Port mit einem internen Rechner verknüpft worden sein.

Endpunktunabhängige Filterung: Bei dieser Filterung wird jedes Paket durchgelassen. Es reicht demnach aus, ein einzelnes Paket an einen beliebigen Rechner jenseits des NAT-Systems zu schicken, um von beliebigen anderen Rechner erreicht zu werden.

Adressenabhängige Filterung: Pakete von einer IP-Adresse A werden nur dann weitergeleitet, wenn zuvor der interne Rechner an A ein Paket geschickt hat. Dabei ist jedoch der Port, auf welchem A das Paket geschickt wurde irrelevant.

Adressen- und portabhängige Filterung: In dieser schärfsten Form der Filterung werden nur dann eingehende Pakete mit der Quelladresse A und dem Quellport a weitergeleitet, wenn zuvor der interne Rechner ein Paket mit der Zieladresse A und Zielport a versandt hat.

2.3 NAT-Traversal

Alle NAT-Varianten erlauben es problemlos, mit einem Host eine Verbindung aufzubauen, der selber keine NAT-Systeme verwendet. Dies ist im normalen Client-Server Ansatz der Fall. Da jedoch in Peer-To-Peer Netzwerken die Rechner der Benutzer jeweils Verbindungen zu anderen Benutzern aufbauen, dürfte es häufig vorkommen, dass 2 Rechner, welche jeweils durch eine NAT-Implementierung vom Internet getrennt sind, eine Verbindung miteinander aufbauen sollen.

Mittels des in [RMMW08] beschriebenen Protokolles kann jeder Knoten im Netzwerk die Filterung und Portzuordnung seines NAT-Systems feststellen. Diese Information kann er über bereits bestehende Verbindungen im Peer-To-Peer Netzwerk bekanntmachen. Startet nun ein Knoten einen Verbindungsaufbau, so weiß er dadurch sowohl seine NAT-Variante als auch die des Zielhosts.

Die Vorgehensweise für einen Verbindungsaufbau hängt dabei von der Kombination der beiden NAT-Varianten ab. Hat man beispielsweise zwei Benutzer, deren NAT-System jeweils eine endpunktunabhängige Portzuordnung aber eine adressenabhängige Filterung hat, so kann keiner direkt zum Anderen eine Verbindung aufbauen. Die gesendeten Pakete würden jeweils an der Filterung des Anderen scheitern. In einem solchen Fall kann Hole Punching verwendet werden, um dennoch eine Verbindung aufzubauen.

Beim Hole Punching fordert ein Rechner A den entfernten Rechner B zum Erzeugen eines Eintrages in der Tabelle seines NAT-Systemes auf. Diese Aufforderung kann wieder mittels bestehender Verbindungen zum Peer-to-Peer Netzwerk zu B weitergeleitet werden. Sobald B die Nachricht erhalten hat, wird dieser einen Verbindungsaufbau zu A initiieren. Dieser wird zwar fehlschlagen, da As NAT-System diese Nachricht nicht durchlässt, aber er erzeugt dabei einen Eintrag in der Tabelle von Bs NAT-System. Dieser wird von A verwendet, um in einem dritten Schritt eine Verbindung aufzubauen (siehe Abbildung 2.1).

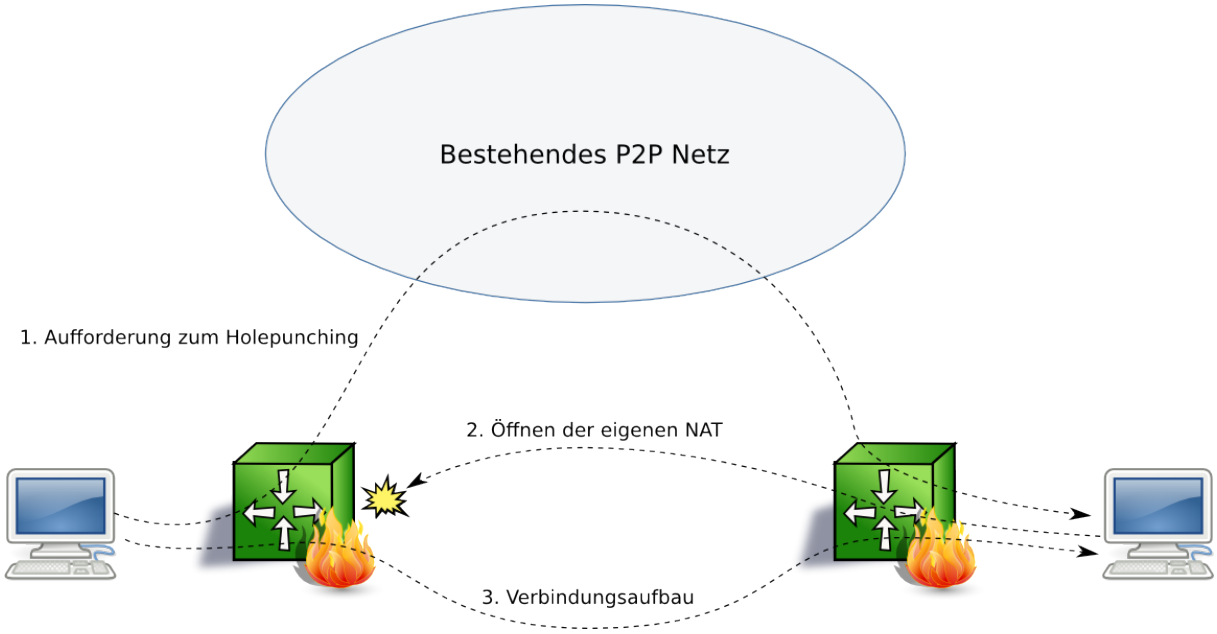

Abbildung 2.1: Hole Punching

In diesem Szenario liegt die Motivation für RUDP-NG. Zwar lässt sich prinzipiell auch mit TCP Hole Punching betreiben, allerdings nutzen einige NAT-Implementierungen zusätzliche Informationen in den Headern der TCP-Pakete, was dessen Einsatz verhindert.

2.3.1 TCP Holepunching

Schaut man sich den Header von TCP-Paketen an (siehe Abbildung 2.2), so hat dieser mehrere Informationen, die Holepunching verhindern können.

Zum einen wird ein Verbindung über einen Handschlag mit drei Paketen aufgebaut. Das Erste hat dabei nur die SYN-Flag gesetzt und eine zufällige Sequenznummer x. Darauf antwortet die Gegenseite mit einem Paket, was sowohl die SYN als auch die ACK-Flag gesetzt hat. Die Sequenznummer startet auch hier zufällig mit dem Wert y. Gleichzeitig bestätigt dieses Paket durch das ACK-Feld mit dem Wert $x+1$ den Empfang des ersten Paketes. Mit einem letzten SYN/ACK Paket schließt der initiale Sender den Verbindungsaufbau ab. Die Sequenznummer ist dabei $x + 1$ und bestätigt wird $y + 1$. [Pos81b, Sek 3.4]

Da beim Holepunching beide Seiten einen eigenständigen Handschlag ausführen,

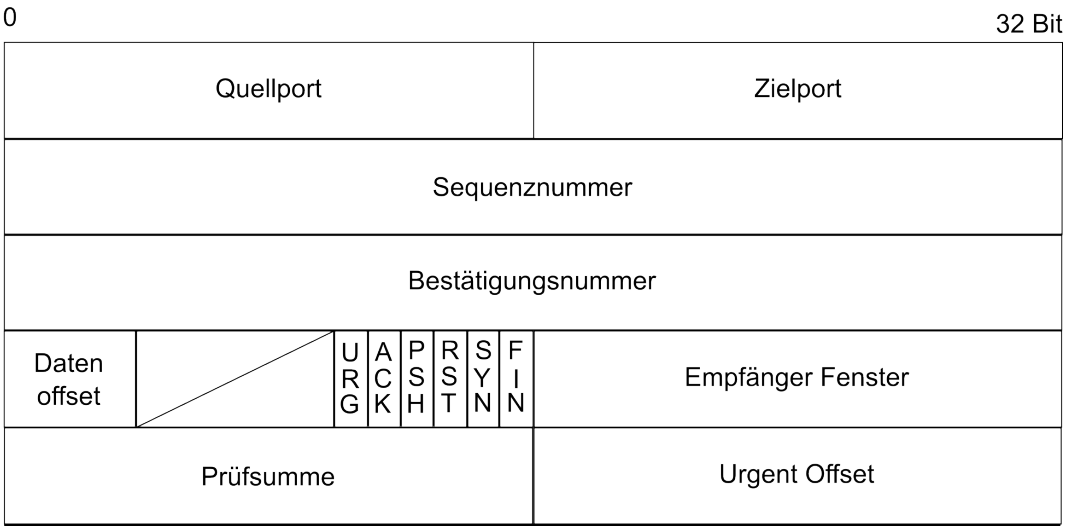

Abbildung 2.2: Der TCP-Header

kann erkannt werden, dass die Pakete nicht zur gleichen Verbindung gehören. So kann der NAT implementierende Router von Rechner B gespeichert haben, dass in Schritt 2 ein SYN-Paket ausging. Laut dem oben erläuterten Handschlag müsste darauf ein SYN/ACK folgen. Kommt nun in Schritt 3 allerdings ein SYN-Paket an, so können diese beiden nicht zur gleichen Verbindung gehören - die NAT-Implementierung kann in solch einem Fall vorsehen, dieses nicht an B weiterzuleiten.

Um das zu umgehen wäre es denkbar, dass B die Sequenznummer seines Paketes mittels eines Sniffers aufzeichnet und über das Peer-to-Peer Netzwerk an A schickt. Dieser könnte damit ein SYN TCP-Paket zusammenbauen und es über einen Raw-Socket[1] an sich selbst schicken. Das Betriebssystem würde dies als einen normalen eingehenden Verbindungsaufbau ansehen und mit einem korrekten SYN/ACK Paket antworten, welches Bs NAT-System passieren würde.

Dieser Ansatz scheitert jedoch aus praktischen Gründen. Unter Windows, welches die Zielplatform für das peers@play Projekt ist, ist es seit Windows XP SP2 nicht mehr möglich, über einen Raw-Socket TCP-Pakete zu schicken[2].

Ein weiteres Problem beim Einsatz von TCP Holepunching ist das RST-Flag.

[1] Ein Raw-Socket arbeitet auf IP-Ebene, so dass beliebige IP-Pakete über diesen generiert werden können

[2] Siehe http://msdn.microsoft.com/en-us/library/ms740548%28VS.85%29.aspx

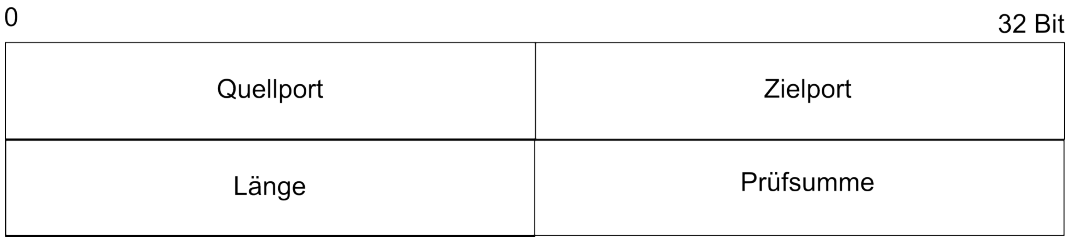

Abbildung 2.3: Der UDP-Header

Dieses wird gesetzt, wenn der Abbruch einer Verbindung erzwungen werden soll. Trifft in Schritt 2 bei As NAT-System ein TCP-Paket ein, welches entweder nicht zuordenbar ist, da es keinen Eintrag in der Tabelle für diesen Port gibt, oder gefiltert wird, so kann mit einem TCP-Paket mit gesetztem RST-Flag antworten. Dies kann allerdings dazu führen, dass Bs NAT-System diese Verbindung als beendet ansieht und folgende Pakete nicht mehr an A weiterleitet. Schritt 2 würde in solch einem Fall nicht einen dauerhaften Eintrag in der Tabelle des eigenen NAT-System verursachen - das zurückkommende RST-Paket führt zur umgehenden Löschung des Eintrages.

2.3.2 UDP Holepunching

Da UDP an sich verbindungslos ist, gibt es keinerlei Möglichkeiten anhand des Headers (siehe Abbildung 2.3) zu sagen, ob weitere Pakete folgen oder nicht.

Aus diesem Grund gibt es für dessen Einträge in der NAT-Tabelle nur einen Timer, der alte Einträge entfernt, zu denen über einen gewissen Zeitraum kein Paket mehr übertragen wurden. Im zweiten Schritt des Hole Punchings bleibt somit der Eintrag bestehen und kann anschließend für einen Verbindungsaufbau genutzt werden.

In einer in [FSK05] durchgeführte Messung wurde festgestellt, dass UDP Hole Punching im Vergleich zu TCP eine höhere Erfolgschance von $\sim 20\%$ hat.

3 Protokoll Entwurf

Da RUDP-NG das Gleiche leisten soll wie TCP, wurden die Spezifikationen von TCP als Grundlage für RUDP-NG verwendet. Jedoch wurde TCP seit seiner Veröffentlichung im Jahre 1974 [CDS74] immer wieder durch neue RFCs an den Stand der Technik angepasst, ohne jemals radikal neu entworfen worden zu sein. Dies wäre bei der weiten Verbreitung des Protokolls undenkbar gewesen. Es wäre zu Inkompatibilitäten gekommen, was verhindert werden musste. Infolgedessen ist das gesamte Protokoll sehr komplex und unübersichtlich geworden, weshalb RUDP-NG sich nur grob an TCP Mechanismen orientiert.

3.1 Einordnung im Schichtenmodell

Im Folgenden wird das hybride Schichtenmodell aus [Tan02] verwendet. Dieses besitzt 5 Schichten:

1. Physikalische Schicht
2. Sicherungsschicht
3. Vermittlungsschicht
4. Transportschicht
5. Anwendungsschicht

RUDP-NG liegt oberhalb der Netzwerk- und Transportschicht. Da es das Gleiche leistet wie TCP, welches alternativ zu UDP auf der Transportschicht verwendet werden kann, kann es als logische Erweiterung dieser gesehen werden, obwohl es aus Sicht des Betriebssystems Teil der Applikation ist. Da der Ort der Implementierung jedoch variieren kann (denkbar wäre es, komplett TCP über Raw-Sockets in einer Applikation neu zu implementieren), wird im folgenden die logische Sichtweise vertreten.

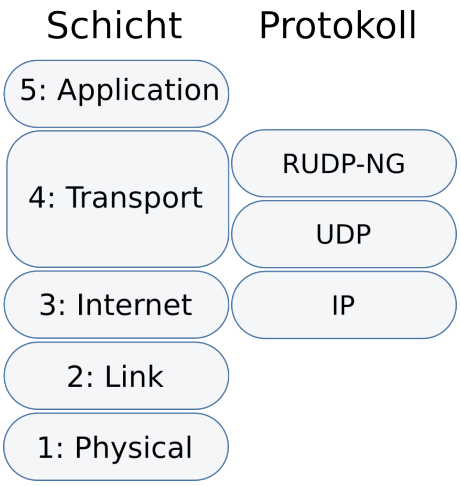

Abbildung 3.1: Das Schichtenmodell

Wie in Abbildung 3.1 dargestellt wird, werden keine Annahmen über die Layer unterhalb von IP oder oberhalb von RUDP-NG gemacht. Denkbar wäre der Einsatz von Ethernet, Token Ring oder FDDI auf Linkebene. Dass allgemeine Applikationen diese Protokoll benutzen können und der Einsatz nicht auf das peers@play Projekt beschränkt ist, wird durch eine lose Koppelung mittels eines Interfaces erreicht (siehe Abschnitt 4.1).

3.2 Datenstrom

RUDP-NG stellt für die oberen Schichten einen verlässlichen Datenstrom bereit. Das bedeutet, dass Bytes blockweise von A nach B verschickt werden und garantiert wird, dass diese in der richtigen Reihenfolge und ohne Lücke der oberen Schicht auf der Gegenseite überreicht werden. Insbesondere muss ein Übertragungsfehler erkannt und mit erneuter Übertragung behoben werden. Ist es trotz Fehlererkennung und Neuübertragung nicht möglich, Daten zum Zielhost zu verschicken - da dieser zum Beispiel abgestürzt ist - so wird dies der oberen Schicht mitgeteilt.

Es gibt jedoch keine Garantie, dass Blöcke, die von einer oberen Schicht zum Versenden übergeben werden, auch in der gleichen Form auf der Gegenseite ankommen, da diese zu groß sein könnten, um sie in einem Paket zu verschicken. Dies ist durch die Maximum Transmission Unit begründet, welche es notwendig machen kann, dass ein Byteblock vor dem Versenden zerlegt werden muss.

3.2.1 Maximum Transmission Unit

Durch die in den unteren Schichten verwendeten Übertragungsmedien (Ethernet, PPPoE/DSL) kann ein IP-Paket eine feste Größe, die Maximum Transmission Unit (MTU), nicht überschreiten. Ist ein IP-Paket größer als dieser Wert, so muss es vor der Übertragung in mehrere kleinere Pakete fragmentiert werden. Dies kann, aufgrund der Heterogenität der im Internet verwendeten Übertragungsmedien, prinzipiell bei jedem Router zwischen Quell- und Zielhost passieren. Erst beim Zielhost werden diese Fragmente wieder zum eigentlichen IP-Paket zusammengebaut. [Pos81a]

Der Zusammenbau findet im Betriebssystem statt. Erst wenn das IP-Paket vollständig ist, also alle Fragmente eingetroffen sind, wird dieses von der Transportschicht bearbeitet und anschließend an die Applikation weitergereicht. Da es zwischen dem Knoten, der das Paket fragmentiert hat, und dem Empfänger auf IP-Ebene keine Neuübertragungen bei Verlust gibt, kann der Fall eintreten, dass auf das letzte Fragment gewartet wird, was jedoch unwiederbringlich verloren gegangen ist. In solch einem Fall wird die Transportschicht, sofern sie dies unterstützt, eine Neuübertragung des kompletten IP-Paketes durchführen - auch wenn nur ein geringer Teil nicht angekommen ist.

Da sich dieses doppelte Übertragen von Daten negativ auf den Gesamtdurchsatz auswirkt, wird versucht, Fragmentierung zu verhindern. Um das zu erreichen, werden in RUDP-NG die Pakete immer so formatiert, dass sie die MTU nicht überschreiten.

3.2.2 Sequenznummern

Um die oben genannten Anforderungen umzusetzen, wird jedes Byte im Datenstrom mit einer fortlaufenden Sequenznummer versehen. Diese starten bei einem zufälligen Wert, der beim Verbindungsaufbau jeweils der Gegenseite mitgeteilt wird. Dadurch ist ein Korrumpieren des Strom durch einen blind schickenden Angreifer oder noch zirkulierende Pakete früherer Verbindungen unwahrscheinlicher, als wenn diese bei einem festen Wert starten würden.[Pos81b, Seite 26]

In jedem Paket wird die Sequenznummer des ersten Bytes der Nutzdaten mitgesendet, wodurch die Sequenznummer jedes einzelnen Nutzbytes eindeutig definiert ist.

Da für Sequenznummern eine feste Anzahl an n Bits im Header reserviert sind, wiederholen sie sich nach 2^n übertragenen Bytes, was berücksichtigt werden muss. Am Gravierendsten sind die Auswirkungen bei der Überprüfung, ob ein Paket vor einem anderen abgeschickt wurde, um diese in der richtigen Reihenfolge an die obere Schicht zu übergeben.

Gibt es ein Paket A der Länge l mit $0 < l < 2^n$, so erhält dieses die Sequenznummer x und das darauf folgende Paket B die Sequenznummer $x + l$. Steht nun aber der Zähler vor Paket A bei $x = 2^n - 1$, so erhält Paket B die Sequenznummer $l - 1$. Würde die Gegenseite jetzt einen normalen Kleiner-als Vergleich machen, so würde diese fälschlich B vor A ordnen, da $l - 1 < 2^n - 1$.

Hat man beispielsweise $n = 8$ Bit im Header für Sequenznummern reserviert und bekam A, welches die Länge $l = 10$ hat, die Sequenznummer $x = 255$, so erhält das darauf folgende Paket B die Sequenznummer 9. Bei einem normalen Kleiner-als Vergleich würde B somit vor A geordnet werden, da $9 < 255$.

Um dies zu verhindern wird in RUDP-NG bei jedem Vergleich mit einer Sequenznummer a der Zahlenraum gleichmäßig in zwei zusammenhängende Hälften aufgeteilt, die jeweils als Kleiner-als und Größer-als a bezeichnet werden. Graphisch dargestellt wird diese Spaltung in Abbildung 3.2. Dabei gibt es jeweils durch den möglichen Überlauf zwei Fälle zu betrachten.

Betrachtet man zunächst die Kleiner-als Relation, so ist eine Sequenznummer a dann kleiner als eine Sequenznummer b, wenn

- $a < b$ und $b - a < 2^{n-1}$ ODER
- $b < a$ und $a - b > 2^{n-1}$ gilt.

Der zweite Teil der Bedingungen ist jeweils dazu dar, um einen Überlauf zu erkennen; 2^{n-1} ist dabei die oben erwähnte Hälfte des Zahlenraumes.

Betrachtet man beispielsweise den Fall $n = 8$, $a = 50$ und $b = 255$, so gilt $a < b$. Um nun zu überprüfen, ob nun auch a als Sequenznummer kleiner als b ist, betrachtet man die erste Bedingung, da deren erster Teil offensichtlich erfüllt ist. Nach dem

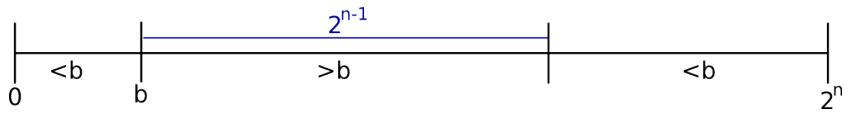

Abbildung 3.2: Visualisierung der Aufteilung des Zahlenraumes.

zweiten Teil müsste gelten $255 - 50 < 128$, was jedoch nicht der Fall ist. Da die Differenz zu groß ist, gilt in diesem Beispiel, dass a größer ist als b.

Dies lässt sich als mathematische Relation aufschreiben. Eine Relation ist dabei eine Menge von Tupeln, wobei a in dieser Relation zu b steht, wenn (a, b) Element dieser Menge ist.

Die Kleiner-als Relation für Sequenznummern mit n Bit in RUDP-NG wird definiert als:

$$R_< := \{y \in \mathbb{N}_0 | y < 2^n\}^2 \bigcap \{(a,b) | (a < b \wedge b - a < 2^{n-1}) \vee (b < a \wedge a - b > 2^{n-1})\} \tag{3.1}$$

Der Teil links der Schnittoperation stellt dabei alle Sequenznummer Kombinationen dar. Dabei werden die Tupel mittels des kartesischen Produktes gebildet. Der Rest setzt sich aus den oben erläuterten beiden Bedingungen zusammen.

Diese Relation erfüllt zwar nicht das Kriterium der Transitivität, was mathematisch gesehen für Ordnungsrelation benötigt wird. Durch Begrenzung der maximal ausstehenden Bytes auf weniger als 2^{n-1} ist diese jedoch praktisch zum Ordnen von Paketen nutzbar.

Darauf aufbauend kann die Größer-als Relation definiert werden als Negation der Kleiner-als Relation ohne Tupel mit gleichen Komponenten.

$$R_> := \left(\{y \in \mathbb{N}_0 | y < 2^n\}^2 \setminus \{a \in \mathbb{N}_0 | (a,a)\}\right) \setminus R_< \tag{3.2}$$

3.2.3 Empfangsbestätigungen

Damit der Datenstrom verlässlich ist, schickt der Empfänger regelmäßig Bestätigungen über den Eingang von Daten. Dadurch erst ist es möglich, einen Übertragungsfehler zu erkennen und anschließend zu beheben. Diese Bestätigungen werden im ACK-Feld des Headers gesendet. Da RUDP-NG bidirektional arbeitet, können

die Bestätigungen somit zusammen mit Daten, die der ursprüngliche Empfänger sendet, übertragen werden. Ein eigenes Paket hierfür zu schicken lohnt sich meistens nicht, da die Summe aus UDP-(8 Bytes) und IP-Header(20 Bytes) ein Vielfaches der Nutzdaten (8 Byte für die bestätigte Sequenznummer) darstellt. Trifft ein Paket mit Daten ein, so wird unter idealen Bedingungen mit der Bestätigung gewartet, bis ein ausgehendes Paket mit Daten verschickt werden soll.

Für TCP gibt es eine Empfehlung [Bra89, 4.2.3.2], wann eine Bestätigung verschickt werden sollte, die zum Teil so übernommen wurde. Ein einzelnes ACK-Paket wird nach dem Empfang eines Paketes mit Nutzdaten verschickt, wenn mindestens eine der folgenden Bedingungen erfüllt ist:

- Es wurde länger als 500 ms auf ein ausgehendes Datenpaket gewartet, welches die Bestätigung hätte mittragen können
- Seit der letzten Bestätigung sind insgesamt Daten mit einem Volumen von mehr als 2 MTU eingetroffen
- Es gibt eine Lücke zwischen Paketen, was entweder auf Paketverlust (siehe Abschnitt 3.6) oder Umsortierung durch das Netzwerk hinweist

3.2.4 Sliding Window

Ein Sliding Window begrenzt die maximale Anzahl der gleichzeitig zu übertragenden Bytes, also derjenigen, die zwar geschickt, deren Empfang jedoch noch nicht bestätigt wurden. Beim Eintreffen jeder Bestätigung wird überprüft, ob durch diese neue Pakete losgeschickt werden dürfen. Eine Bestätigung der Sequenznummer x bedeutet auch gleichzeitig den Empfang aller Bytes, deren Sequenznummer kleiner als x ist (im Sinne der oben erläuterten Relation). Somit lassen sich mit dem ACK-Feld in einem Paket mehrere Nutzdaten-Pakete der Gegenseite gleichzeitig als empfangen vormerken.

Wie groß dieses Fenster ist, wird in TCP durch das Minimum aus Receive und Congestion Window bestimmt.

3.2.4.1 Congestion Window

Das Congestion Window gibt an, wieviele Bytes ohne Empfangsbestätigung gesendet werden dürfen, ohne dass das Netzwerk überlastet wird. Da im Internet

immer von einem Knotenpunkt zum nächsten geroutet wird, kann es praktisch an jedem Knoten zu einem Stau kommen. Das bedeutet, dass die IP-Pakete im Arbeitsspeicher des Routers gehalten werden, bis die richtige ausgehende Leitung wieder frei ist. Da die Router über begrenzte Ressourcen verfügen, können auch nur eine gewissen Anzahl an Paketen dort vorgehalten werden. Ist diese Kapazität erschöpft, so bleibt nichts anderes übrig, als Pakete zu verwerfen. RUDP-NG verwendet, wie TCP, die Information über diese verlorene Pakete, um so Rückschlüsse auf die Kapazität des Netzwerkes zu gewinnen. Treten solche Verluste auf, so wird das Congestion Window reduziert, um das Netzwerk zu entlasten.

In Abschnitt 3.5 wird genauer auf den in RUDP-NG verwendeten Algorithmus eingegangen, der zur Bestimmung des Congestion Windows verwendet wird.

3.2.4.2 TCPs Receive Window

Neben dem Netzwerk gibt es noch den Empfänger, welcher einen limitierenden Faktor darstellt. Bei TCP gibt es deshalb ein Empfangsfenster, welches in jedem Paket angibt, wieviel Platz der Absender noch im Puffer für einkommende Pakete zur Verfügung hat. Überschreitet der Sender dieses Limit, so müssen die Daten auf der Gegenseite aus Platzmangel verworfen werden.

Dieser Puffer wird benötigt, damit Daten bis zum nächsten Lesen durch die Anwendungen zwischengespeichert werden können. Bleibt dieser Lesevorgang aus, da beispielsweise die Applikation gerade Berechnungen durchführt, stauen sie sich dort.

Das Interface der Transportschicht im peers@play Projekt (siehe Abschnitt 4.1.3) übergibt ankommende Daten per Event an obere Schichten, ein explizites Lesen und damit verbundene Zwischenspeichern findet nicht statt. Aus diesem Grunde fällt das aus TCP bekannte Receive Window in RUDP-NG Paketen komplett weg. Stattdessen wird einmalig beim Verbindungsaufbau eine maximale Fenstergröße ausgetauscht. Mit dieser kann ein Empfänger überprüfen, ob ein einkommendes Paket im akzeptierten Bereich ist.

3.3 Maximum Segment Lifetime

Anders als TCP, das in seinen Paketen 32 Bit für Sequenznummern reserviert [Pos81b, Seite 24], verwendet RUDP-NG dafür 64 Bit. Dies ist durch die fortschreitende Entwicklung im Bereich des Netzwerkdurchsatzes begründet. Zum Zeitpunkt der TCP Spezifizierung war noch Ethernet mit 10 Megabit/s Stand der Technik. Bei maximaler Übertragung (unter Vernachlässigung des Overheads durch diverse andere Layer) werden 2^{32} Bytes in

$$\frac{2^{32} \cdot 8\,\text{Bit}}{10 \cdot 1000000\,\frac{\text{Bit}}{\text{s}}} \approx 3435,0\,\text{s} = 57,25\,\text{min} \tag{3.3}$$

übertragen. Dies bedeutet, dass es fast eine Stunde dauert, ehe eine Sequenznummer in einer Verbindung erneut vorkommt. Dass ein Paket solange im Netz verbleibt und anschließend den Strom korrumpiert, ist, zumal es auf IP-Ebene noch die TTL gibt, die die Lebensdauer eines Paketes auf maximal 255 Hops begrenzt, äußerst unwahrscheinlich.

Bei heute erhältlichen 10 Gigabit/s Ethernet schrumpft diese Zeit jedoch auf

$$\frac{2^{32} \cdot 8\,\text{Bit}}{10 \cdot 1000000000\,\frac{\text{Bit}}{\text{s}}} \approx 3,4\,\text{s} \tag{3.4}$$

Diese geringe Zeitspanne verletzt den in [Pos81b] definierten Maximum Segment Lifetime (MSL) Wert, der dort für TCP mit 2 min angegeben ist. Dieser gibt an, welche Zeit vergehen muss, bevor eine Sequenznummer erneut verwendet werden darf. In [JBB92] wurde diskutiert, ob man zusätzliche Bits für Sequenznummern im Header über Optionen erlauben sollte, entschied sich aber aus Kompatibilitätsgründen dagegen und verwendete Zeitstempel zur genaueren Differenzierung.

Um diese Problematik in die ferne Zukunft zu verschieben, verwendet RUDP-NG von vornherein 8 Bytes für Sequenznummern. Berechnet man, ab welcher Übertragungsgeschwindigkeit bei 64 Bit die MSL wieder erreicht wird, so liegt dies bei:

$$\frac{2^{64} \cdot 8\,\text{Bit}}{120\text{s}} \approx 123 \cdot 10^{16}\,\frac{\text{Bit}}{\text{s}} \tag{3.5}$$

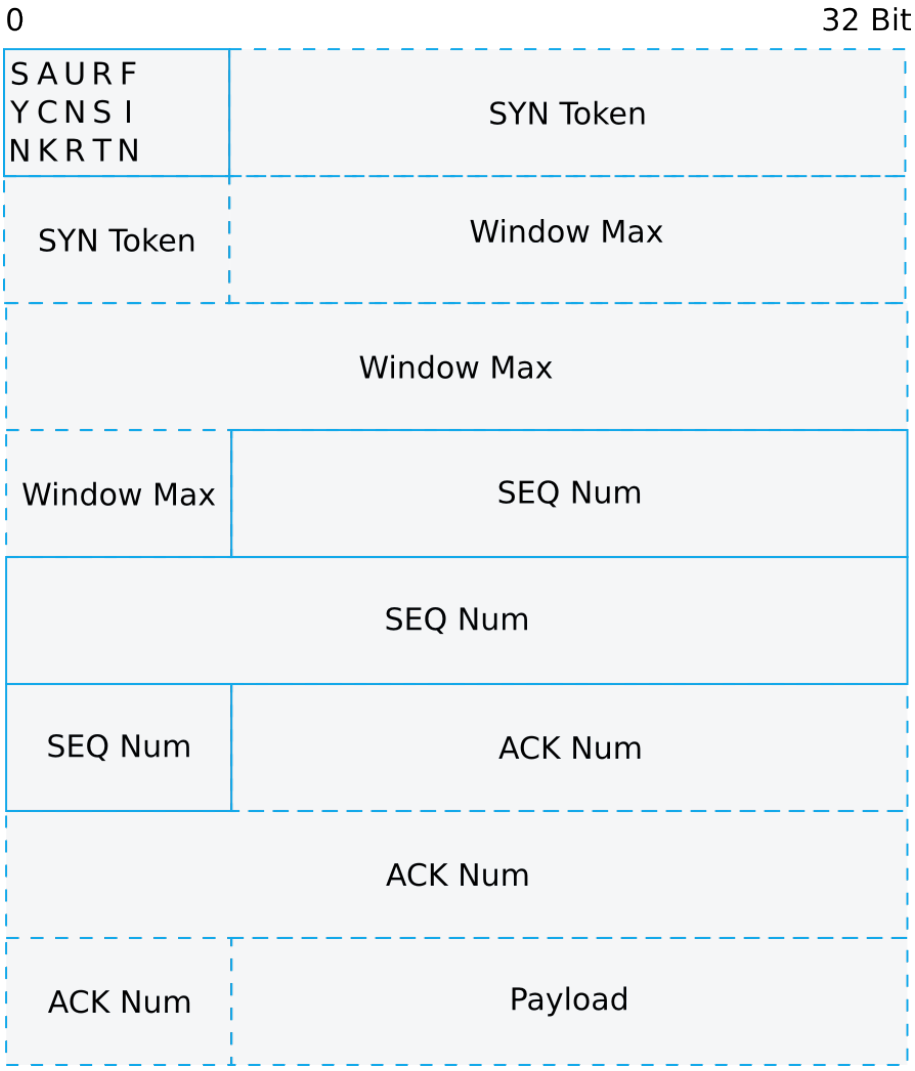

Abbildung 3.3: Mögliche Felder im RUDP-NG Header. Optionale Felder sind gestrichelt.

3.4 Paketaufbau

Da das Protokoll auf UDP aufbaut, sind einige Informationen, die im TCP Paketheader angegeben werden, bereits dort verfügbar, so dass sie nicht im RUDP-NG Header mit verschickt werden müssen. Dies sind Quellport, Zielport und eine Prüfsumme (Siehe Abbildung 2.3). Zwar ist in dem RFC zu UDP angegeben, dass diese Prüfsumme optional ist [Pos80], sie kann jedoch als Anwendungsprogrammierer über eine Socketoption explizit eingeschaltet werden. Für RUDP-NG wird dieses vorausgesetzt.

RUDP-NG Pakete verfügen über ein Byte, welches diverse Flags beinhaltet. Dieses

steht an erster Stelle von jedem Paket und bestimmt maßgeblich den weiteren Paketaufbau. Die einzelnen Flags in diesem heißen SYN, ACK, UNRELIABLE, RST und FIN. SYN wird, wie bei TCP, zum Verbindungsaufbau eingesetzt (siehe Abschnitt 3.8), ACK bedeutet, dass das Paket über ein ACK-Feld im Header verfügt, UNRELIABLE ist bei Paketen gesetzt, die ohne eine Verbindung geschickt werden und RST und FIN beenden eine Verbindung (siehe Verbindungsabbau - Abschnitt 3.9).

Wie oben angemerkt haben die einzelnen Flags Einfluss auf den Paketaufbau. Ist nur das SYN-Flag gesetzt, so handelt es sich um eine Verbindungsanfrage der Gegenseite. In diesem Fall folgen ein 32 Bit Integer, welcher zur Identifikation von RUDP-NG Paketen dient und den Wert 1682131800 hat, und die maximale Fenstergröße. Der Wert wurde eingeführt, da in Tests immer wieder STUN-Pakete [RMMW08] das Initiieren einer Verbindung ausgelöst haben. Mit diesem Wert kann ein initiales RUDP-NG Paket mit einer hohen Wahrscheinlichkeit von einem Paket einer anderen Anwendung unterschieden werden. Es folgt die 8 Byte lange Sequenznummer und, im Falle eines gesetzten ACK Bits, die Sequenznummer der Gegenseite, bis zu der alle Bytes ohne Lücke vorliegen. Den Rest des Paketes macht dann die Nutzlast der oberen Schicht aus. Diese kann auch nicht vorhanden sein, was zum Beispiel beim Senden einer expliziten Empfangsbestätigungen der Fall ist.

Der Quelltext zum Parsen eines RUDP-NG Paketes ist im Anhang unter B.1 zu finden, Abbildung 3.3 zeigt alle möglichen Felder eines RUDP-NG Paketes zusammen.

3.5 Überlaststeuerung

Überlaststeuerung beschäftigt sich mit der Bestimmung des in Abschnitt 3.2.4.1 vorgestellten Congestion Windows. Auch 36 Jahre nach der ersten Veröffentlichung von TCP noch ist dies noch ein aktives Forschungsfeld. Regelmäßig werden neue Verfahren entwickelt, die eine effizientere Auslastung des Netzwerkes im Allgemeinen oder unter bestimmten Bedingungen versprechen. Für RUDP-NG wurde CUBIC [HRX08] zu diesem Zwecke gewählt, da dieses in einem Vergleich mit Compound und New Reno die höchste Übertragungsgeschwindigkeit hatte [ARFK10] und durch seine kubische Funktion einfach zu implementieren ist.

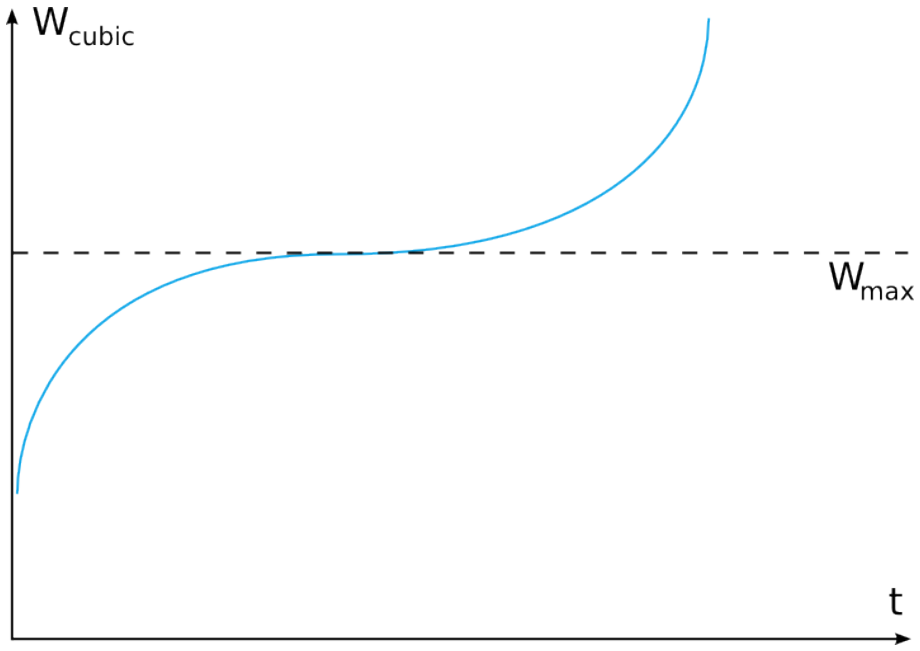

Abbildung 3.4: Kubisches Wachstum des Congestion Window bei CUBIC

Bei CUBIC gibt es neben dem Congestion Window (im Folgenden W_{cubic}) auch noch eine weitere Variable (im Folgenden W_{max}), welcher sich W_{cubic} nach einem Paketverlust vorsichtig annähert. Tritt ein Paketverlust auf, so wird in W_{max} der alte Wert von W_{cubic} um einen Faktor von $1 - \beta$ (mit $0 < \beta < 1$) reduziert gesichert.

$$W_{max} \leftarrow \beta \cdot W_{cubic}$$

Die Fenstergröße wird in Abhängigkeit von der Zeit seit dem letzten Paketverlust bestimmt. Die Funktionsvorschrift dafür lautet gemäß [HRX08]

$$W_{cubic}(t) = C \left(t - \sqrt[3]{W_{max} \frac{\beta}{C}} \right)^3 + W_{max} \tag{3.6}$$

wobei C ein skalierender Faktor und t die Zeit seit dem letzten Paketverlust ist.

Schaut man sich den Graphen der Funktion an (Abbildung 3.4), so kann man 3 Phasen erkennen. In einer ersten Phase wird sich langsam, um das Netzwerk nicht wieder zu überlasten, W_{max} angenähert. Ist dieses Niveau erreicht, so wird auf diesem eine kurze Zeit verharrt, bevor in einer letzten Phase versucht wird, ungenutzte Kapazitäten zu erschließen. Dieses in der letzten Phase auftretende Wachstum wird, sobald das Limit erreicht ist, wieder durch einen Paketverlust

beendet und zur Phase 1 gewechselt.

Zusätzlich werden in RUDP-NG Paketverluste ignoriert, wenn diese auftreten, falls $W_{cubic} < W_{max}$ gilt. Die dahinterstehende Überlegung ist, dass in dieser Phase sich gerade von einem staubedingten Paketverlust erholt und einem sicheren Limit angenähert wird. Tritt nun ein Paketverlust auf, so ist es wahrscheinlicher, dass dieser entweder von dem vorherigen Stau stammt (in [Pax99] wurde gezeigt, dass durch Staus verursachte Paketverluste gebündelt auftreten) oder ein mediumbedingter Verlust ist, bei welchem ein Runterfahren des Congestion Windows nicht indiziert wäre.

Trotz dieser Regel kann sich W_{cubic} wirklich verringern und ist nicht auf den Wertebereich zwischen $W_{cubic}(0)$ und W_{max} beschränkt. Da bei einem Stau es zu kontinuierlichen Paketetverlusten kommt, bis dieser behoben wird, führt dies dazu, dass bei $W_{cubic} = W_{max}$ es unmittelbar zu einer Verringerung des Fensters kommt. In diesem Fall reduziert sich W_{max} auf $\beta \cdot W_{max}$.

Ohne diese Vorkehrung verzehnfachte sich in Messungen die Übertragungszeit bei einer zufälligen Verlustrate von 1%.

Diese Regel gilt nicht während der Initialisierung.

Bei einer neuen Verbindung wird initial eine Form des Slow Start Algorithmus [APB09, Sek. 3.1] verwendet. Bei diesem wird das Congestion Window initial auf 2 MTU gesetzt. Für jedes bestätigte Paket wird dieses um eine MTU erhöht. Es handelt sich demnach um ein exponentielles Wachstum, da durch das höhere Congestion Window mehr Pakete gleichzeitig verschickt werden können, welche beim Bestätigen wiederum eine Erhöhung des Congestion Windows verursachen.

Tritt ein Paketverlust ein, so wird zu CUBIC gewechselt. Das aktuelle Fenster wird mit β multipliziert in W_{max} gespeichert und es wird wie oben beschrieben verfahren.

Ein letztes Problem in diesem Zusammenhang stellen Verbindungen dar, über welche längere Zeit keine Daten mehr übertragen wurden. Da der letzte Paketverlust in diesem Fall lange zurückliegt, wird W_{cubic} Werte annehmen, die das Netzwerk unter keinen Umständen verkraftet. Liegen dann wieder Daten zum Versenden vor, kommt es durch die enorme Datenmenge zu einem sehr hohen Paketverlust und möglicherweise auch einer Beeinträchtigung anderer Netzwerkteilnehmer. Um dies

zu verhindern wird, in Anlehnung an [APB09, Sek 4.1], beim Senden von Daten nach einer Pause von mehr als RTO Zeiteinheiten (siehe Abschnitt 3.6) wieder mit dem Slow Start Algorithmus begonnen. Auf diese Weise werden die möglicherweise geänderten Netzwerkbedingungen berücksichtigt.

3.6 Erneutes Übertragen

Die Entscheidung, ein Paket mit Nutzdaten erneut zu übertragen, wird immer vom Sender getroffen. Um diese Entscheidung zu treffen, muss bei TCP eine von zwei Bedingungen erfüllt sein.

1. Es wurden 4 Pakete ohne Payload mit gleichem ACK Wert empfangen [APB09, Sek 3.2]
2. Es wurde nach RTO Zeiteinheiten noch kein Bestätigung empfangen

Diese Auslöser für eine erneute Übertragung wurden in RUDP-NG übernommen. Die erste Bedingung erklärt sich im Zusammenhang mit der unter Abschnitt 3.2.3 gestellten Forderung, dass ein ACK geschickt werden soll, wenn es eine Lücke beim Empfang von Paketen gibt. Schickt A beispielsweise die Pakete $\{p_i \mid 0 < i < 5\}$ los und kommen diese als p_1, p_3, p_4 und p_2 an, so werden durch das verspätete p_2 2 ACK-Pakete geschickt: jeweils beim Eintreffen von p_3 und p_4.

Hier zeigt sich die eigentliche Problematik, die dieser ersten Regel zugrunde liegt. Da man nicht in die Zukunft schauen kann, lässt sich nicht sagen, ob ein Paket sich nur verspätet oder endgültig verloren ist. Hier gilt es abzuwägen; wartet man zu lange mit der Übertragung, so wirkt sich dies negativ auf die Latenz aus, wartet man zu kurz, so wirkt sich dies negativ auf den Durchsatz aus, da Daten doppelt übertragen werden. Zwar wurde die Wahl auf gerade 4 Pakete mit gleichem ACK-Wert als Indikator für ein verlorenes Paket für TCP ohne eine fundierte Datenlage gewählt[Pax99], da jedoch die in [Pax99] erhobenen Daten nahelegen, dass dies eine vernünftige Abwägung ist, wurde dies für RUDP-NG übernommen.

Die Festlegung eines Zeitlimits, ab dem ohne eintreffende Bestätigung der Sender Pakete erneut schickt (RTO - Retransmission Timeout), ist notwendig, da die erste Bedingung nicht in allen Fällen hilft. Das letzte Paket, was die Transportschicht im Puffer hat, könnte anderenfalls auf unabsehbare Zeit verzögert werden. Geht gerade dieses verloren, so würde der Verlust erst bemerkt, wenn die obere Schicht

erneut Daten zum Versenden bereitstellt und somit die doppelten ACK-Pakete des Empfängers auslösen würde.

Die Bestimmung ist auch hier eine Abwägungssache, lässt sich jedoch einfacher mathematisch begründen. Der im Folgenden beschriebene Algorithmus wurde aus [PA00] entnommen.

Als Grundlage des Algorithmus, wird zunächst die Round Trip Time (RTT) benötigt. Diese gibt an, wieviel Zeit vergeht, wenn ein Paket die Strecke vom Quell- zum Zielhost und wieder zurück hinter sich bringt. In RUDP-NG wird dies ermittelt, indem beim Senden die Pakete im Speicher einen Zeitstempel bekommen. Trifft die Empfangsbestätigung ein, so kann mittels Bildung der Differenz die RTT ermittelt werden. Da einzelne Messungen jedoch massiven Schwankungen unterworfen sind, wird aus den RTT Messungen eine geglättete Paketumlaufzeit berechnet (SRTT - Smoothed Round Trip Time). Zusätzlich wird die Schwankung der Messungen in RTTVAR (Round Trip Time Variation) festgehalten.

Initial beträgt RTO 3 Sekunden. Bei der ersten Messung der RTT werden die folgenden Zuweisungen durchgeführt.

1. $\text{SRTT} \leftarrow \text{RTT}$
2. $\text{RTTVAR} \leftarrow \frac{\text{RTT}}{2}$
3. $\text{RTO} \leftarrow \text{SRTT} + 4 \cdot \text{RTTVAR}$

Bei einer nachfolgenden Messung werden SRTT und RTTVAR jeweils mit einem Glättungsfaktor angepasst. Dieser ist $0,125$ für δ und $0,25$ für γ.

1. $\text{RTTVAR} \leftarrow (1 - \gamma) \cdot \text{RTTVAR} + \gamma \cdot |\text{SRTT} - \text{RTT}|$
2. $\text{SRTT} \leftarrow (1 - \delta) \cdot \text{SRTT} + \delta \cdot \text{RTT}$
3. $\text{RTO} \leftarrow \text{SRTT} + 4 \cdot \text{RTTVAR}$

Ergibt beispielsweise eine erste Messung RTT = 50ms, so gilt SRTT = 50ms, RTTVAR = 25ms und RTO = 150ms.

Wird in einer zweiten Messung eine Änderung auf RTT = 75ms festgestellt, so wird folgt RTTVAR = 25ms, SRTT = $53,125$ms und RTO = $153,125$ms.

Um auf sich ändernde Netzwerkbedingungen schnell reagieren zu können, führt RUDP-NG mindestens einmal pro RTT eine Messung durch. Dies ist nur eine minimale zusätzliche Last, da der dafür benötigte Systemaufruf zum Erzeugen eines

Zeitstempels sowieso für jedes ausgehende Paket gemacht werden muss. Dadurch ist es erst möglich, mit einem Timer festzustellen, ob ein Paket erneut übertragen werden muss, da seit seinem Verschicken RTO Zeiteinheiten vergangen sind.

3.7 Nagle Algorithmus

Schreibt die Anwendung öfters kleinste Datenmengen und schickt die Transportschicht diese sofort los, so erzeugt dies einen enormen Overhead. Stellt man sich eine Telnet Sitzung vor, in der der Benutzer einzelne ASCII Zeichen schreibt, welche von der Transportschicht einzeln zum Zielhost übertragen werden, so ist jedem dieser Bytes ein IP, UDP und RUDP-NG Header von insgesamt 45 Bytes angeheftet. Dies macht prozentual gesehen einen Overhead von 4400% aus.

Eine bessere Auslastung erreicht man, wenn die Transportschicht Zeichen zwischenspeichert und zeitverzögert in einem Paket losschickt. Verzögert man jedoch zulange, so wird sich dies in einer geringeren Interaktivität beim Benutzer bemerkbar machen. Für TCP wurde deshalb ein Algorithmus [Nag84] entwickelt (nach seinem Entwickler auch als Nagle-Algorithmus bezeichnet), der beide Extreme miteinander abwägt und für RUDP-NG übernommen wurde: Wenn die Anwendungsschicht Daten zum Senden bereitstellt, so werden diese nur geschickt, wenn

- bisher Daten mit einem Volumen von einer MTU zusammengekommen sind ODER
- zur Zeit keine unbestätigten Daten mehr vorliegen.

Damit sollen zum einen volle Pakete versendet werden oder maximal ein Paket mit Overhead unterwegs sein. Falls Applikationen auf eine geringe Latenz angewiesen sind, so lässt sich dieser Algorithmus abschalten (siehe Kapitel 4.1).

3.8 Verbindungsaufbau

RUDP-NG verwendet einen Drei-Wege-Handschlag, der logisch dem von TCP entspricht. Will A eine Verbindung zu B aufbauen, so wird ein RUDP-NG Paket geschickt, dessen einzige gesetzte Flag SYN ist. Als Sequenznummer wird eine zufällige Zahl SEQ_A gewählt.

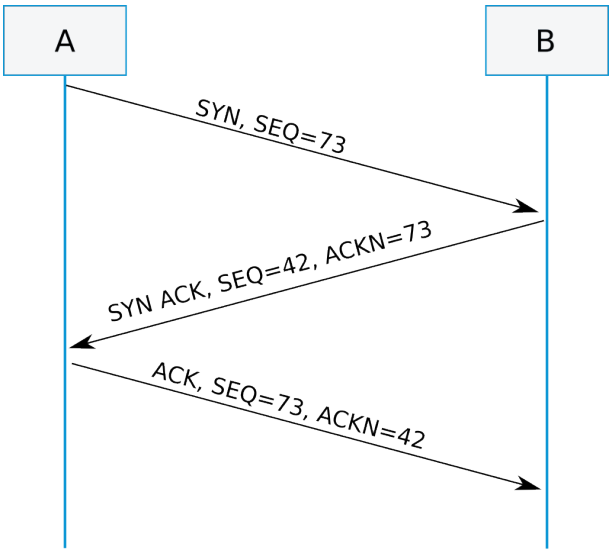

Abbildung 3.5: Verbindungsaufbau

B antwortet darauf mit einem Paket, bei dem sowohl das SYN als auch das ACK Flag gesetzt sind. Dessen Sequenznummer SEQ_B ist ebenfalls zufällig gewählt. Im ACK-Feld wird SEQ_A gespeichert.

A schließt den Handshake mit einem Paket mit gesetzten ACK Bit ab. Die Sequenznummer ist SEQ_A, als ACK-Wert wird SEQ_B verwendet.

Abbildung 3.5 stellt den Verbindungsaufbau schematisch dar.

Bleibt während des Handschlages eine Antwort aus, so wird der jeweils letzte Schritt 5 mal wiederholt. Dazwischen wird beim n-ten Versuch jeweils 2^{n-1} Sekunden auf eine Antwort gewartet.

SEQ_A und SEQ_B werden von beiden Hosts jeweils gespeichert, da dies jeweils der Index des ersten Nutzbytes sein wird.

3.9 Verbindungsabbau

Ziel eines korrekten Verbindungsabbaus ist es, dass sich beide Seiten darauf einigen, eine Verbindung zu schließen, ohne dass dabei Daten verloren gehen. Schließt beispielsweise A die Verbindung abrupt und B wollte noch ausstehende Daten schicken, so gehen diese verloren.

Aus diesem Grund müssen beide dem Verbindungsabbau zustimmen. Startet A den Verbindungsabbau, weil dies von der oberen Schicht gefordert wurde, so setzt er, sobald alle noch ausstehenden Daten verschickt wurden, in den folgenden Paketen jeweils das FIN-Flag. Die Gegenseite, B, sieht dies und wird, sobald auch hier alle Daten von der Gegenseite bestätigt wurden, ebenfalls das FIN-Flag setzen.

Empfängt eine Partei ein Paket mit FIN-Flag und hat zuvor selbst ein Paket mit FIN-Flag geschickt, so schickt sie noch ein FIN-Paket und schließt die Verbindung.

Da die FIN-Pakete verloren gehen könnten und nicht Teil der von RUDP-NG bereitgestellten Verlässlichkeit sind, hat man prinzipiell ein unlösbares Problem, was sich auf das "Coordinated Attack Probem" [HM84] abbilden lässt. Jede Seite kann sich niemals sicher sein, dass die Gegenseite die Bestätigung der Bestätigung erhalten hat, eine gewisse Unsicherheit bleibt immer.

Aus diesem Grunde wird mit Timeouts gearbeitet, welche im nächsten Abschnitt erläutert werden. Zusätzlich kommt das RST Flag zum Einsatz. Schickt eine Seite noch Pakete, obwohl die andere Seite die Verbindung schon geschlossen hat, so wird mit einem RST Flag geantwortet. Erhält eine RUDP-NG Verbindung solch ein Paket, wird die Verbindung ohne das Senden weiterer Pakete terminiert.

3.10 Keep Alive

In Kapitel 2.3 wurde die Motivation für RUDP-NG ausführlich beschrieben. Dort wurde dargelegt, dass der Vorteil bei UDP ist, dass die Einträge in den NAT Tabellen nur per Timer ausgetragen werden und nicht aufgrund von empfangenen RST-Paketen.

Um zu verhindern, dass solch ein Timer eine noch aktive Verbindung austrägt, werden von RUDP-NG regelmäßig Pakete geschickt. Da laut [FSK05] einige NAT-Implementierungen einen Timeout von 20 Sekunden haben, werden diese Pakete alle 10 Sekunden geschickt, wenn derzeit keine Nutzlast übertragen wird. Dies lässt sich auch gleichzeitig dafür nutzen, um eine nicht mehr antwortende Seite zu erkennen, wie dies bei einem Absturz der Fall wäre. Wurde seit mehr als 50 Sekunden kein Paket mehr empfangen, kann die Verbindung mit einer ausreichenden Wahrscheinlichkeit als inaktiv angesehen werden.

4 Implementierung

Da das peers@play Projekt in C# und auf dem .NET Framework implementiert ist, wurde dieses für einen nahtlosen Anschluss ebenfalls für RUDP-NG verwendet.

Zudem wurde Gears4Net [SWHW09] eingesetzt, welches ein Framework für ein neuartiges Programmiermodell ist. Dessen Vorteile und Grundlagen werden in Abschnitt 4.3 dargelegt.

4.1 Das ITransport Interface

Zur Abtrennung der Transportschicht wurde schon vor dieser Arbeit die ITransport Schnittstelle entworfen. Dieses Vorgehen erlaubt ein einfaches Auswechseln der Transportprotokolle und dient der Übersichtlichkeit. RUDP-NG implementiert jene Schnittstelle und bietet dadurch seine Dienste für obere Schichten an. Die oberen Schichten brauchen auch für mehrere offene Verbindungen maximal eine Instanz jeder Implementierung dieser Schnittstelle.

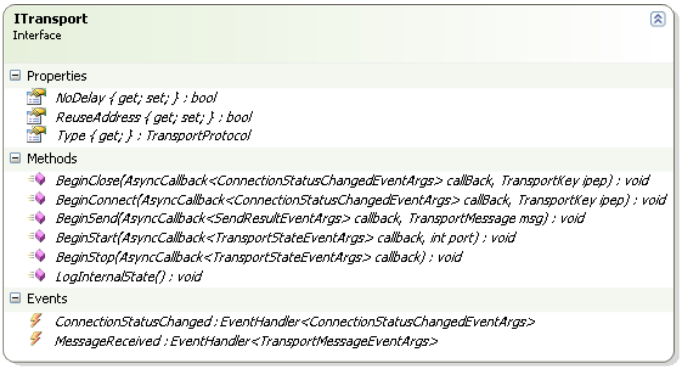

Abbildung 4.1: Das ITransport Interface

4.1.1 Eigenschaften

ITransport bietet zwei Möglichkeiten, um das interne Verhalten zu beeinflussen. Zum einen kann man mittels *ReuseAddress* für jede ausgehende Verbindung den gleichen lokalen Port verwenden, zum anderen lässt sich mit *NoDelay* der Nagle-Algorithmus abschalten. Dies führt zu einer kürzeren Antwortszeit auf Kosten eines geringeren Durchsatzes (siehe Abschnitt 3.7).

4.1.2 Methoden

Alle Methoden, welche logisch gesehen einen Rückgabewert haben, benutzen den als Parameter gegebenen Methodenzeiger, um das Ergebnis zu liefern. Sie selbst arbeiten asynchron, weshalb das Ergebnis nach Endes des Funktionsaufrufes noch nicht feststeht.

Eine Initialisierung findet durch das Aufrufen der *BeginStart* Methode statt. Über den Methodenzeiger wird mitgeteilt, ob das Starten erfolgreich war oder nicht. Fehlerursache könnte zum Beispiel sein, dass der angegebene Port, auf dem auf einkommende Verbindungen gelauscht werden soll, bereits belegt ist.

Nachdem RUDP-NG gestartet wurde, lässt sich mittels *BeginConnect* eine Verbindung zu einem entfernten Rechner aufbauen. Sollte der Verbindungsaufbau (siehe auch Abschnitt 3.8) nicht innerhalb des eingestellten Zeitfensters zustande kommen, so wird auch in diesem Fall der Callback aufgerufen.

Sobald eine Verbindung besteht, kann über diese mittels *BeginSend* Daten versendet werden. Der Callback wird dabei dann aufgerufen, wenn entweder die Gegenseite den Empfang der Daten bestätigt hat oder die Verbindung unerwartet beendet wurde. Dadurch wird das Konzept der Verlässlichkeit für die oberen Schichten umgesetzt.

Geschlossen werden Verbindungen durch *BeginClose*, welches einen Verbindungsabbau in Gang setzt (siehe Abschnitt 3.9). Durch *BeginStop* werden alle Ressourcen - insbesondere der Socket Handle - freigegeben.

4.1.3 Events

Vom Netzwerk ausgelöste Ereignisse werden per Event an die obere Schicht weitergegeben. Dazu gehört zum einen das Ändern des Status einer Verbindung (*ConnectionStatusChanged*) und das Empfangen von Nachrichten (*MessageReceived*). Dabei werden die Nachrichten nicht in der Reihenfolge durchgereicht, in der sie empfangen, sondern gesendet werden. Ein Event ist dabei der Aufruf einer Methode, die die obere Schicht zuvor angegeben hat.

Dadurch, dass die Transportschicht aktiv die Daten zum Bearbeiten abgibt und nicht passiv auf ein Lesen angewiesen ist, ist das Fehlen eines Receive Windows in den Paketen zu erklären.

4.2 Sequenznummern

Die in Formel 3.1 beschriebene Relation lässt wegen der in C# verfügbaren unsigned Typen effizient implementieren. Diese haben nämlich die Eigenschaft, dass bei Bildung einer Differenz mit einem größeren Subtrahenden als Minuenden es zu einem Überlauf kommt und das Ergebnis wieder positiv ist. Dies kann dazu genutzt werden, um bei Sequenznummern zu bestimmen, wie sie zueinander geordnet sind. Ist die Differenz größer als die Hälfte des maximalen Wertes, den dieser Datentyp annehmen kann, so liegt der Minuend vor dem Subtrahenden.

```
1  public class SequenceComparer : IComparer<ulong>
2  {
3      public int Compare(ulong x, ulong y)
4      {
5          ulong diff = y - x;
6          if (diff == 0)
7              return 0;
8          else if (diff < UInt64.MaxValue / 2)
9              return -1;
10         return 1;
11     }
12 }
```

Listing 4.1: Implementierung des IComparer Interfaces

Genutzt wird dies für einkommende Pakete einer Verbindung. Für diese wird das in .NET vorhandene *SortedDictionary* verwendet, welches anhand eines Schlüssels Objekte speichert. Hier wird als Schlüssel die Sequenznummer und als zu speicherndes Objekt das gesamte eingegangen Paket verwendet. Ein *SortedDictionary* ermöglicht es, das Kriterium, nach dem die Schlüssel geordnet werden, durch implementieren des *IComparer* Interfaces festzulegen. Dieses hat eine Methode, welche jeweils zwei Schlüssel vergleicht und diese als "gleich", "kleiner als" oder "größer als" bewertet. Der Rückgabewert dieser Methode ist ein Integer, welcher für die einzelnen Fälle jeweils 0, einen Wert kleiner als 0 oder einen Wert größer als 0 annimmt. Listing 4.1 zeigt die Implementierung des IComparer Interfaces, wie es für die Sequenznummern verwendet wird. Intern wird beim Einfügen in ein *SortedDictionary* in $O(log_2(n))$ mittels binärer Suche die Stelle bestimmt, an der das Objekt liegen soll und dort abgelegt.

Benötigt wird dies, um Pakete, die in der falschen Reihenfolge ankommen, effizient zu verwalten. Solche werden in das *SortedDictionary* eingefügt und immer nur so weit an die obere Schicht gegeben, wie es keine Lücken zwischen ihnen gibt. Da die Datenstruktur sortiert ist, kann immer anhand des ersten Objektes entschieden werden, ob dieses bereits der oberen Schicht übergeben werden kann. Ist dies der Fall, so wird es anschließend aus der Liste entfernt.

4.3 Gears4Net

Bei der Netzwerkprogrammierung gibt es zwei verschiedene Programmiermodelle. Auf der einen Seite kann man mit blockierenden Socketoperationen arbeiten, wobei beim Lesen aus einem Socket der aktuelle Thread solange angehalten wird, bis Daten zum Lesen bereitstehen. Dies hat den Vorteil, dass der Quelltext einfach zu verstehen ist. Listing 4.2, welches Teil eines Telnet Logins sein könnte, verdeutlicht dies.

```
1 String name = socket.readLine();
2 if(!isExistingName(name))
3     return;
4 String password = socket.readLine();
5 login(name, password);
```

Listing 4.2: Arbeiten mit blockierenden Socketoperationen

Der Nachteil dieses Ansatzes ist, dass pro Socket, aus dem gelesen werden soll, ein Thread zur Verfügung stehen muss. Jeder Thread beansprucht dabei durch seinen Stack einen Speicher, der, abhängig vom Betriebssystem, zwischen 256 und 2048 kb groß ist. Hat man demnach 1024 offene Verbindungen, so verbraucht dies im schlimmsten Fall bereits 2 GB des Arbeitsspeichers.

Die Alternative ist der Einsatz von asynchronen Socketoperationen. Bei diesen werden neue Daten mittels eines Callbacks der Anwendung zur Verfügung gestellt. Da dieser Callback jedoch kontextlos erfolgt, muss der aktuelle Status in einer Variable zwischengespeichert werden. Das ist zwar effizient, führt aber zu einem nicht einfach nachvollziehbaren Quelltext. Der dahinterstehende Protokollablauf ist nicht sofort ersichtlich. Listing 4.3 illustriert dies anhand des oben eingeführten Beispieles.

```
1  State state = INITIAL;
2
3  void OnReceiveData(String data) {
4      if(state == INITIAL) {
5          name = data;
6          if(!isExistingName(name))
7              return;
8          state = GOT_NAME;
9      } else if(state == GOT_NAME) {
10         password = data;
11         login(name, password);
12     }
13 }
```

Listing 4.3: Arbeiten mit blockierenden Socketoperationen

Gears4Net [SWHW09] vereint die Vorteile beider Ansätze, indem es das Iteratorkonzept aus C# verwendet. In diesem kann man über Objekte iterieren, die in einem inneren Codeblock gerade erst erzeugt werden (siehe Listing 4.4). Der innere Block hat dabei keinen eigenen Stack. Der Compiler transformiert diesen Teil in eine eigene Klasse, deren Methoden jeweils den Code zwischen den *yield* Schlüsselwörtern darstellen. Lokale Variablen werden zu Attributen dieser Klasse. Als öffentliche Methode steht *MoveNext* zur Verfügung, welche den nächsten Wert zurückgibt. Dazu wird in einer interne Variable wird gespeichert, welche Methode

zuletzt ausgeführt wurde. Wird der nächste Wert angefragt, so wird durch diese Variable die entsprechende nächste Methode bestimmt. [ECM01, Sek 26.2]

```
IEnumerator<int> GetPrimes() {
    int i=2;
    yield return i; i += 1;
    yield return i; i += 2;
    yield return i; i += 2;
    yield return i;
}

void Main() {
    foreach(int prime in GetPrimes())
        Console.WriteLine(prime);
}
```

Listing 4.4: Beispiel eines C# Iterator

Darauf aufbauend verfolgt Gears4Net den Ansatz, dass sämtliche Protokolle mit Hilfe dieser Iteratoren geschrieben werden. Als Rückgabewerte werden Wartebedingungen verwendet. Da jede erfüllte Wartebedingung zu einem neuen Zustand führt, werden die mit diesen Iteratoren geschriebenen Protokolle auch Zustandsmaschinen genannt. Ein Thread kann mehrere solcher Iterationen pseudo-parallel bearbeiten. Immer wenn eine Wartebedingung eines Iterators erfüllt ist, wird bei diesem *MoveNext* aufgerufen. Es ist also möglich, dass mit nur einem Thread mehrere Iteratoren auf verschiedene Bedingungen warten. Der Overhead, der im Vergleich mit rein asynchronen Operationen entsteht, ist dabei minimal. Das Telnet Beispiel ist in Gears4Net geschrieben (Listing 4.5) vom Ablauf so verständlich, wie bei Verwendung blockierender Sockets.

```
IEnumerator<ReceiverBase> HandleLogin(StateMachine st) {
    yield return Receive<Message<String>>;
    String name = st.CurrentMessage.Value;
    if(!isExistingName(name))
        yield break;
    yield return Receive<Message<String>>;
    String password = st.CurrentMessage.Value;
    login(name, password);
}
```

Listing 4.5: Gears4Net Beispiel

4.4 Architektur

Da Gears4Net verwendet wurde, sind die aktiven Komponenten in RUDP-NG nicht einzelne Threads sondern Zustandsmachinen.

Die *RUDP* Klasse, die das ITransport Interface implementiert, stellt selbst eine Zustandsmachine dar. Diese hat jedoch nur sehr beschränkte Aufgaben. Sie verwaltet den UDP-Socket indem sie UDP-Pakete aus diesem entgegen nimmt und an die einzelnen Verbindungen weiterreicht. Haben diese Pakete das SYN-Flag und die richtige RUDP-NG Identifikation (siehe Abschnitt 3.4), so werden neue *RUDP-Connection* Objekte angelegt. Das Gleiche passiert, wenn durch *BeginConnect* eine ausgehende Verbindung erzeugt werden soll.

Aufrufe von *BeginSend* und *BeginClose* werden ebenfalls in *RUDPConnection* gehandhabt.

Eine RUDPConnection beginnt zunächst mit einem Handshake. Dabei wird unterschieden zwischen einer ausgehenden und einer eingehenden Verbindung, da diese jeweils unterschiedliche Pakete senden (siehe Abschnitt 3.8). Ist der Verbindungsaufbau abgeschlossen, so werden mit *HandleChopping*, *HandleOutgoing* und *HandleIncoming* drei weitere Zustandsmachinen gestartet.

Mittels *BeginSend* übergebene Daten zum Verschicken werden in *HandleChopping* in Pakete umformatiert, deren Größe durch die MTU begrenzt ist. Diese werden anschließend in einer Warteschlange zum Übertragen eingefügt, wo sie von *HandleOutgoing* bearbeitet werden. Ist die *NoDelay* Eigenschaft deaktiviert, so wird der Nagle Algorithmus angewendet (siehe Abschnitt 3.7), und das Einreihen in die Wartschlange kann sich verzögern. Da es dabei passieren kann, dass die Warteschlange leer läuft und somit noch verzögerte Daten in diese eingespeist werden sollten, wartet *HandleChopping* alternativ auch auf ein Signal, welches andere Zustandsmachinen emittieren können. Durch dieses wird dann die leere Queue mit Paketen aufgefüllt, deren Größe unterhalb der MTU liegt.

HandleOutgoing ist für das Versenden von Paketen unter Berücksichtigung des Congestion Windows verantwortlich. Zudem werden Pakete mit Zeitstempeln versehen und nach Ablauf von RTO Sekunden erneut übertragen. Dazu verwendet es die Warteschlange, die *HandleChopping* zuvor mit Paketen gefüllt hat. Dabei wartet diese Zustandsmachine ebenfalls auf zwei alternative Bedingungen. Zum einen ist dies der Ablauf eines Timers zur erneuten Übertragung eines Paketes, zum anderen kommt wieder ein Signal zum Einsatz, durch den die Zustandsmachine explizit geweckt werden kann. Dieses wird dann ausgelöst, wenn etwas an der Queue verändert wurde. Dies tritt nicht nur ein, wenn *HandleChopping* dort neue Pakete einfügt, sondern auch, wenn welche entfernt werden.

Dies passiert dann, wenn *HandleIncoming* ein Paket erhalten hat, welches einen neuen ACK-Wert enthält. In diesem Fall werden alle Pakete aus der Queue entfernt, die dadurch implizit auch bestätigt wurden. *HandleOutgoing* muss in diesem Falle geweckt werden, da durch die Bestätigung wieder so viele neue Pakete geschickt werden können, wie das Congestion Window Platz bietet. Auf der anderen Seite werden von *HandleIncoming* auch eingegangene Pakete mit Nutzlast gehandhabt. Diese werden in das oben erwähnte *SortedDictionary* eingetragen und entsprechend ihrer Reihenfolge nach oben weitergegeben. Bestätigungen für diese werden ebenfalls hier verzögert (siehe Abschnitt 3.2.3) versendet. Zusätzlich kümmert diese Zustandsmachine sich sowohl um die eingehenden als auch die ausgehenden Keep-Alive Pakete. Bleiben die eingehenden Pakete für eine zu lange Zeit aus, wird die Verbindung terminiert.

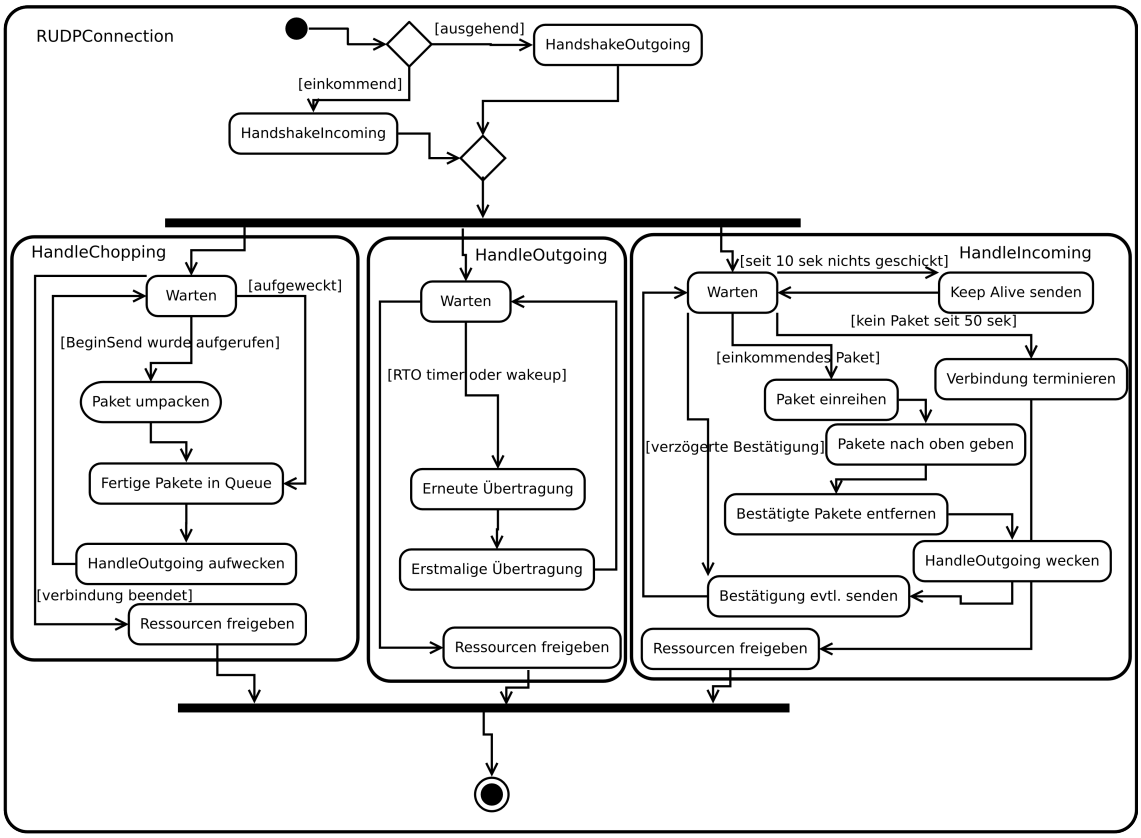

Abbildung 4.2: Aktivitätsdiagramm einer RUDPConnection

5 Auswertung

Im Anschluss an die Implementierung wurde RUDP-NG unter verschiedenen Netzwerkbedingungen getestet und mit einer TCP Implementierung verglichen, um einen Richtwert für die jeweiligen Kondition zu erhalten. Zudem wurden die Messungen ebenfalls mit der lidgren-network-gen3 Bibliothek durchgeführt, welche ein ebenfalls auf UDP aufbauendes verlässliches Transportprotokoll bietet.

5.1 Versuchsaufbau

Damit für alle Messungen reproduzierbare Bedingungen vorliegen, wurde eine geschlossene Versuchsumgebung geschaffen, die äußere Einflüsse ausschließt. Dazu wurde ein Rechner mit zwei Netzwerkkarten zwischen zwei PCs geschaltet. Auf dem mittleren Host wurde zur Emulation bestimmter Netzwerkbedingungen WANem [NKMR08] gestartet. Dieses erlaubt es, auf IP-Ebene Pakete zu verzögern, zu verlieren, zu duplizieren oder in geänderter Reihenfolge weiterzuleiten. Zusätzlich ist eine Limitierung der Bandbreite möglich.

Auf dem sendenden Rechner kommt Windows 7 mit Compound TCP zum Einsatz. Im Folgenden ist dies gemeint, wenn von TCP die Rede ist.

5.2 Versuchsprogramm

Zur Durchführung der Messreihen wurde ein Programm geschrieben, welches eine Implementierung der ITransport Schnittstelle verwendet, um eine Verbindung aufzubauen und anschließend mit einem Pseudozufallszahlengenerator erzeugte Daten überträgt. Der Initialisierungswert ist dabei aus Gründen der Reproduzierbarkeit

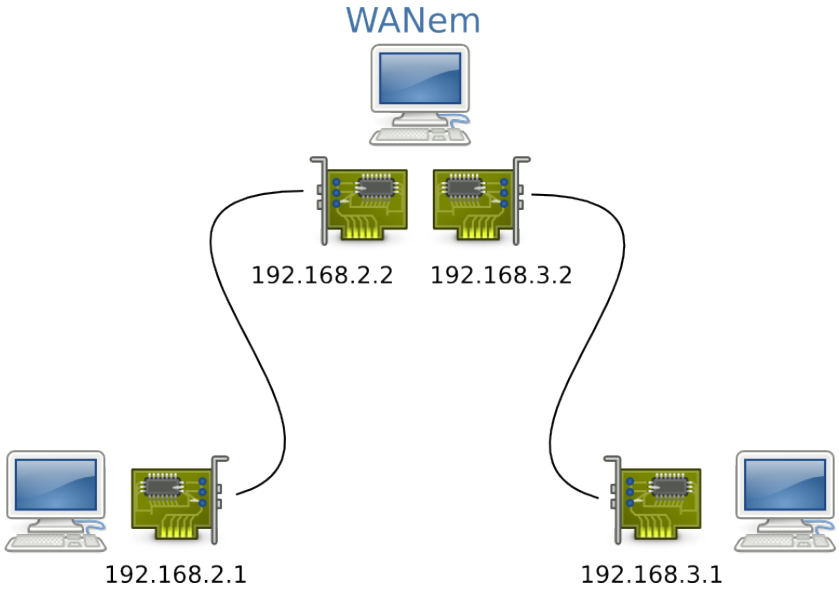

Abbildung 5.1: Versuchsaufbau

über alle Messungen konstant. Da beide Kommunikationspartner diesen Wert besitzen, kann der Empfänger die Daten verifizieren. So ist es möglich, eventuelle Übertragungsfehler zu erkennen.

Bei den einzelnen Protokollen musste somit nur jeweils die ITransport Implementierung ausgetauscht werden. Für lidgren-network-gen3 musste diese erst implementiert werden.

Zum Messen wird die *DateTime.Ticks* Eigenschaft des .NET Frameworks verwendet, welche eine Auflösung von 10^{-7} Sekunden hat[1].

5.3 Lidgren Network Gen3

Der Einsatz der Lidgren Bibliothek (Version vom 15.7.2010) gestaltete sich schwieriger als erwartet. Als einziges Protokoll führte hier ein Beschränken der Bandbreite zu einem Verbindungsabbruch, wobei die Beschränkung vor dem eigentlichen Versuch durchgeführt wurde. Erst durch Verwendung einer internen Option, durch welche die ausgehende Bytes pro Sekunde beschränkt werden können, waren Messungen überhaupt erst möglich. Damit Lidgren Network Gen3 (kurz LN3) nicht

[1]Siehe http://msdn.microsoft.com/de-de/library/system.datetime.ticks.aspx

einen unfairen Vorteil hat, wurde dieser Wert um 25% höher gewählt, als eigentlich Bandbreite zur Verfügung steht. Würde man den exakten Wert der Bandbreite angeben, wäre die Überlaststeuerung, welche eines der Hauptprobleme ist, trivial. Dass diese Bibliothek unter realen Bedingungen eingesetzt werden kann, bei denen auch der Benutzer nicht weiß, welche Bandbreite gerade zur Verfügung steht, darf deshalb bezweifelt werden.

5.4 Messungen

Es wurden jeweils $5 \cdot 10^6$ Bytes übertragen, wobei eine Bandbreitenbegrenzung von 1581 Kbps zum Einsatz kam. Dies entspricht einer T-1 DS1 Anbindung. Davon ausgehend wurden Messreihen gemacht, bei denen in WANem einzelne Parameter zur Simulation von Störungen verändert wurden.

Diese waren das Neuordnen von Paketen (Tabelle A.1), das Duplizieren von Pakete (Tabelle A.2) und das Verlieren von Paketen (Tabelle A.3). In einer letzten Reihe wurden alle 3 Parameter zusammen erhöht (Tabelle A.4). Die Parameter Verlust, Duplizierung und Neuordnen sind in Prozent angegeben, die Zeitangaben zur Verzögerung der Pakete und zur totalen Übertragungszeit in Millisekunden. Es wurden jeweils 5 Messungen durchgeführt, das arithmetische Mittel gebildet und sowohl Minumum als auch Maximum festgehalten.

5.4.1 Neuordnen der Pakete

Im Internet ist das Umsortieren von Paketen auf dem Weg vom Empfänger zum Ziel keine Seltenheit, da jedes IP-Paket prinzipiell eine andere Route nehmen kann. In einer in [Pax99] beschriebenen Beobachtung von TCP Verbindungen passierten 0.6% aller Pakete in einer falschen Reihenfolge den betrachteten Knotenpunkt.

In dieser ersten durchgeführten Messungen hatte das Umsortieren keine gravierenden Auswirkungen auf die Übertragungszeit von TCP oder RUDP-NG. Bei LN3 zeigte sich jedoch eine deutlich erhöhte Streuung der Messwerte bei einer Umordnungsrate von einem Prozent. Ein möglicher Erklärungsansatz wäre, dass die verfälschte Reihenfolge als Paketverlust fehlgedeutet wird und eine erneute Übertragung dieser Pakete stattfindet. Da dadurch Daten doppelt übertragen werden, verlängert sich die gesamte Übertragungszeit.

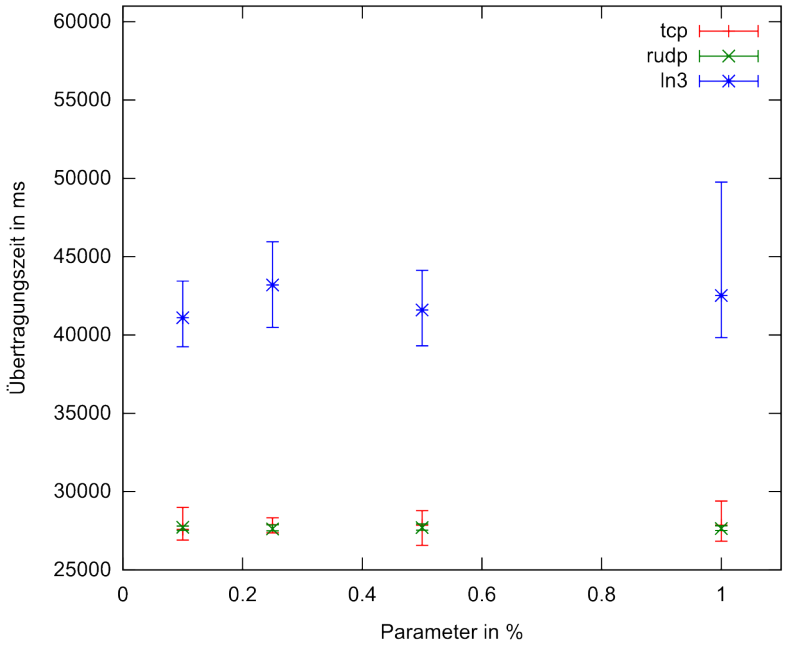

Abbildung 5.2: Neuordnen der Pakete

5.4.2 Paketeduplizierung

Duplizierte Pakete tauchen im Internet weitaus seltener auf als umgeordnete Pakete. In der oben referenzierten Studie wurden im Beobachtungszeitraum insgesamt nur 66 Fälle beobachtet. Anders als die Umsortierung, welche durch die Struktur des Internets zu erklären ist, kommt bei Duplizierung nur ein Fehler in Betracht. Dieser könnte etwa durch falsch konfigurierte Router oder Firewalls verursacht werden.

Auch bei dieser Messung gab es keine großen Abweichungen bei TCP oder RUDP-NG. LN3 hingegen zeigte erneut eine erhöhte Streuung. Denkbar wäre, dass dieses ebenfalls mehrfache Empfangsbestätigungen verwendet, um Paketverlust zur erkennen, aber einen niedrigeren Schwellwert hat. Dies würde wieder zu überflüssigen erneuten Übertragungen führen.

5.4.3 Paketverlust

Paketverlust, welcher nicht durch Überlastungen des Netzwerkes verursacht wird, tritt insbesondere bei kabellosen Technologien auf. Löst dieser eine Verminderung

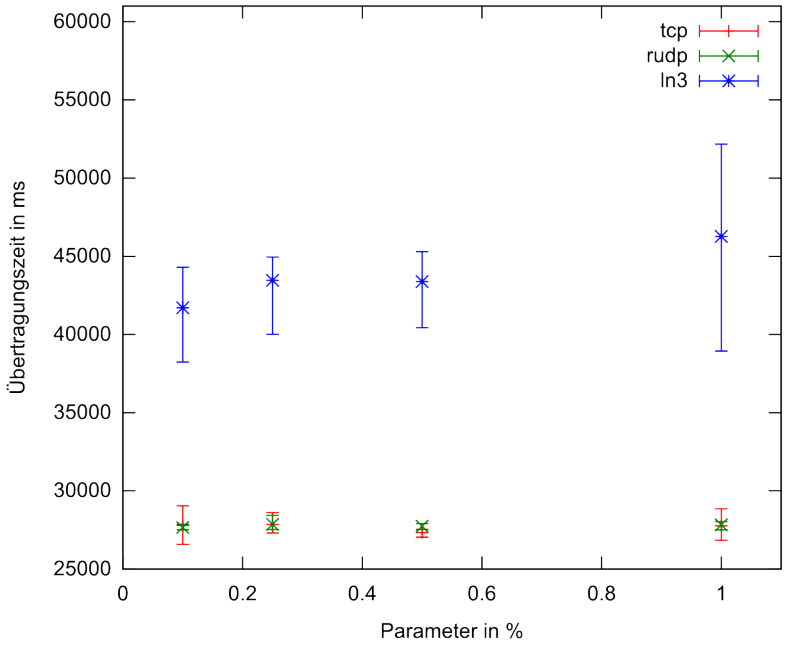

Abbildung 5.3: Duplizierung der Pakete

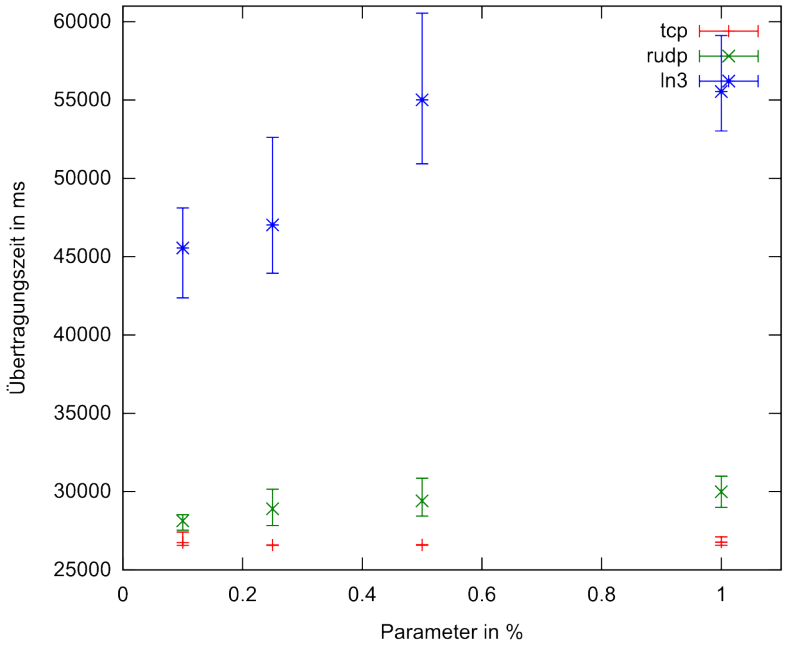

Abbildung 5.4: Verlust von Paketen

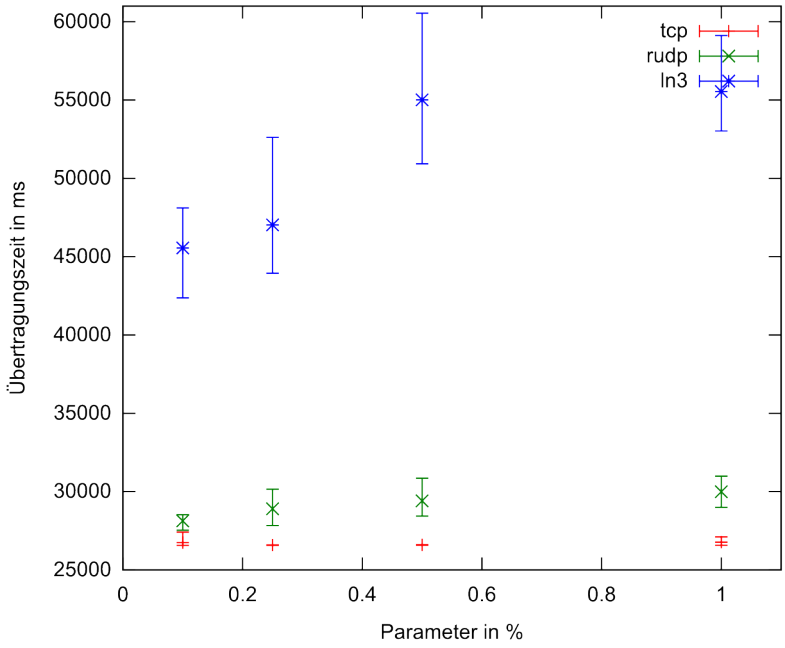

Abbildung 5.5: Messung mit gemeinsam erhöhten Parametern

der Senderate aus, so kann dies zu einem völligen Zusammenbruch der Verbindung führen.

In dieser Messreihe sieht man zum ersten Mal eine Differenz zwischen TCP und RUDP-NG, welche jedoch mit maximal 3 Sekunden bei einem Prozent Verlustrate noch im Bereich des Akzeptablen liegt. Interessant ist in diesem Zusammenhang der Unterschied der Übertragungszeit pro Protokoll zwischen den Messungen bei 0.1% und 1% Verlustrate. Während dies bei TCP 24 Millisekunden und bei RUDP-NG 1,85 Sekunden sind, hat LN3 mit 9,99 Sekunden hier den größten Wert.

5.4.4 Korrelierte Paramter

In einer letzten Messung wurden alle Werte gemeinsam erhöht, um zu überprüfen, ob sich gewisse Parameter zusammen wesentlich nachteiliger auf die Übertragungszeit auswirken.

Eine solche Auswirkung konnte jedoch bei keinem der verwendeten Protokolle nachgewiesen werden. Schaut man sich die Graphen der reinen Verlustreihe mit

dem der korrelierten Reihe, so ist es wahrscheinlich, dass die korrelierte Reihe hauptsächlich von der Verlustrate bestimmt wird.

6 Zusammenfassung und Ausblick

Mit RUDP-NG wurde ein verlässliches, auf UDP basierendes Protokoll entwickelt, welches im Rahmen des peers@play Projektes genutzt werden kann, um direkte Verbindungen zwischen Teilnehmern eines Peer-to-Peer Netzwerkes herzustellen, wo dies mit TCP nicht möglich wäre. Dabei verfügt es über Flusskontrolle, um das Netzwerk nicht zu überlasten. Durch die Implementierung der *ITransport* Schnittstelle kann es ebenso benutzt werden wie TCP. An der Programmlogik der oberen Schichten muss nichts geändert werden.

Durch Messungen wurde gezeigt, dass RUDP-NG auch unter widrigen Netzwerkbedingungen nur minimal mehr Zeit zum Übertragen von Daten braucht als TCP. Wenn man bedenkt, dass es vor Entwicklung dieses Frameworks unter der in Abschnitt 2.3.1 beschriebenen Netzwerkumgebung nur möglich war, mittels Routing im Peer-To-Peer Netzwerk Nachrichten auszutauschen, ist dieses eine deutliche Verbesserung.

In zukünftigen Arbeiten könnte die Flusskontrolle von RUDP-NG weiter verbessert werden. Insbesondere würde eine genauere Differenzierung zwischen medium- und staubedingtem Paketverlust unter bestimmten Netzwerkbedingungen zu einem besseren Durchsatz führen. Dies hätte das Potential, die erhöhte Übertragungszeit im Vergleich zu TCP bei Paketverlust (siehe Abschnitt 5.4.3) zu verbessern.

Dazu könnte man andere Indikatoren nehmen als den bloßen Paketverlust. Da bei einer Überlastung des Netzwerkes, bevor es zu einem Paketverlust kommt, sich Pakete in den Warteschlangen der Router stauen, führt dies zu einer wachsenden Latenz[BOP94]. Diese kann fortlaufend gemessen und dazu genutzt werden, um einen Stau zu erkennen. Tritt ein Paketverlust ein, obwohl sich die Latenz zuvor nicht erhöht hat, kann bei diesem auf eine Reduktion des Congestion Windows verzichtet werden.

Eine andere Möglichkeit wäre, die Paketverluste statistisch zu erfassen. Treten diese beispielsweise auch bei einem sehr kleinen Fenster auf, so sind diese sehr wahrscheinlich nicht staubedingt. Man könnte demnach in einer ersten Phase bei einem kleinen Fenster die mediumbedingte Paketverlustrate ermitteln. Darauf aufbauend könnte man während der eigentlichen Übertragung Paketverluste innerhalb dieses ermittelten Wertes ignorieren. Erst wenn die Paketverluste den Grenzwert überschreiten, wäre eine Verkleinerung des Fensters angebracht.

Anhang A

Tabellen der Messwerte

Prot.	Delay	Loss	Dup	Reord	AVG	min	max
tcp	10	0	0	0.1	27597.6835	26908.6928	28991.6880
rudp	10	0	0	0.1	27721.8620	27529.5856	27799.9744
ln3	10	0	0	0.1	41101.1004	39246.4336	43442.4672
tcp	10	0	0	0.25	27896.1126	27369.3552	28330.7376
rudp	10	0	0	0.25	27615.7094	27499.5424	27870.0752
ln3	10	0	0	0.25	43196.1129	40478.2048	45956.0816
tcp	10	0	0	0.50	27854.0521	26568.2032	28791.4000
rudp	10	0	0	0.50	27707.8419	27539.6000	27940.1760
ln3	10	0	0	0.50	41595.8118	39306.5200	44123.4464
tcp	10	0	0	1.0	27854.0521	26838.5920	29402.2784
rudp	10	0	0	1.0	27641.7468	27519.5712	27820.0032
ln3	10	0	0	1.0	42517.1366	39837.2832	49761.5536

Tabelle A.1: Messergebnisse, neuordnen der Pakete

Prot.	Delay	Loss	Dup	Reord	AVG	min	max
tcp	10	0	0.1	0	27785.9542	26588.2320	29041.7600
rudp	10	0	0.1	0	27661.7756	27509.5568	27870.0752
ln3	10	0	0.1	0	41703.9673	38234.9792	44293.6912
tcp	10	0	0.25	0	27852.0492	27309.2688	28611.1408
rudp	10	0	0.25	0	27856.0550	27529.5856	28430.8816
ln3	10	0	0.25	0	43458.4902	40007.5280	44954.6416
tcp	10	0	0.50	0	27337.3091	27038.8800	27539.6000
rudp	10	0	0.50	0	27741.8908	27529.5856	27910.1328
ln3	10	0	0.50	0	43378.3750	40428.1328	45295.1312
tcp	10	0	1.0	0	27769.9312	26848.6064	28851.4864
rudp	10	0	1.0	0	27828.0147	27509.5568	28030.3056
ln3	10	0	1.0	0	46266.5280	38935.9872	52175.0240

Tabelle A.2: Messergebnisse, Paketduplizierung

Prot.	Delay	Loss	Dup	Reord	AVG	min	max
tcp	10	0.1	0	0	26750.4652	26568.2032	27409.4128
rudp	10	0.1	0	0	28134.4553	27539.6000	28531.0256
ln3	10	0.1	0	0	45553.5027	42370.9264	48109.1776
tcp	10	0.25	0	0	26586.2291	26568.2032	26598.2464
rudp	10	0.25	0	0	28899.5555	27830.0176	30153.3584
ln3	10	0.25	0	0	47029.6252	43943.1872	52625.6720
tcp	10	0.50	0	0	26594.2406	26578.2176	26618.2752
rudp	10	0.50	0	0	29410.2899	28440.8960	30854.3664
ln3	10	0.50	0	0	55011.1020	50933.2384	60547.0624
tcp	10	1.0	0	0	26774.4998	26578.2176	27108.9808
rudp	10	1.0	0	0	29991.1251	28991.6880	30984.5536
ln3	10	1.0	0	0	55545.8710	53026.2480	59115.0032

Tabelle A.3: Messergebnisse, Paketverlust

Prot.	Delay	Loss	Dup	Reord	AVG	min	max
tcp	10	0.1	0.1	0.1	26630.2924	26588.2320	26668.3472
rudp	10	0.1	0.1	0.1	28410.8528	28200.5504	28751.3424
ln3	10	0.1	0.1	0.1	44562.0771	39176.3328	49591.3088
tcp	10	0.25	0.25	0.25	26680.3644	26648.3184	26718.4192
rudp	10	0.25	0.25	0.25	29151.9184	28300.6944	30283.5456
ln3	10	0.25	0.25	0.25	47852.8089	44503.9936	53617.0976
tcp	10	0.50	0.50	0.50	26832.5833	26668.3472	27249.1824
rudp	10	0.50	0.50	0.50	28815.4345	28410.8528	29021.7312
ln3	10	0.50	0.50	0.50	50542.6768	45675.6784	53136.4064
tcp	10	1.0	1.0	1.0	27014.8454	26878.6496	27409.4128
rudp	10	1.0	1.0	1.0	30037.1913	29312.1488	30754.2224
ln3	10	1.0	1.0	1.0	53905.5123	51283.7424	57102.1088

Tabelle A.4: Messergebnisse, gemeinsame Erhöhung der Parameter

Anhang B

Quelltexte

```
1  void Parse(byte[] udpPacket)
2  {
3      int i=0;
4      byte flags = udpPacket[i++];
5
6      if(flags == (byte)FirstByteflags.SYN)
7      {
8          i+= MAGIC_SYN_TOKEN.Length;
9          MaximumWindow = readULong(udpPacket, ref i);
10     }
11
12     SeqNum = readULong(udpPacket, ref i);
13
14     if ((flags & (byte)FirstByteflags.ACK) != 0)
15         AckNum = readULong(udpPacket, ref i);
16
17     Data = new byte[udpPacket.Length - i];
18     Array.Copy(udpPacket, i, Data, 0, Data.Length);
19 }
```

Listing B.1: Parsen eines RUDP Paketes

Literaturverzeichnis

[AJ07] Audet, F. und C. Jennings: *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*. RFC 4787 (Best Current Practice), Januar 2007.

[APB09] Allman, M., V. Paxson und E. Blanton: *TCP Congestion Control*. RFC 5681 (Draft Standard), September 2009.

[ARFK10] Abdeljaouad, I., H. Rachidi, S. Fernandes und A. Karmouch: *Performance analysis of modern TCP variants: A comparison of Cubic, Compound and New Reno*. In: *Communications (QBSC), 2010 25th Biennial Symposium on*, Seiten 80 –83, 12-14 2010.

[BOP94] Brakmo, Lawrence S., Sean W. O'Malley und Larry L. Peterson: *TCP Vegas: New Techniques for Congestion Detection and Avoidance*. Seiten 24–35, 1994.

[Bra89] Braden, R.: *Requirements for Internet Hosts - Communication Layers*. RFC 1122 (Standard), Oktober 1989. Updated by RFCs 1349, 4379, 5884.

[CDS74] Cerf, V., Y. Dalal und C. Sunshine: *Specification of Internet Transmission Control Program*. RFC 675, Dezember 1974.

[ECM01] ECMA: *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Dezember 2001.

[FSK05] Ford, Bryan, Pyda Srisuresh und Dan Kegel: *Peer-to-peer communication across network address translators*. In: *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, Seiten 13–13, Berkeley, CA, USA, 2005. USENIX Association. http://www.brynosaurus.com/pub/net/p2pnat/.

[HM84] Halpern, Joseph Y. und Yoram Moses: *Knowledge and common knowledge in a distributed environment.* In: *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, Seiten 50–61, New York, NY, USA, 1984. ACM.

[HRX08] Ha, Sangtae, Injong Rhee und Lisong Xu: *CUBIC: a new TCP-friendly high-speed TCP variant.* SIGOPS Oper. Syst. Rev., 42(5):64–74, 2008.

[JBB92] Jacobson, V., R. Braden und D. Borman: *TCP Extensions for High Performance.* RFC 1323 (Proposed Standard), Mai 1992.

[Nag84] Nagle, J.: *Congestion Control in IP/TCP Internetworks.* RFC 896, Januar 1984.

[NKMR08] Nambiar, Manoj, Hemanta Kumar Kalita, Debadatta Mishra und Shirish Rane: *WANem - The Wide Area Network emulator*, August 2008. http://wanem.sf.net/.

[PA00] Paxson, V. und M. Allman: *Computing TCP's Retransmission Timer.* RFC 2988 (Proposed Standard), November 2000.

[Pax99] Paxson, V.: *End-to-end Internet packet dynamics.* Networking, IEEE/ACM Transactions on, 7(3):277 –292, jun 1999.

[Pos80] Postel, J.: *User Datagram Protocol.* RFC 768 (Standard), August 1980.

[Pos81a] Postel, J.: *Internet Protocol.* RFC 791 (Standard), September 1981. Updated by RFC 1349.

[Pos81b] Postel, J.: *Transmission Control Protocol.* RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[RMK+96] Rekhter, Y., B. Moskowitz, D. Karrenberg, G. J. de Groot und E. Lear: *Address Allocation for Private Internets.* RFC 1918 (Best Current Practice), Februar 1996.

[RMMW08] Rosenberg, J., R. Mahy, P. Matthews und D. Wing: *Session Traversal Utilities for NAT (STUN).* RFC 5389 (Proposed Standard), Oktober 2008.

[SWHW09] Saternus, Martin, Torben Weis, Sebastian Holzapfel und Arno Wacker: *Gears4Net.* Parallel Processing Workshops, International Conference on, 0:113–120, 2009.

[Tan02] Tanenbaum, Andrew S.: *Computer Networks (4th Edition)*. Prentice Hall PTR, 4 Auflage, August 2002.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Duisburg, 29. September 2010 _____

Unterschrift