OST
Ostschweizer
Fachhochschule

P2P Library in Golang

Studien- und Bachelorarbeit

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Frühjahrssemester 2021

Autor(en):	Dominik Dietler, Fabienne König, Sinthujan Lohanathan
Betreuer:	Dr. Thomas Bocek
Experte:	Dr. Guilherme Sperb Machado
Gegenleser:	Prof. Frank Koch

Abstract

TomP2P ist eine Peer to Peer (P2P) Library, die eine Distributed Hash Table (DHT) für verteilte Anwendungen bereitstellt. Die Realisierung der Library in der Programmiersprache Java und die Verwendung des Internetprotokolls TCP verlangsamen die Applikation. Im Gegensatz dazu, bietet das neue QUIC-Protokoll mit seinem intelligenten Congestion Control Algorithmus eine performantere und auf die heutigen Anforderungen zugeschnittene Alternative. Sie ist allerdings wegen ihres breiten Anwendungsspektrums sehr komplex. Entwickelt werden soll deshalb ein auf UDP aufbauendes Protokoll, das die Zuverlässigkeit von TCP sicherstellt und zur Congestion Control den «Bottleneck Bandwidth and Round-trip propagation time» (BBR) Algorithmus von QUIC verwendet. Für die sichere Übertragung der Daten von P2P soll das Noise-Framework zum Einsatz kommen.

Zu Beginn wurde basierend auf dem UDP-Protokoll ein Automatic Repeat reQuest (ARQ) Protokoll in der Entwicklungsumgebung Goland realisiert. Parallel dazu wurde eine Testumgebung aufgebaut, welche die Implementation integral auf die Standard- und Spezialfälle hin testet. Die Continuous Integration (CI) erfolgte auf einer virtuellen Maschine der OST mit Gitlab CI. Im späteren Verlauf wurde die für die BBR unverzichtbare Delivery Rate Estimation entwickelt. Die Korrekheit wurde anschliessend auf dem Internet mithilfe von GCloud getestet und interaktiv verbessert. Gegen Ende wurde die Security mit dem Noise-Framework realisiert.

Die entwickelte Lösung erlaubt eine sichere, performante und zuverlässige Kommunikation zwischen mehreren Peers.

Management Summary

Ausgangslage

In dieser Bachelorarbeit soll die bereits bestehende TomP2P Library um ein geeignetes Protokoll erweitert werden. Bei TomP2P handelt es sich um eine P2P-Library, die eine Distributed Hash Table (DHT) mittels einer dezentralisierten Key-Value-Infrastruktur für verteilte Anwendungen bereitstellt. [1]

Da TomP2P mit der Nutzung der beiden Standard Protokollen TCP und UDP an seine Grenzen kommt, soll ein eigenständiges, minimales Protokoll implementiert werden. Dieses Protokoll soll zwar spezifisch auf die Anforderungen von TomP2P zugeschnitten sein, es soll allerdings bei Bedarf auch von anderen Anwendungen verwendet werden können.

Die jeweiligen Konzepte können aus dem Ursprungsprojekt TomP2P übernommen werden. Zudem gibt es bereits eine ähnliche Automatic Repeat Request (ARQ) Implementation namens ATP-go. Bei diesem Projekt handelt es sich um eine ehemalige Bachelor- und Studienarbeit, die für unsere Implementation teilweise als Basis dienen kann.

Vorgehen und Technologie

Für die Implementation wurde wegen seiner performanten Laufzeit Golang verwendet.

In einem ersten Schritt wurde ein auf dem UDP-Protokoll aufbauendes Automatic Repeat Request (ARQ) Protokoll implementiert, mit welcher die Reliability garantiert werden konnte. Danach galt es den für die «Bottleneck Bandwidth and Round-trip propagation time» (BBR) Algorithmus wichtige «Delivery Rate Estimation» zu integrieren, damit mittels Flusskontrolle eine Überlastung während der Übertragung verhindert werden kann.

Anschliessend wurde ein für das Projekt passendes Verschlüsselungskonzept gesucht und implementiert. Hierbei wurde auf das Noise Framework zurückgegriffen. Für die Ver- und Entschlüsselung wird auf ChaCha20-Poly1305 gesetzt, für die Hash-Funktion wurde Blake2b gewählt und als Handshake wird Curve25519 eingesetzt. Die implementierten Features wurden jeweils durch passende Unit- und Integrationstests abgedeckt.

Ergebnisse und Ausblick

Die entwickelte Lösung erlaubt eine sichere und zuverlässige Kommunikation zwischen mehreren Netzwerkendknoten, den sogenannten Peers. Zudem wurden zwei Ansätze zur genauen Messung der Sendegeschwindigkeit mit ihren Vor- und Nachteilen ausgearbeitet, die in einer Folgearbeit als Grundlage zur Realisierung des BBR-Algorithmus verwendet werden können.

Die Ergebnisse zeigen, dass die Implementierung von Netzwerkprotokollen im Allgemeinen kaum mit einfachen Unit-Tests realisiert werden können, sondern ganzheitliche Integration-Tests benötigen. Zudem konnten die Hindernisse und Schwierigkeiten beim «Delivery Rate Estimation» Algorithmus aufgezeigt werden.

Inhaltsverzeichnis

Einführung

1. Aufgabenstellung

In dieser Bachelor- und Studienarbeit geht es um die Erneuerung der TomP2P Library.
Bei TomP2P handelt es sich um eine P2P-Library, die eine Distributed Hash Table (DHT) mittels einer dezentralisierten Key-Value-Infrastruktur für verteilte Anwendungen bereitstellt. Hierbei verfügt jeder Peer über eine Tabelle, um seine Werte zu speichern. Auf diese Weise werden in TomP2P Key-Value-Paare auf verteilten Systemen abgespeichert. [1]

Da TomP2P mit der Nutzung der beiden Standardprotokolle TCP und UDP an seine Grenzen kommt, soll ein eigenständiges, minimales Protokoll implementiert werden. Dieses Protokoll soll zwar spezifisch auf die Anforderungen von TomP2P zugeschnitten sein, es soll allerdings bei Bedarf auch von anderen Anwendungen verwendet werden können.

Bei der zu implementierenden Library handelt es sich um ein UDP basiertes Transportprotokoll, das aufgrund der Laufzeitoptimierung mittels Verwendung von Nebenläufigkeit in Golang implementiert werden soll. Da UDP im Gegensatz zu TCP jedoch keine zuverlässige Verbindung bietet, müssen folgende Features eigenständig implementiert werden.

1.1. Zu implementierende Features

In einem ersten Schritt soll mittels Automatic Repeat Request (ARQ) die Zuverlässigkeit für die TomP2P Library implementiert werden. Dieser error-control-Mechanismus gewährleistet als allererstes eine zuverlässige Datenübertragung durch Sendewiederholungen.

Danach gilt es den «Bottleneck Bandwidth and Round-trip propagation time» (BBR) Algorithmus in das Protokoll zu integrieren. Damit soll während der Übertragung eine Überlastung mittels Flusskontrolle auf der Senderseite verhindert werden. Der von Google entwickelte Algorithmus optimiert somit hauptsächlich die Sendegeschwindigkeit des Datenverkehrs.

Anschliessend ist ein für das Projekt passendes Verschlüsselungskonzept zu suchen und zu implementieren. Hierbei kann beispielsweise auf das Noise Framework zurückgegriffen werden. Die Security soll dabei die wichtigsten Aspekte abdecken. Alle implementierten Features sollen jeweils durch passende Unit- und Integrationstests abgedeckt werden.

Denkbare optionale Features für diese Arbeit sind z.B. das Erstellen eines kurzen Benchmarks für die entwickelte Lösung. Dabei soll sie mit bekannten Protokollen (z.B. TCP, QUIC, etc.) verglichen werden. Weiter könnte eine Portierung des in dieser Arbeit erarbeiteten Protokolls in eine andere Hochsprache (bspw. C#) durchgeführt werden. Zudem wäre es möglich, eine minimale Version des RPC Konzepts der bestehenden TomP2P von Java in die neue Library in Golang zu implementieren, um im Anschluss RPC Verbindungen mit der neuen Library zu testen.

1.1.1. Ziele

Das Ziel der Arbeit ist es, ein ARQ Protokoll in Golang zu entwickeln. Dabei sollen die folgenden aus der Aufgabenstellung entnommenen Teilziele erreicht werden:

- Implementation eines ARQ Protokolls. Wichtig ist auch, dass Reflection Angriffe verhindert werden
- BBR congestion control
- Security Implementation, z.B. mit dem Noise Framework. Optional kann man einen Mechanismus auf Basis DNS implementieren für einen Schlüsselaustausch.
- Das Testen der einzelnen Implementationen mit entsprechenden Testcases
- Eine High-level Socket Schnittstelle, analog zu TCP
- Test und Benchmarks und optional Benchmarks im Internet

2. Technologien und Vorgehen

2.1. Golang

Golang wurde aus der Idee heraus geboren, eine optimierte und vereinfachte Programmiersprache für die parallele Programmierung zu werden. Es sollte eine Programmiersprache entstehen, die neben effizienter Code-Kompilierung und schneller Code-Ausführung auch einen einfachen Programmierungsprozess aufweist. Die erste stabile Version der drei für Google tätigen Entwickler Robert Giesemer, Rob Pike und Ken Thompson wurde 2012 als Open-Source-Projekt veröffentlicht. [2]

Neben einem ausdrucksstarken, aber leichtgewichtigen Typsystem und der Plattformunabhängigkeit, ist die Nebenläufigkeit ein wichtiger Aspekt von Golang. Sie ist direkt in Golang integriert und führt durch das einfache Starten und Synchronisieren von nebenläufigen Prozessen zu einer schnelleren Programmausführung. Die Nebenläufigkeit war unter anderem einer der Gründe, weshalb für diese Arbeit auf Golang gesetzt wurde.

Die Go-Syntax ist zwar an die Syntax von C angelegt, es gibt aber einige Optimierungen und Einflüsse aus anderen Sprachen. Auf einige sprachliche Besonderheiten, unter anderem in Bezug auf die Nebenläufigkeit, wird im Evaluation: Delivery Rate Estimation genauer eingegangen. [3]

2.2. CI

Die gitlab-CI wurde basierend auf dem Standard Go-Template [4] eingerichtet. Das Template wurde gemäss der «Gitlab CI for Go projects» [5] angepasst und mit Test-Coverage Analyse erweitert. Ein zweiter Job zur Erstellung von Unit-Test-Reports [6] vervollständigt die gitlab-CI.

3. Related Work

3.1. ATP-go

Bei ATP-go handelt es sich um eine frühere Bachelor- und Studienarbeit. Damals wurde ein 0-RTT Netzwerkprotokoll mit Reliability Features und einer built-in asymmetrischen Verschlüsselung implementiert. Somit ist es ähnlich wie TCP, versucht jedoch einige seiner Nachteile auszubügeln. Es eignet sich vor allem für Anwendungen mit Peer-to-Peer Übertragung.

Aufgrund der Ähnlichkeiten von ATP-go und dem in dieser Arbeit implementierten P2P-Library konnte vor allem am Anfang des Projekts auf die bereits implementierten Funktionen teilweise zurückgegriffen werden. Diese wurden laufend erweitert und verbessert. [7]

3.2. KCP-go

Bei KCP-go handelt es sich um ein weiteres ARQ Protokoll. Im Gegensatz zu ATP-go verfügt es über sehr viele ineinander verflochtene Features. Um ein simples und minimales Protokoll zu implementieren, wurde ATP-go anstelle von KCP-go als Grundlage für diese Arbeit gewählt. [8]

3.3. QUIC

QUIC hat im Gegensatz zu den zuvor vorgestellten Protokollen wohl am wenigsten mit der in dieser Arbeit entwickelte Golang Implementierung gemeinsam. Am Anfang des Projekts war es trotzdem wichtig, sich in das QUIC Protokoll einzuarbeiten.

Bei QUIC handelt es sich um ein experimentelles und heute weit verbreitetes Netzwerkprotokoll von Google, das mit dem Ziel, den Internetverkehr insgesamt zu beschleunigen, entwickelt wurde. Damit hat es gewisse Ähnlichkeiten mit der Zielsetzung des in dieser Arbeit implementierten Protokolls.

Gegen Ende der Bachelor- und Studienarbeit Ende Mai 2021 wurde QUIC im RFC 9000 standardisiert und ist dadurch neben TCP und UDP nun offizieller Internet-Standard. Obwohl HTTP anfangs stark an QUIC gekoppelt war, werden die beiden demnächst offiziell durch die IETF getrennt, wodurch HTTP über QUIC zu HTTP/3 wird. [9]

Theorie und Implementation

Um der Arbeit einen roten Faden zu verleihen, wird in diesem Kapitel pro Schwerpunkt jeweils erst ausführlich auf die Theorie eingegangen. Anschliessend werden die umgesetzten Implementation beschrieben. Falls ein Kapitel keinen Theorieteil beinhaltet, wird direkt die Umsetzung beschrieben.

4. Automatic Repeat Request (ARQ)

Der Automatic Repeat Request ist ein error-control-Mechanismus, der eine zuverlässige Datenübertragung über unzuverlässige Verbindungen ermöglicht. Diese wird durch Sendewiederholungen, wie es auch bei TCP gemacht wird, gewährleistet.

Wenn das Paket beim Empfänger ankommt, sendet der Empfänger dem ursprünglichen Sender ein «Acknowledgement» (ACK) als Antwort zurück. Dadurch weiss der Sender, dass das Paket erfolgreich beim Empfänger angekommen ist.

Bei manchen Implementationen sendet der Empfänger auch eine Art negative Bestätigung in Form eines «Negative Acknowledgements» (NACK). Dadurch wird der Fehler schneller kommuniziert und eine erneute Übertragung des Pakets wird zeitnaher ausgeführt. Der Trade-off ist jedoch, dass das Netz dadurch stärker belastet wird. Daher wurde in unserer Implementation komplett auf NACKs verzichtet. Stattdessen wartet der Sender einfach ab, ob ein ACK bei ihm ankommt. Sollte in einem bestimmten Zeitintervall kein ACK ankommen, so sendet er sein Paket erneut. Durch dieses Retransmission Timeout (RTO) kann auf die Implementation von NACKs verzichtet werden. Der Sender sendet sein Paket so lange, bis er ein ACK vom Empfänger erhält, oder bis er eine vordefinierte Anzahl von erneuten Übertragungen überschreitet. [10]

Mit dieser Übertragungsweise ergeben sich drei Probleme, auf die im Folgenden eingegangen wird.

Lost Packet Scenario

Beim Lost Packet Scenario besteht das Problem bereits beim Versenden des Pakets. Es geht auf dem Weg zum Empfänger verloren. Der Sender wartet somit auf ein ACK, das nie abgeschickt wurde. Nach Ablauf des RTO sendet der Sender das Paket erneut.

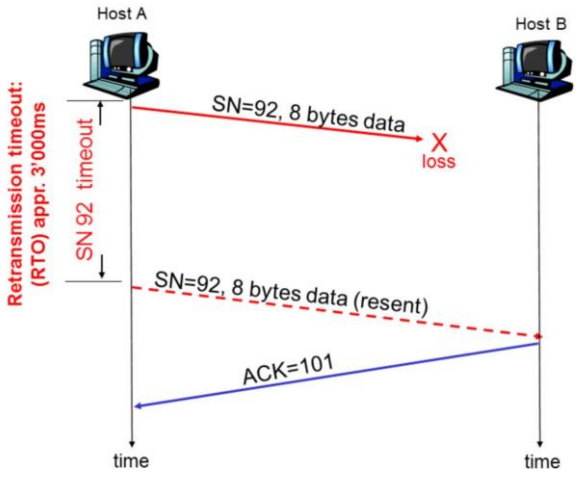

Abbildung 1 Lost Packet Scenario [11]

Lost Acknowledgement Scenario

Beim Lost Acknowledgement Scenario kommt das vom Sender gesendete Paket zwar beim Empfänger an, das vom Empfänger gesendete ACK geht aber auf dem Rückweg verloren. Auch hier wartet der Sender das RTO ab und sendet das Paket anschliessend einfach erneut.

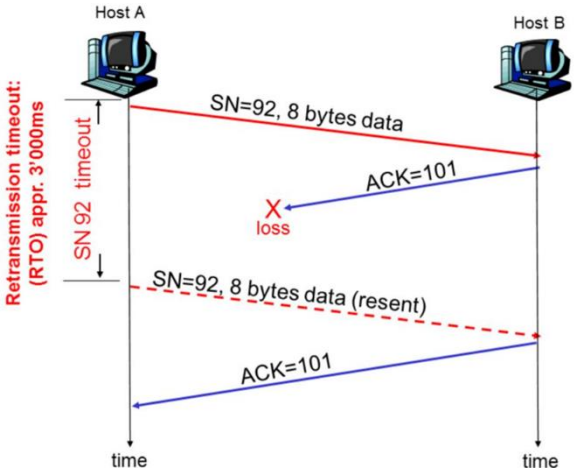

Abbildung 2 Lost Acknowledgement Scenario [11]

Negative Acknowledgement (NACK)

Ein dritter Fehlerfall tritt auf, wenn ein Paket fehlerhaft beim Empfänger ankommt. Statt eines ACKs kann er optional ein NACK (Negative Acknowledgement) zurück an den Sender schicken. Anderenfalls wartet er einfach bis der RTO abgelaufen ist und der Sender das Paket erneut gesendet hat. In der Implementation der P2P-Library dieser Arbeit wurde, wie bereits erwähnt, auf das NACK verzichtet.

4.1. Stop-and-Wait ARQ

Die Version «Stop-and-Wait» ist das simpelste ARQ Verfahren. Der Sender muss nach dem Senden eines Pakets auf das dazugehörige ACK warten, bis er sein nächstes Paket senden kann. Kommt innerhalb des RTO kein ACK beim Sender an, so sendet er das ursprüngliche Paket erneut. Dies macht das «Stop-and-Wait ARQ» relativ langsam und ineffizient.

Ein bekanntes Beispiel hierfür ist TFTP (Trivial File Transfer Protocol).

Wie im untenstehenden Diagramm ersichtlich ist, sendet der Empfänger auf jedes erhaltene Paket ein ACK als Antwort.

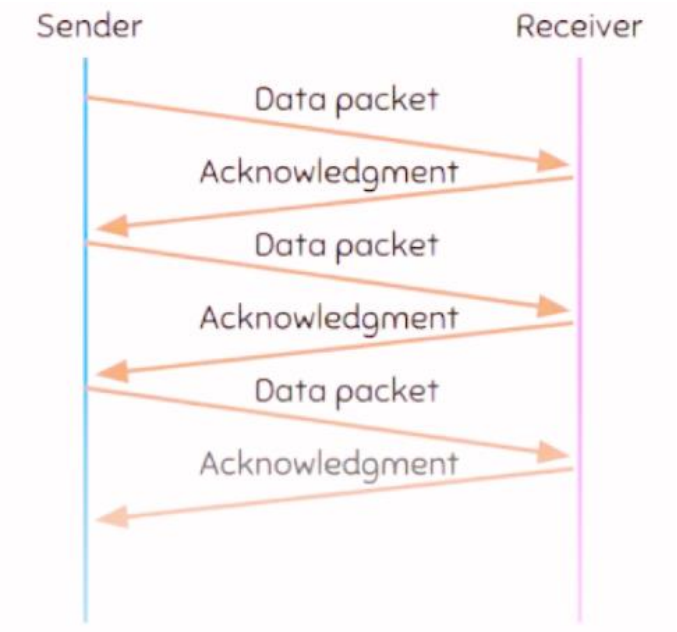

Abbildung 3: Stop and Wait ARQ Protokoll Diagramm [12]

4.2. Go-Back-N ARQ

Der Hauptunterschied zu «Stop-and-Wait ARQ» besteht darin, dass der Empfänger hier einen Buffer besitzt, in dem er die ankommenden Pakete zwischenspeichern kann.

Go-Back-N ARQ ermöglicht es dem Sender, mehrere Pakete zu senden, bevor er auf ein ACK des Empfängers warten muss. Die Anzahl möglicher Pakete wird über die Window Size des sogenannte Sliding-Windows festgelegt. Ist die Window Size ***n***, können ***n-1*** Pakete ohne Quittung gesendet werden. Das ***nte*** Paket muss dann wieder mit einem ACK bestätigt werden. Dadurch wird ein deutlich grösserer Durchsatz erreicht, als dies beim «Stop-and-Wait ARQ» der Fall ist.

Der Empfänger führt damit Buch über die Sequenznummern der Pakete, die bei ihm ankommen. Er weiss dadurchgenau, welche er als nächstes zu erwarten hat. Stimmt die Sequenznummer des ankommenden Paketes mit der erwarteten Nummer überein, so sendet er die empfangene Sequenznummer mittels ACK zurück. Somit werden Duplikate bereits bestätigter Pakete oder Pakete mit höheren Sequenznummern ignoriert.

Der Sender sendet beispielsweise bei einer Window Size von 4 vier Pakete. Sobald der Empfänger das erste angekommene Paket mittels ACK bestätigt hat, verschiebt sich das Sliding Window um ein Paket. Der Sender kann nun drei weitere Pakete senden, bis das zuletzt kommunizierte Sliding Window voll ist.

Die Pakete können vom Sender allerdings auch kumulativ mit ***n+i*** bestätigt werden. Dabei bestätigt der Sender bei 3 erhaltenen Paketen (z.B. 0,1,2) nur ein ACK Nr. 2 und bestätigt somit auch die vorherigen Sequenznummern.

Im untenstehenden Diagramm überträgt der Sender die ersten vier Pakete. Der Sender bestätigt die ersten beiden. Beim dritten Paket mit der Sequenznummer 2 geht jedoch das ACK verloren. Dann werden SEQ 4 und 5 gesendet. Die beiden Pakete werden beim Empfänger verworfen, da sie ausserhalb der Sequenzreihenfolge ankommen. Der Sender muss nun alle vier Pakete erneut senden. Es betrifft die Pakete 2 bis 5.

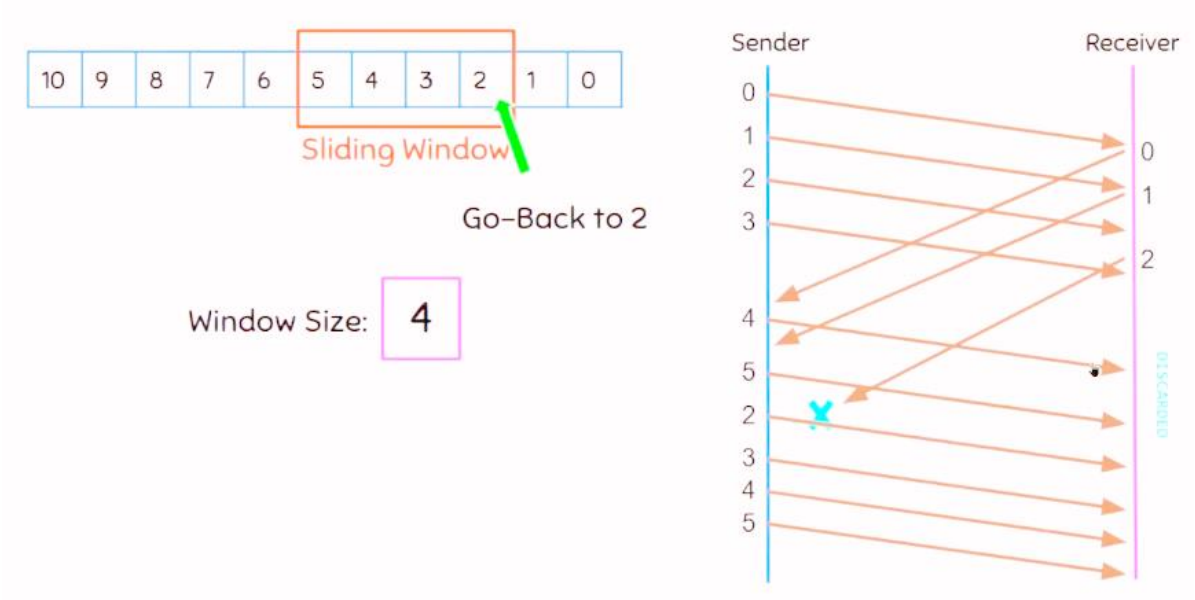

Abbildung 4: Go-Back-N ARQ Protokol – Sequnz- Diagramm [13]

Sollte vor Ablauf des RTO kein ACK beim Sender angekommen sein, so schickt er alle Pakete innerhalb des Sliding Windows erneut.

Das hauptsächliche Problem bei «Go-Back-N ARQ» besteht darin, dass alle Pakete innerhalb eines Sliding Windows erneut gesendet werden müssen, auch wenn nur eins davon unordnungsgemäss übertragen worden ist. Man geht also zurück zur letzten unbestätigten Sendungsnummer N (daher der Name «Go-Back-N»). Hiermit kann eine Menge an Übertragungskapazität verschwendet werden.

Ein Beispiel für «Go-Back-N ARQ» ist TCP (Transmission Control Protocol) ohne Options.

4.3. Selective Automatic Repeat Request (Selective ARQ)

Im Vergleich zu «Go-Back-N ARQ» verwirft «Selective ARQ» die in falscher Reihenfolge erhaltenen Pakete nicht, sondern speichert sie zwischen.

Nach einem RTO muss der Sender nur das älteste, nicht bestätigte Paket erneut senden. Kommt dieses Paket korrekt beim Empfänger an, kann er alle zwischengespeicherten Pakete vom Buffer an die Vermittlungsschicht übergeben. [14]

Im untenstehenden Diagramm überträgt der Sender, wie beim vorherigen «Go-Back-N» Beispiel, die ersten vier Pakete. Das dritte Paket mit der Sequenznummer 2 ist fehlerhaft, und wird vom Empfänger nicht in seinem Buffer abgespeichert. Der Sender kann nun statt allen vier Paketen, die sich im Sliding Window befinden erneut zu senden, einfach nur das fehlerhafte Paket Nummer 2 noch einmal senden. Anschliessend kann der Sender beim Paket mit der Sequenznummer 6 weiterfahren, da Nummer 3 bis 5 beim Empfänger nicht verworfen wurden. Die Window Size bestimmt hierbei, wie viele Pakete der Empfänger im Buffer behält, bevor er sie weiterschickt. Damit wird auch implizit bestimmt, wie viele Pakete der Sender nacheinander senden kann.

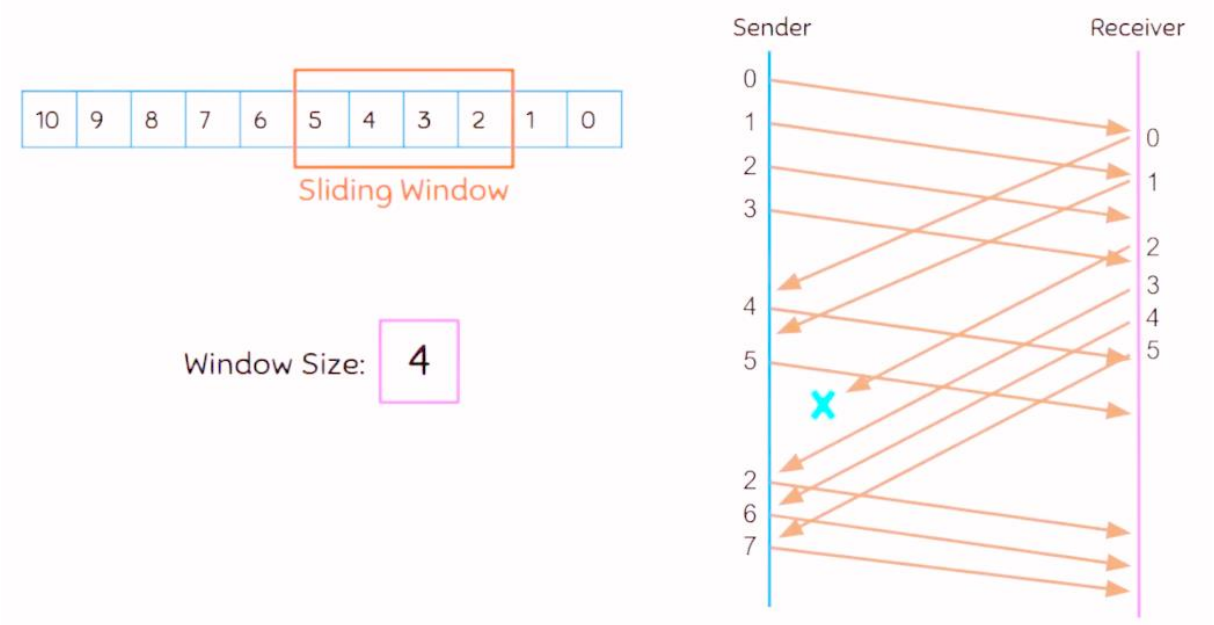

Abbildung 5: Selective Repeat ARQ Protokoll Diagramm [15]

Ein Beispiel für «Selective ARQ» ist TCP SACK [=Selective ARQ].

4.4. ARQ Implementation

Für die Implementation unseres Protokolls wurde die Variante Selective ARQ gewählt. Sie funktioniert ähnlich wie das TCP SACK Prinzip, das unten als Grafik abgebildet ist:

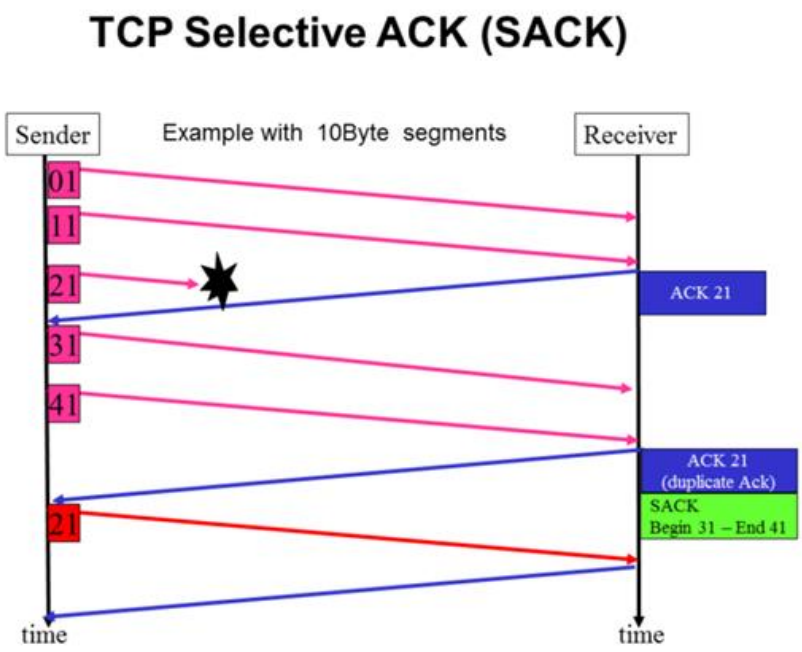

Abbildung 6 TCP SACK Sequnzdiagramm [11]

Beim TCP SACK werden Acknowledgements kumulativ mit einem ACK (bspw. ACK 21 + 31 + 41) bestätigt. In der hier implementierten Version versendet das ARQ für jedes Paket ein ACK und kennt entsprechend das «Cumultative ACK» von TCP nicht. Die Selective-ARQ Eigenschaft ist trotzdem erfüllt, da in falscher Reihenfolge ankommende Pakete in einem Buffer zwischenspeichert werden. Es wird zusätzlich zum «normalen» Bestätigungs-ACK das letzte ACK zurückgegeben, bis zu dem der Empfänger die Pakete in korrekter Reihenfolge ohne Fehlerfälle empfangen konnte.

Entsprechend würde der Empfänger in Analogie zum oberen SACK Sequenzdiagramm seine ACKs wie folgt kommunizieren:

Versendete Sequenz	**ACK (als nächstes erwartete Sequenz mit folgender Anfangsnummer)**	**Letztes ACK mit korrekter Reihenfolge**
01-10	11	11
11-20	21	21
21-30	31	21
31-40	41	21
41-50	51	51

Zwar werden in unserem Protokoll mehr ACKs versendet, was zu einer grösseren Netzauslastung führt. Dies bietet aber auch den entscheidenden Vorteil, dass der Sender theoretisch schneller auf ein vermeintlich verloren gegangenes Paket noch vor Ablauf der RTO reagieren kann. Dies wurde in dieser Arbeit jedoch nicht realisiert, könnte aber in einer Folgearbeit implementiert werden.

4.4.1. ARQ Szenario

Im untenstehenden Diagramm ist der Ablauf der in dieser Arbeit entwickelte Implementation des Selective ARQs dargestellt.

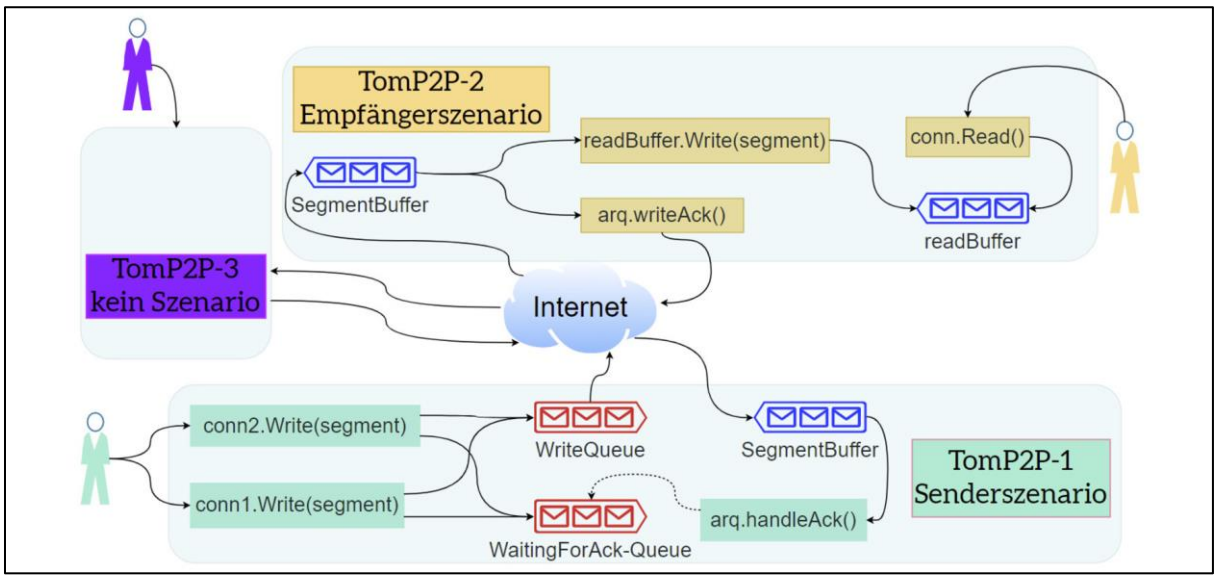

Abbildung 7 Übersichtsdiagramm des in dieser Arbeit implementierten Selective ARQs

Im einfachsten Fall öffnet ein Peer, der etwas senden möchte, eine Connection. Hierfür wird eine dem ARQ unterliegende UDP-Connection aufgebaut. Der Sender schreibt die zu sendenden Pakete über **`conn.Write(Segment)`** in die **`WriteQueue`**. Gleichzeitig speichert er dieses Segment im **`WaitingForAck-Queue`** und wartet so auf ein ACK vom Empfänger.

Die Pakete werden über die UDP-Connection an den Empfänger gesendet und landen dort im **`SegmentBuffer`**. Von hier liest der Empfänger die Pakete aus und überträgt sie mittels **`readBuffer.Write(segment)`** in seinen **`readBuffer`**. Nun kann der Empfänger mit **`conn.Read()`** die Pakete aus dem **`readBuffer`** auslesen. Parallel dazu sendet er dem Sender ein ACK als Bestätigung.

Der Sender verarbeitet das ACK und löscht es aus seiner **`WaitingForAck-Queue`**. Wurden alle Pakete erfolgreich übertragen, wird die Connection wieder geschlossen.

Im Ausnahmefall wird das Paket nochmals an den Empfänger gesendet. Dies geschieht nach Ablauf des RTT (default=3s) sofern in dieser Zeit kein ACK vom Empfänger beim Sender ankommt.

4.4.2. Wichtige implementierte Klassen

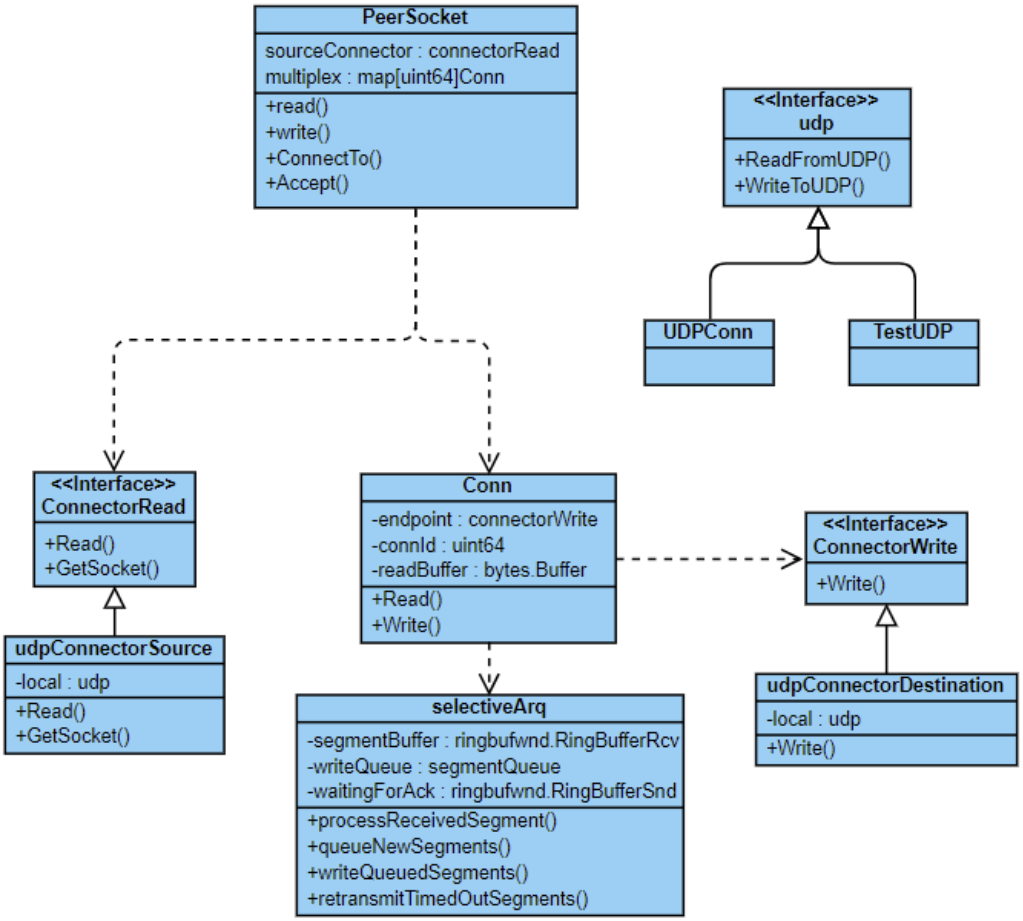

Abbildung 8 Klassendiagramm der implementierten ARQ Version

Das oben abgebildete Klassendiagramm erhebt keinen Anspruch auf Vollständigkeit und dient nur zum Verständnis der Implementation. Weniger zum Verständnis des grossen Ganzen beitragende Structs, Felder und Methoden wurden weggelassen. Zudem ist die Fehlerbehandlung wegen der Übersicht vollständig ausgeblendet. Auch wurden zur Vereinfachung die Methodenamen ohne Parameter und Rückgabewerte übernommen.

Eine wichtiges Struct ist das «Segment». Da dieses aber Verbindungen zu fast allen in der Grafik abgebildeten Structs hat, wurde es weggelassen. Dies macht die obige Abbildung durch die verringerte Anzahl Verbindungen anschaulicher und übersichtlicher.

PeerSocket

Der **PeerSocket** wird durch die Methode **SocketListen()** erstellt und verfügt über einen **SourceConnector**, der die eingehenden Pakete vom Socket entgegennimmt. Im Fall von UDP «hört» der **PeersSocket** auf eine gegebene Adresse. Beispielsweise könnte er auf der IP-Addresse 127.0.0.1 und den UDP-Port 1234 hören. Das Multiplex Feld ist für das Verwalten der Connections zuständig. Es ist vom Typ **map** und entspricht der Java-Map. Hier werden alle vom Peer mittels **ConnectTo()** eingegangen Verbindungen gespeichert. Mit **Accept()** kann er Verbindung von anderen Peers annehmen. Die **read()** und **write()** Methoden laufen dabei in einer Endlosschleife ab. Sie lesen und schreiben ständig von/auf der UDP-Schnittstelle. Dies wird durch die ausgehenden Pfeile vom **PeerSocket** Struct gekennzeichnet.

Interface ConnectorWrite und ConnectorRead

Um der implementierten Lösung einen Framework-Charakter zu verleihen, sind **ConnectorWrite** und **ConnectorRead** als Interfaces erstellt worden. Dadurch kann jeder, der diese Interfaces implementiert, die Logik der gesamten Lösung nutzen. Somit könnte beispielsweise statt einem UDP-Objekt ein TCP-Objekt mitgegeben werden. Dies würde praktisch wenig Sinn machen, da TCP selbst über ein eigene ARQ Implementation verfügt.

Interface ConnectorRead und udpConnectorSource

Dieses Interface bzw. **udpConnectorSource** kapselt das UDP-Objekt zur Kommunikation mit dem Internet. Sie wird hierbei nur zum Lesen verwendet und ist pro Peer/Benutzer einmalig. Alle beim Peer ankommenden UDP-Segmente werden hier abgeholt und zur Verarbeitung weitergegeben. Dabei wird **GetSocket()** zur internen Weitergabe des **local**-Feldes vom **udpConnectorSource** verwendet. Denn wenn ein Peer mit einer fremden IP-Addresse zum Verbindungsaufbau ein SYN-Paket sendet, muss der empfangende Peer eine neue Connection mit dem dazugehörigen UDP-Endpoint erstellen. Dieser UDP-Endpoint ist der gleiche, wie beim **local**-Feld vom **udpConnectorSource**. Mit der **GetSocket()** Funktion kann er diesen Socket erreichen und das **local**-Feld von **udpConnectorWrite** der neuen **Conn** initialisieren.

Interface ConnectorWrite und und udpConnectorWrite

Im Gegensatz zum **udpConnectorSource** wird dieses Struct für jede neu erstellte Verbindung kreiert. Damit können den anderen Peers Segmente über die **Write()** Funktion versendet werden.

Conn

Das **Conn** Objekt ist die Schnittstelle zur Kommunikation mit einem anderen Peer. Der Peer kann auf diese Weise mit einem anderen Peers, mit der er eine Verbindung aufgebaut hat, mittels **connection.Read()** bzw. **connection.Write()** kommunizieren. Dabei verfügt er über ein Feld namens **endpoint**. Dieser **endpoint** wird benötigt, um auf das UDP-Interface eines anderen Peers zu schreiben. Die **ConnID** dient der eindeutigen Identifikation der Connection. Der **readBuffer** dient als Zwischenspeicher für die in der korrekten Reihenfolge erhaltenen Segmente. Sobald der Benutzer **connection.Read()** aufruft, werden dort alle bis dahin von der UDP-Connection erhaltenen Bytes der Applikation bzw. dem Benutzer übergeben.

SelectiveArq

Das **`selectiveArq`** Objekt ist eine Art Manager für die Verwaltung der Segmente des Peers. Sobald der Peer **`connection.Write()`** aufruft, werden die vom Peer an die Connection übergebenen Daten mit **`queueNewSegments()`** der **`writeQueue`** des **`selectiveArq`** Objekts übergeben. Dies wird anschliessend mittels **`writeQueuedSegments()`** auf die UDP-Connection über das Connection-Objekt geschrieben.

In der Abbildung fehlt ein Pfeil vom **`selectiveArq`** zur Connection, welcher der Übersicht halber weggelassen wurde. Der fehlende Pfeil würde die Übergabe der Segmente an die Endpoints vom Conn Objekt und somit das Schreiben der Segmente auf die UDP-Schnittstelle repräsentieren.

Das **`waitingForACK`** Feld speichert noch nicht bestätigte Segmente ab. Falls sie innerhalb eines bestimmten RTO (default=3s) nicht bestätigt wurden, werden sie mit **`retransmitTimedOutSegments()`** erneut versendet.

Neue, mittels **`peer.read()`** erhaltene UDP-Segmente werden dem **`Conn`** Objekt übergeben, die mit **`processReceivedSegment()`** an den **`segmentBuffer`** von **`selectiveArq`** weitergegeben werden. Dieser stellt die korrekte Reihenfolge der erhaltenen Segmente sicher. Der Rückgabewert der Methode gibt die sortierten Segmente an den **`readBuffer`** zurück und behält die in falscher Reihenfolge angekommenen Segmente so lange zurück, bis die fehlenden Segmente die Reihenfolge komplettieren.

UDP Interface

Das UDP Interface ist die eigentliche Schnittstelle zur Kommunikation mit dem Internet über UDP. Sie wird in einem späteren 7.2 genauer beschrieben.

5. Bottleneck Bandwidth and Round-trip propagation time (BBR)

Bei TCP wird häufig der Verkehr verlangsamt, sobald Staus durch verlorene Packages wahrgenommen werden. Um dies zu verhindern, kommt BBR zum Einsatz. BBR wartet nicht, bis aufgrund eines Staus Datenpakete verloren gehen, sondern versucht die mögliche Kapazität zu errechnen und den Verkehr entsprechend umzuleiten.

Bei BBR handelt es sich, im Gegensatz zu dem bei TCP verwendeten, verlustbasierten Congestion Control Algorithmus (CCA), um einen modellbasierten Algorithmus. Verlustbasierte CCA sind auf Paket-Verluste angewiesen, um Congestion, also Stau, zu erkennen. Modellbasierte Algorithmen hingegen berechnen ein Modell des Netzwerks basierend auf Messdaten, was sie verlässlicher macht als verlustbasierte Algorithmen. Das BBR-Modell basiert auf der Schätzung von zwei Parametern:

- der Engpassbandbreite (**`BtlBw`**):
  geschätzt aus einer Stichprobe der maximalen Übertragungsrate
- des Round-Trip Propagation Delays (**`Rtprop`**):
  geschätzt aus einer Stichprobe des minimalen Round-Trip Delays

Das Ziel ist es, einen Betriebspunkt mit maximalem Durchsatz und gleichzeitig minimaler Verzögerung zu finden. Dies führt dazu, dass BBR die Daten mit einer Geschwindigkeit losschickt, die das Netzwerk auch bewältigen kann. Im Durchschnitt wird dadurch ein Durchsatz erreicht, der um 4% höher liegt als bei verlustbasierten CCAs.

Folgende Abbildung zeigt Round-Trip Time und Delivery Rate Schwankungen mit der Menge von Daten «in-flight»[1]. Die blauen Linien zeigen die **`Rtprop`** Beschränkung, die grünen Linien die **`BtlBw`** Beschränkung und roten Linien den Bottleneck Buffer. Betrieb in den schattierten Bereichen ist nicht möglich, da mindestens eine der Beschränkungen verletzt werden würde.

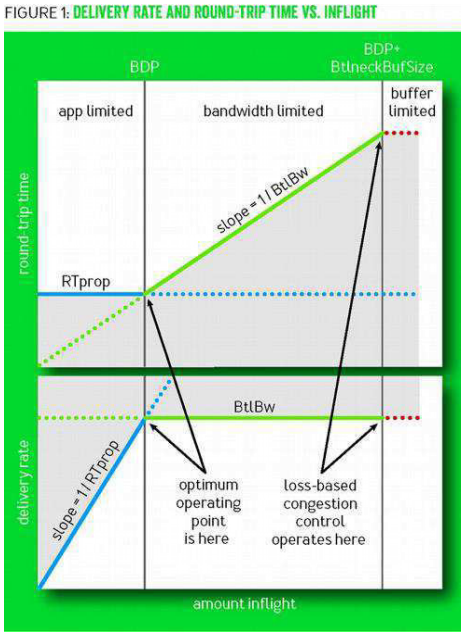

Abbildung 9 Round-Trip Time und Delivery Rate Schwankungen [16]

[1] Daten die gesendet, aber noch nicht bestätigt wurden

5.1. BBR Algorithmus

BBR verfügt über vier Zustände. Diese sind in der folgenden Darstellung visualisiert. In den n 5.1.2 wird auf diese Zustände genauer eingegangen.

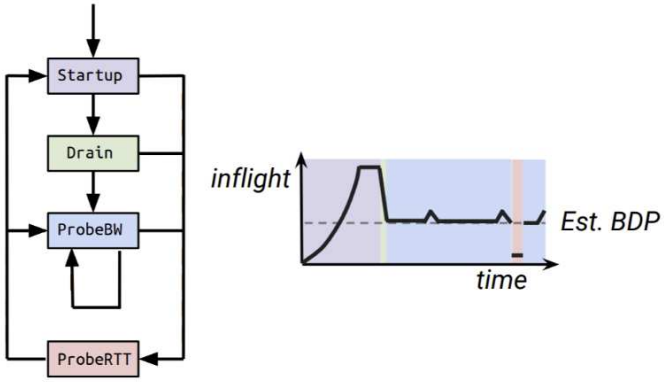

Abbildung 10 Die vier Zustände von BBR [17]

5.1.1. Begriffe

Die in diesem Kapitel behandelten Begriffe werden hier zuerst kurz erklärt.

Pacing_gain

Mit der **`pacing_gain`** variiert BBR die Sendegeschwindigkeit. Wenn die **`pacing_gain > 1`** ist, wird die Sendegeschwindigkeit grösser, der Zeitabstand zwischen den Paketen kleiner und es sind mehr Pakete «in-flight». Bei einer **`pacing_gain < 1`** gilt das Umgekehrte.

Pacing_rate

Die **`pacing_rate`** ist die Geschwindigkeit, mit der effektiv gesendet wird. Sie setzt sich zusammen aus **`pacing_gain * BBR.BtlBw * BBR.Rtprop = pacing_rate`**.

BBR.BtlBw

Die **`BBR.BtlBw`** ist die geschätzte «Flaschenhals»-Bandbreite. Bei jeder Verbindung existiert eine solche als tiefste Bandbreite. Ziel des BBR Algorithmus ist es, auf dieser Bandbreite zu senden, sodass der bestmögliche Durchsatz entsteht.

BBR.RTProp

Die **`BBR.RTProp`** ist die geschätzte RTT von BBR.

InitialCwnd

Zu Beginn weiss der Sender nicht wie gross sein **`Cwnd`**, also sein Congestion Window, sein soll. Dies wird anhand des Algorithmus versucht herauszufinden. Es muss aber mit einem initialen Wert gestartet werden, der meistens bei 1, 2 oder 4 MSS[2] liegt. [18]

[2] Maximum Segment Size

5.1.2. Startup

In der Startup-Phase beginnt die Einrichtung des BBR mit vordefinierten Parametern. Konkret wird zu Beginn die initiale **`pacing_rate = initialCwnd / 1ms`** gesetzt. Das **`initialCwnd`** kann unterschiedlich gross sein. Empfohlen werden 1, 2 oder 4 MSS. Da unsere UDP Segmente 1400 B gross sind, wären 2800 B = 2 MSS. Dies würde dann einer Anfangssendegeschwindigkeit von 2800kB/s entsprechen.

Nun wird ausgehend von der Anfangssendegeschwindigkeit versucht die **`BBR.BtlBw`** zu erreichen. Diese kann von einigen bps bis zu 100 Gbps reichen. Um diese Bandbreite möglichst schnell zu erreichen, wird die **`BBRHighGain`** als Faktor zur Erhöhung der gemessenen Bandbreite verwendet. Wenn nun also mit einer Senderate von **2800kB/s** begonnen wird, wird sie mit jeder RTT und den Faktor **`BBRHighGain`**, also **2/ln(2) ~= 2.89**, erhöht. Das bedeutet, dass die Bandbreite bei jeder RTT mehr als verdoppelt wird und somit schnell die maximal mögliche Datenrate erreicht wird. Ab einem Plateau registriert BBR, dass die maximal durchlässige Anzahl Pakete «in-flight» erreicht wurde und nicht mehr gesendet werden darf. Dies geschieht mit folgendem Algorithmus:

```
BBRCheckFullPipe():
    if BBR.filled_pipe or
       not BBR.round_start or rs.is_app_limited
      return  // no need to check for a full pipe now
    if (BBR.BtlBw >= BBR.full_bw * 1.25)  // BBR.BtlBw still growing?
      BBR.full_bw = BBR.BtlBw    // record new baseline level
      BBR.full_bw_count = 0
      return
    BBR.full_bw_count++ // another round w/o much growth
    if (BBR.full_bw_count >= 3)
      BBR.filled_pipe = true
```

In Worten gefasst, geht BBR davon aus, dass die Pipe voll ist, wenn die **`BBR.BtlBw`** trotz Erhöhung der Senderate 3-mal nacheinander um den Faktor 2.89 nicht signifikant erhöht (mind. Faktor 1.25) wird. Bei einer vollen Pipe wird dann in den Drain-Modus gewechselt.

5.1.3. Drain

Im Drain-Modus wird als erstes eine allfällig während dem Startup-Modus erzeugte Queue geleert. Wie im Code ersichtlich ist, wird auch hier der **pacing_gain** verwendet. Der **pacing_gain** wird weit unter 1 gewählt. Im untenstehenden Code wird er auf den Kehrwert von **BBRHighGain** gesetzt. Dadurch werden die Pakete «in-flight» reduziert. Ob nun die Pakete «in-flight» auf die gewünschte Anzahl reduziert wurde, wird mit der Funktion **BBRInflight(1.0)** überprüft. Sie berechnet aufgrund der geschätzten Werte **BBR.BtlBw** und **BBR.RTprop**, die maximal in der Leitung erlaubten Pakete. Dabei ist der Faktor 1.0 erneut der **pacing_gain.** Da dieser Faktor vor der Drain-Phase bei 2.89 lag, wird so lange in der Drain-Phase geblieben, bis sich die Paketanzahl «in-flight» auf ein **pacing_gain**-Äquivalent von 1.0 reduziert. Sobald dies erreicht wird, kann in den ProbeBW-Modus gewechselt werden.

Wie man die **packets_in_flight** berechnet, ist im RFC nicht aufgeführt. Unsere Lösung verfügt im **waitingforACK**-Feld vom **Arq** Struct über eine Funktion **NumOfSegments()**, die genau das berechnet. Dabei gibt sie die Pakete, die noch nicht bestätigt wurden und somit «in-flight» sind, wieder.

```
BBREnterDrain():
    BBR.state = Drain
    BBR.pacing_gain = 1/BBRHighGain  // pace slowly
    BBR.cwnd_gain = bbr_high_gain    // maintain cwnd

BBRCheckDrain():
    if (BBR.state == Startup and BBR.filled_pipe)
      BBREnterDrain()
    if (BBR.state == Drain and packets_in_flight <= BBRInflight(1.0))
      BBREnterProbeBW()  // we estimate queue is drained
```

5.1.4. ProbeBW

In diesem Modus wird davon ausgegangen, dass man die optimale Bottleneck Bandwidth erreicht hat. Nun wird nach jeder RTT die **pacing_gain** angepasst. Diese Anpassung erfolgt in 7 sogenannten Phasen. Dabei wird während der ersten beiden Phasen in einem kleineren Rahmen wieder eine Optimierung der Bottleneck Bandwidth vorgenommen.

In der ersten Phase ist die **pacing_gain** 1.25. Damit wird erneut versucht die höchstmögliche Datenrate zu erreichen. Anschliessend wird wegen einer allfällig in der Phase entstandenen Queue die **pacing_gain** auf 0.75 gesetzt, um die Queue zu leeren. Danach folgt 5 Phasen/RTTs lang ein **pacing_gain** von 1, um die erreichte, optimale Bandbreite zu erhalten. Dieser Vorgang wird während der gesamten Verbindung in einem Zyklus wiederholt.

5.1.5. ProbeRTT

ProbeRTT wurde im Rahmen dieser Arbeit nicht ausführlich angeschaut. Sie kommt während einer Verbindung sehr selten zum Einsatz. Mit ihr soll bestimmt werden, ob sich die RTT während der Verbindung signifikant verändert hat und somit neu geschätzt werden muss.

5.2. Pacing

Das Pacing ist unverzichtbar für den BBR-Algorithmus. Durch Pacing reguliert der Sender die Sendegeschwindigkeit. Diese wird in Bytes/s gemessen. Der Kehrwert der Sendegeschwindigkeit sagt aus, wie viel Zeit vergeht, bis ein Byte versendet wurde. Er wird als Masseinheit in s/Byte angegeben. Wenn man diesen Wert mit 1400 multipliziert, erhält man die Zeitdauer für das Versenden eines Pakets. Der nächste Zeitpunkt für das Versenden eines Pakets wird mit folgendem vorgegebenem Code implementiert.

```
next_send_time = Now() + packet.size / pacing_rate
```

Dabei sagt die **pacing_rate** mittels **pacing_gain** aus, ob schneller oder langsamer als zuvor gesendet werden soll.

5.3. BBR Implementation

Der BBR-Algorithmus wurde für die Implementation des Protokolls in dieser Arbeit aus dem Draft-RFC [19] übernommen. Er konnte am Ende aber leider nicht vollständig getestet oder ausgeführt werden. Ein korrekt funktionierender BBR-Algorithmus bedingt einer genauen «Delivery Rate Estimation». Diese konnte wegen den in Evaluation: Delivery Rate Estimation aufgeführten Hindernissen, nicht in ausreichender Genauigkeit implementiert werden.

Jedoch wurde das Pacing, welches Teil von BBR ist, mit einem konstanten Zeitabstand implementiert und getestet.

```
//timeToWait:= seg.size()/arq.bbr.BBRGetPacingRate * 1000000000 //* 1000000000 =>
to Nanoseconds
timeToWait := arq.timeToWait
nextSendtime := arq.bbr.last_time_send.Add(time.Duration(timeToWait))
   for {
      if !time.Now().Before(nextSendtime) {
      break
      }
   }
```

Der auskommentierte Code kommt nur zur Anwendung, wenn BBR eingesetzt wird und somit dynamisch die Sendezeit bestimmt wird.

In unserem Fall wurde aber eine konstante Zeit für das Testen der «Delivery Rate Estimation» gewählt. Hierbei wird diese Zeit in der Variable **arq.timeToWait** gesetzt. In unserem Fall sind das 10ms.

6. Security

Für die Implementation von Ver- und Entschlüsselung wurde, nach Betrachtung von verschiedenen Optionen, das Noise Framework [20] gewählt. Ausschlaggebend dafür waren die existierende native Go Implementation [21], sowie der Fakt, dass dieses Framework auch bereits bei ATP-go eingesetzt wurde.

Zum Reduzieren der benötigten Nachrichten beim Verbindungsaufbau wurde auf das Aushandeln der zu verwendenden Verschlüsselung verzichtet und weitverbreitete Standards festgesetzt. Konkret wurden Curve25519 als Diffie-Hellman-Schlüsselaustausch-Funktion, BLAKE2b als Hash-Funktion und ChaCha20-Poly1305 als Cipher-Funktion gewählt.

6.1. Handshake-Pattern

Das Noise Framework unterstützt 12 fundamentale interaktive Handshake-Patterns für den Schlüssel-Austausch. Diese Patterns sind mit zwei Buchstaben benannt, welche den Status der Static Keys des Initianten und des Antwortenden indizieren.

Der erste Buchstabe steht für den Static Key des Initianten:

- N = **N**o static key for initiator
- K = Static key for initiator **K**nown to responder
- X = Static key for initiator **X**mitted ("transmitted") to responder
- I = Static key for initiator **I**mmediately transmitted to responder, despite reduced or absent identity hiding

Der zweite Buchstabe steht für den Static Key des Antwortenden:

- N = **N**o static key for responder
- K = Static key for responder **K**nown to initiator
- X = Static key for responder **X**mitted ("transmitted") to initiator

Noise unterscheidet zwischen zwei Arten von Schlüsseln, den oben erwähnten Static Keys und den Ephemeral Keys. Statische Schlüssel sind für den Langzeitgebrauch designiert und können wiederverwendet werden. Ephemeral Keys werden für jede Session neu generiert und nur innerhalb dieser gebraucht.

In dieser Arbeit wurde das IX-Pattern gewählt, das heisst der Initiator sendet seinen Static Key als Klartext im SYN-Paket. Der Antwortende kann dann seinen Static Key, sowie die generierten gemeinsamen Schlüssel, mit dem Static Keys des Initianten verschlüsselt zurücksenden. Somit braucht es von den beiden Kommunikations-Teilnehmern jeweils eine Nachricht für den Schlüssel-Austausch. Vom Sender das SYN-Paket und vom Empfänger das darauf antwortende ACK.

```
IX:
-> e, s
<- e, ee, se, s, es
```

- s = static key
- e = ephemeral key
- ee, se, es = gemeinsamer DH-Schlüssel

Bei diesem Pattern wird das SYN-Paket unverschlüsselt gesendet, was bedeutet, dass keine Perfect Forward Secrecy herrscht.

Um bereits den ersten Handshake-Payload verschlüsseln zu können, wäre auf eine 0-RTT Verschlüsselung umzustellen. Um solch eine 0-RTT Verschlüsselung zu ermöglichen, müsste das Pattern so angepasst werden, dass der Initiator den Static Key des Antwortenden schon kennt. Dies wäre beispielsweise über das IK-Pattern möglich. Diese Umstellung könnte Teil einer Folgearbeit sein.

6.2. Curve25519

Bei Curve25519 handelt es sich um ein kryptographisches Verfahren auf Basis von elliptischen Kurven. Der Schlüsselaustausch erfolgt hierbei über Diffie-Hellman. Ein Vorteil Verschlüsselungskonzepte mit elliptischen Kurven besteht unteranderem in der Kürze der Schlüsseln. [22]

Ein Problem beim Einsatz von Verfahren mit elliptischen Kurven besteht darin, sie eine komplexe Mathematik aufweisen. Es muss auf die richtige Kurve gesetzt werden, damit sie Verschlüsselung sicher ist. Curve25519 setzt, wie bereits am Namen zu erkennen ist, auf eine kurze Kurvengleichung mit der Primzahl 2^{255-19}. Sie ist somit relativ einfach, womit Fehler in der Anwendung verhindert werden können. [23]

6.2.1. BLAKE2b

Bei BLAKE2b handelt es sich um eine Hash-Funktion, die das SHA-3 Verfahren unterstützt. Die Version «b» von BLAKE ist für 64 Bit optimiert. Für 8 – 32 Bit Plattformen existiert die Variante BLAKE2s. BLAKE2 gehörte zu den Finalisten im SHA-Auswahlsverfahren der NIST. Laut Entwickler ist es genau so sicher wie SHA-3 und eine der schnellsten kryptographischen Hashfunktionen. BLAKE2 ist von ChaCha abgeleitet. [24]

6.3. ChaCha20 - Poly1305

Bei ChaCha20 handelt es sich um einen Verschlüsselungs-Algorithmus. Im Gegensatz zum bekannten Advanced Encryption Standard (AES) arbeitet der ChaCha20 bitweise statt blockweise, was ihn sicherer macht. Zudem ist die Implementierung eines ChaCha20 einfacher als bei einem AES.

Poly1305 ist ein Message Authentication Code (MAC). Er wird verwendet, um die Datenintegrität und die Authentizität einer Nachricht zu verifizieren. [25]

7. Testing in Golang

7.1. Herausforderungen beim Testing

Die Schwierigkeit beim Testing unseres Golang Codes war es Unit-Tests zu schreiben. Das was unter-anderem dadurch bedingt, dass viele Funktionen voneinander abhängig waren. Sauberes Testen der einzelnen Funktionen verlangt jedoch, dass diese Abhängigkeiten gemockt werden.

Am Beispiel des write-loops vom Peer, ist dies gut ersichtlich:

```
func (socket *PeerSocket) write() {
   for {
      socket.multipartLock.Lock()
      for _, c := range socket.multiplex {
         c.arq.retransmitTimedOutSegments(timeNow(), c.arq.bbr)
         _, err := c.arq.writeQueuedSegments(timeNow())
         if err != nil {
            socket.errorHandler(err)
         }
      }
      socket.multipartLock.Unlock()
      time.Sleep(1*time.Nanosecond)
   }
}
```

In diesem Code-Ausschnitt ist zu sehen, dass einerseits Funktionen wie beispielsweise:

```
c.arq.retransmitTimedOutSegments(timeNow(), c.arq.bbr),
_, err := c.arq.writeQueuedSegments(timeNow())
```

abhängig von anderen Funktionen sind. Wenn sauber nach Lehrbuch unit-tested werden soll, müssten diese beiden Funktionen gemockt werden. Mocken ist in Golang aber wegen nicht vorhandener Vererbung umständlich und produziert viel Code.

Des Weiteren gibt es in der Implementation Endlos-Schleifen, die bis zum Ende der Verbindung andauern. Diese könnten mit Delegates elegant gelöst werden. Darauf wird hier aber nicht tiefer eingegangen, da das Testing-Problem mit einem anderen ganzheitlichen Ansatz gelöst werden konnte.

7.2. Mocking

Das Mocken beansprucht zwar viel Code und ist komplex, wenn es aber nur an konkreten Stellen angewendet wird, kann die Komplexität und die Code Duplikation in einem akzeptablen Rahmen gehalten werden.

Dabei stellte es sich als am einfachsten heraus, das UDP-Interface durch einen selbst implementieren Mock zu ersetzen.

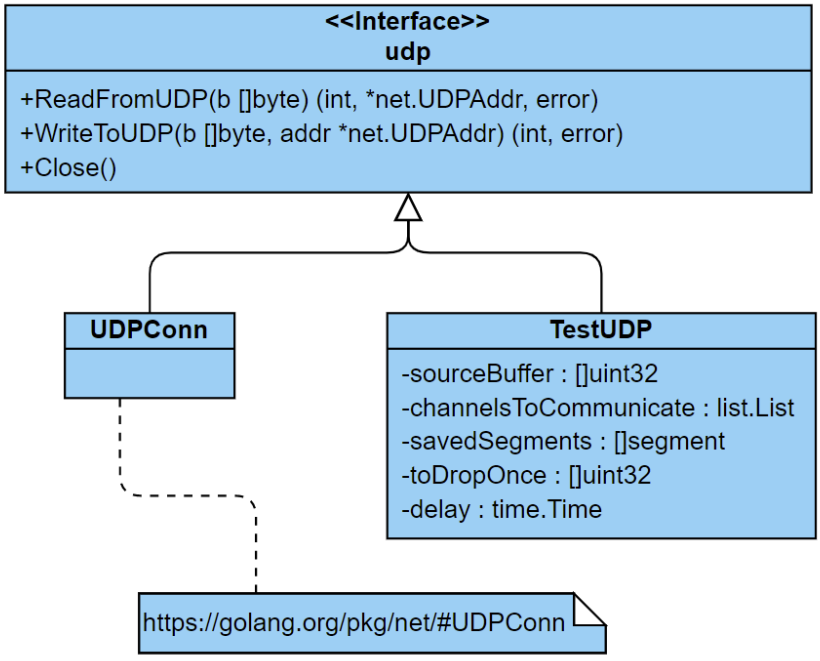

Abbildung 11 Mocking des UDP Interfaces

In der obigen Grafik ist das UDP Interface als Struct mit den für die Implementation verwendeten Methoden zu sehen. Diese werden vom «UDPConn» Struct des built-in Golang Packages «net» implementiert, das links in der Grafik abgebildet ist. Es wird für den produktiven Code verwendet. Der Übersicht halber wurden alle Methoden und Felder von «UDPConn» weggelassen. Für nähere Informationen zu dieser Klasse sei auf den in der Grafik aufgeführten Link[3] verwiesen. Rechts davon ist das Mock-Struct aufgeführt. Der implementierte Mock verfügt über verschiedene Attribute, die im Folgenden genauer erklärt werden.

[3] https://golang.org/pkg/net/#UDPConn

SourceBuffer

Der **`SourceBuffer`** ist ein Golang-Channel. Channels sind für die Kommunikation zwischen zwei verschiedenen Goroutines zuständig. Weil für das Testing jeweils zwei Peers in separaten Threads verwendet werden, eignen sich Channels gut für die Kommunikation der UDP-Pakete zwischen den Peers. Hierbei stellt der **`sourceBuffer`** den Empfangskanal des Peers dar, auf den andere Peers schreiben können.

ChannelsToCommunicate

`ChannelsToCommunicate` stellt eine Liste von Golang-Channels dar, mit denen kommuniziert werden soll. Wenn beispielsweise Peer1 mit Peer2 und Peer3 kommunizieren möchte, muss er den im **`sourceBuffer`** befindlichen Golang-Channel von Peer2 und Peer3 in dieser Liste eintragen.

SavedSegments

`SavedSegments` beinhaltet alle Segmente, die während einer Verbindung von einem Peer zum anderen übertragen werden. Dabei wird bei jedem Empfang eines Pakets neben dem Weiterleiten auf den **`sourceBuffer`**, das Paket auch bei **`savedSegments`** zwischengespeichert. Dadurch kann nach einer Testverbindung jedes Segment in der Reihenfolge, in der sie der Peer empfangen hat, mit den erwarteten Segmenten verglichen werden. So wird neben der Reihenfolge auch der korrekte Inhalt des Segments sichergestellt.

ToDropOnce

Dieses Feld beinhaltet Segmente, die beim Senden verworfen werden. Soll beispielsweise das 2te ACK-Segment vom Empfänger an den Sender nicht gesendet werden, wird es in diesem Feld angegeben. Hier kann auch überprüft werden, ob ein Retransmit erfolgt ist, und ob das dazugehörige ACK noch einmal erfolgreich versendet wurde.

Delay

Dieses Feld wurde nicht implementiert. Theoretisch könnte man hier aber einen Delay simulieren.

7.3. Testcode Initialisierung

Die Initialisierung wird vor jeder Testmethode ausgeführt. Sie ist in der Methode **`SetupTest()`** deklariert und im folgenden Code-Ausschnitt abgebildet. Die Testcode Initialisierung wurde in drei Teile unterteilt, auf die im Folgenden genauer eingegangen wird.

```
func (suite *PeerSocketTestSuite) SetupTest() {
   /*------------------------Teil 1 Start------------------------*/
   endpoint1 := make(chan []byte, 1000000)
   endpoint2 := make(chan []byte, 1000000)
   manipulator1 := &testUDP{
      savedSegments: make([]*segment, 0),
      toDropOnce:    make([]uint32, 0),
      sourceBuffer:  endpoint1,
   }

   manipulator2 := &testUDP{
      savedSegments: make([]*segment, 0),
      toDropOnce:    make([]uint32, 0),
      sourceBuffer:  endpoint2,
   }
    manipulator2.pushChannelToCommunicate(endpoint1)
   /*------------------------Teil 1 Ende------------------------*/

   /*------------------------Teil 2 Start------------------------*/
   socketListen1 := &udpConnectorSource{
      local:        manipulator1,
      errorHandler: func(err error) { log.Printf("Error :%v\n", err) },
   }
   socketListen2 := &udpConnectorSource{
      local:        manipulator2,
      errorHandler: func(err error) { log.Printf("Error :%v\n", err) },
   }
   /*------------------------Teil 2 Start------------------------*/

   /*------------------------Teil 3 Start------------------------*/
   suite.socketA = SocketListen(socketListen1)
   suite.socketB = SocketListen(socketListen2)
   suite.endpoint1 = endpoint1
   suite.endpoint2 = endpoint2
   suite.manipulator1 = manipulator1
   suite.manipulator2 = manipulator2
   /*------------------------Teil 3 Start------------------------*/
}
```

Teil 1

Im ersten Teil werden die Endpoints, über die kommuniziert werden soll, erstellt. Es handelt sich hierbei um Golang-Channels. Diese werden von beiden Peers genutzt, um untereinander zu kommunizieren. Anschliessend werden zwei **`TestUDP`** Objekte erstellt. Soll eine reale Verbindung getestet werden, müsste an dieser Stelle statt eines **`TestUDP`** ein reales UDP Objekt erstellt werden. Dieses wird pro Peer einmal benötigt. Die Endpoints werden den Manipulatoren übergeben, auf denen die Peers später die empfangenen Segmente lesen werden.

Hierbei ist der **`manipulator1`** für Peer **`socketA`** vorgesehen und der **`manipulator2`** für Peer **`socketB`**. Die gesetzten Felder für das **`testUDP`** wurden bereits im vorherigen Kapitel beschrieben. Das Feld **`toDropOnce`** ist auskommentiert. Das liegt daran, dass der Standardfall getestet wird und somit keine Pakete bei der Sendung verloren gehen sollen.

Später wird **`manipulator2.pushChannelToCommunicate(endpoint1)`** aufgerufen. Somit wird das Feld «**`ChannelToCommunicate`**» im **`testUDP`** gesetzt, womit **`socketB`** auf den gleichen Endpoint schreiben kann, auf den **`SocketA`** liest.

Teil 2

In Teil zwei werden, wie bereits in 4.4.2 beschrieben, zwei **`udpConnectorSource`** Objekte erstellt. Hier ist das local Feld statt einer realen Verbindung ein **`testUDP`** Objekt.

Teil 3

In Teil drei wird der **`udpConnectorSource`** an die **`SocketListen`** Funktion übergeben, wodurch ein **`socket`** erzeugt wird. Nun ist der **`socket`** bereit neue Verbindungen zu erzeugen oder anzunehmen. Dieser **`socket`** und auch die **`endpoint`** und **`manipulator`** Objekte werden dem **`suite`** Objekt zugewiesen. Das hat den Vorteil, dass im späteren Verlauf in anderen Testmethoden auf diese Objekte zugegriffen werden kann.

7.4. Testcode Test

In diesem Kapitel wird genauer auf die eigentliche Testmethode eingegangen. Auch dieser ist in drei Schritte unterteilt, die im Verlauf des Kapitels nacheinander beschrieben werden.

Zuerst ist der Ablauf des Testcodes als Sequenzdiagramm dargestellt:

```
/*
+---------+                +-----------+
| Sender  |                | Receiver  |
+---------+                +-----------+
     |                           |
     | SYN                       |
     |-------------------------->|
     |                           |
     |                   ACK SYN |
     |<--------------------------|
     |                           |
     |           "Hello World B" |
     |<--------------------------|
     |                           |
     | ACK "Hello World B"       |
     |-------------------------->|
     |                           |
     | "Hello World A"           |
     |-------------------------->|
     |                           |
     |       ACK "Hello World A" |
     |<--------------------------|
     |                           |
*/

func (suite *PeerSocketTestSuite) TestOneToOneConnectionNormalBehavior() {

   /*------------------------Teil 1 Start------------------------*/
   expectedA := []byte("Hello World A")
   expectedB := []byte("Hello World B")
   mutex := sync.WaitGroup{}
   mutex.Add(2)

   manipulator := &testUDP{
      savedSegments: make([]*segment, 0),
      //toDropOnce:    make([]uint32, 0),
   }
   manipulator.pushChannelToCommunicate(suite.endpoint2)
   /*------------------------Teil 1 Ende------------------------*/
```

Teil 1

In Teil eins wird der Kommunikationskanal für **socketA** gesetzt, damit er nach **socketB** kommunizieren kann. Dabei funktioniert **manipulator.pushChannelToCommunicate(suite.endpoint2)** nach dem gleichen Prinzip, wie es im vorherigen Kapitel in Teil 1 der Fall war.

```
/*------------------------Teil 2 Start------------------------*/
go func() {
    conn := suite.socketA.ConnectTo(&udpConnectorDestination{
        local:        manipulator,
        errorHandler: func(err error) { log.Printf("Error :%v\n", err) },
        remoteAddr:   createUDPAddress("127.0.0.1", 3033),
    }, uint32(0))
    suite.NotNil(conn)
    //Channel's are slow in Performance. So wait until SYN Packet is processed.
    time.Sleep(25 * time.Millisecond)
    n, err := conn.Write(expectedA)
    suite.handleTestError(err)
    buffer := make([]byte, 128)
    n, err = conn.Read(buffer)
    fmt.Println("buf1: ", buffer)
    suite.Equal(expectedB, buffer[:n])
    suite.handleTestError(err)
    mutex.Done()
}()
go func() {
    conn :=suite.socketB.Accept()
    suite.NotNil(conn)

    n, err := conn.Write(expectedB)

    suite.handleTestError(err)
    buffer := make([]byte, 128)

    n, err = conn.Read(buffer)

    suite.Equal(expectedA, buffer[:n])
    suite.handleTestError(err)
    mutex.Done()
}()
mutex.Wait()
/*------------------------Teil 2 Ende------------------------*/
```

Teil 2

Im zweiten Teil wird der eigentliche Testcode ausgeführt. Mit dem in 4.4.2 vorgestellten Klassendiagramm und deren Beschreibungen, sollte dieser Code selbsterklärend sein.

Zudem ist hier ersichtlich, dass der Code ohne Weiteres genau wie der Production Code geschrieben wird. Mit dem einzigen Unterschied, dass das UDP-Objekt jetzt ein **`testUDP`** Objekt ist.

```
   /*------------------------Teil 3 Start------------------------*/
   //suite.manipulator1.savedSegments is not synchronized. Therefore wait 1 sec.
Must sync it programmatically.
   time.Sleep(1000 * time.Millisecond)

   expectedACK0SegmentA := createAckSegment(0,0,initialReceiverWindowSize-1)
   expectedSEQSegmentA := createFlaggedSegment(0, flagSEQ, expectedB)
   expectedACK1SegmentA := createAckSegment(1,1, initialReceiverWindowSize-1)

   //set expected for socket A receiver
   expectedBufferA := make([]*segment, 0)
   expectedBufferA = append(expectedBufferA, expectedACK0SegmentA)
   expectedBufferA = append(expectedBufferA, expectedSEQSegmentA)
   expectedBufferA = append(expectedBufferA, expectedACK1SegmentA)
   segmentSliceEquals(suite, expectedBufferA, suite.manipulator1.savedSegments)

   //set expected for socket B receiver
   synBuffer := make([]byte, 0)
   expectedSynSegmentB := createFlaggedSegment(0, flagSYN, synBuffer)
   expectedACKSegmentB := createAckSegment(0,0, initialReceiverWindowSize -1)
   expectedSEQSegmentB := createFlaggedSegment(1, flagSEQ, expectedA)
   expectedBufferB := make([]*segment, 0)
   expectedBufferB = append(expectedBufferB, expectedSynSegmentB)
   expectedBufferB = append(expectedBufferB, expectedACKSegmentB)
   expectedBufferB = append(expectedBufferB, expectedSEQSegmentB)
   segmentSliceEquals(suite, expectedBufferB,suite.manipulator2.savedSegments)
   /*------------------------Teil 3 Ende------------------------*/
}
```

Teil 3

In Teil drei werden die eigentlichen Assertions gemacht. Dabei werden die zu erwartenden Segmente durch die auch im Production Code verwendeten Factory Funktionen wie beispielsweise **`createAck-Segment()`** erzeugt. Hierbei ist es wichtig, dass die **`expected segments`** in der korrekten Reihenfolge in einer Slice gespeichert werden, sodass sie später mit der Methode **`segmentSliceEquals()`** mit den **`actual segments`** verglichen werden können, die im **`savedSegments`** Feld vom **`testUDP`** Struct gespeichert sind.

8. Delivery Rate Estimation [17]

Eine wichtige Rolle für eine erfolgreiche Implementation des BBRs übernimmt die «Delivery Rate Estimation». Für die Berechnung dieser wird im Draft RFC [22] ein passender Algorithmus vorgeschlagen.

Die Delivery Rate Estimation schätzt aufgrund von Sende- oder Empfangszeitpunkt der ACKs die Sendegeschwindigkeit seitens Sender ab und ist somit unabdingbar für den BBR. Denn der BBR kann erst dadurch seine Sendegeschwindigkeit bestimmen und regulieren.

8.1. Ack_rate

Mit der **ack_rate** wird die Anzahl abgelieferter Daten pro Zeiteinheit ausgedrückt.

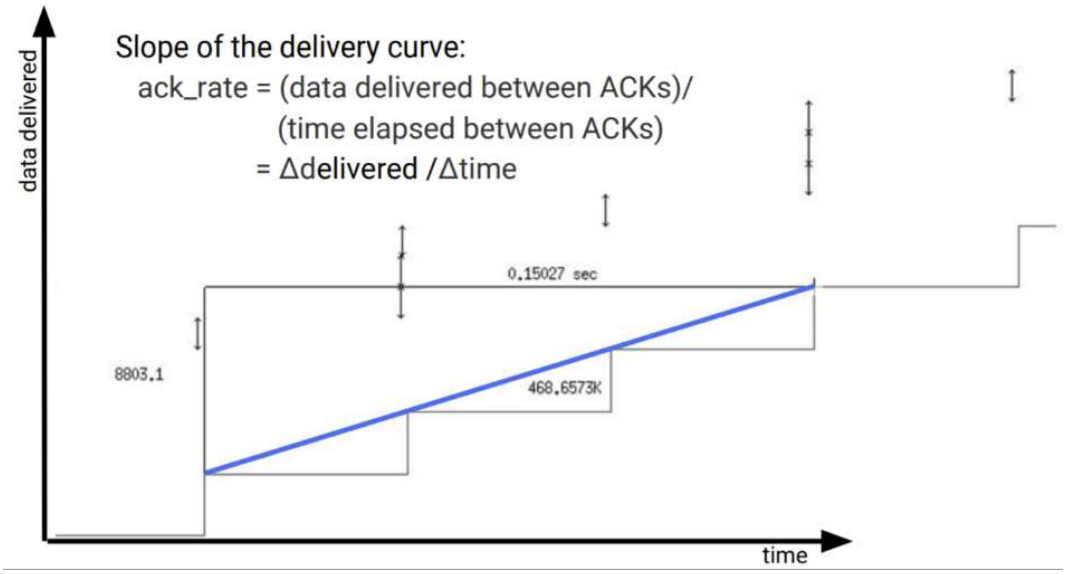

Abbildung 12 Anzahl abgelieferte Daten pro Zeiteinheit [17]

Im obigen Bild wird die **ack_rate** berechnet. Auf der X-Achse ist die Zeit, die zwischen drei erhaltenen ACKs vergangen ist, dargestellt. Auf der Y-Achse sind während dieser Zeit drei ACKs angekommen. Daraus kann der Sender folgern, dass während dieser Zeit drei Pakete mit n Bytes geliefert wurden. Die blaue Steigung stellt dabei die **ack_rate** dar.

Zur Berechnung der **ack_rate** wird ausdrücklich davor gewarnt auf eine Berechnung mit dem RTT zurückzugreifen. Der Grund dafür ist im folgenden Bild veranschaulicht und wird weitergehend diskutiert:

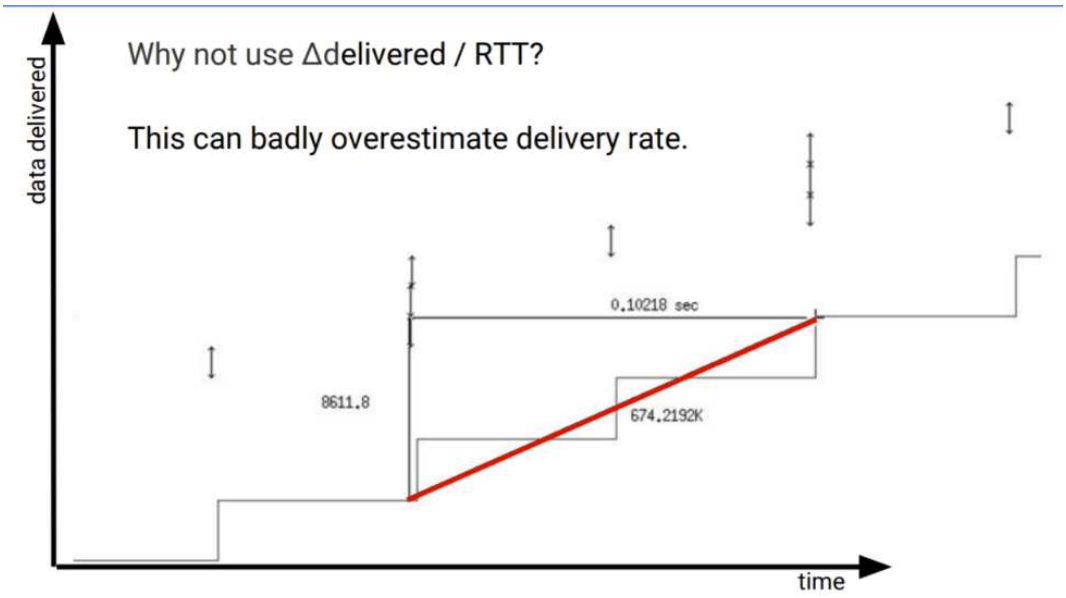

Abbildung 13 Negativbeispiel mit stark überschätzer Delivery Rate durch Verwendung des RTT [17]

Im obenstehenden Diagramm ist ein Worst Case veranschaulicht, bei dem der Sendezeitpunkt eines Pakets sehr nahe vor dem Empfang des nächsten ACKs liegt. Dabei reflektiert die vergangene Zeit auf der X-Achse die RTT des Pakets. Die Y-Achse hingegen repräsentiert die ab dem Sendezeitpunkt erhaltenen ACKs, also die Menge an gelieferten Daten.

Aufgrund der unterschiedlichen Steigungen der blauen Geraden aus dem ersten Diagramm, und der roten Geraden aus dem zweiten Diagramm ist ersichtlich, dass die beiden nicht gleichgesetzt werden können. Zudem sollte die rote Kurve entlang der Spitzen der Treppenfunktion verlaufen, wie die blaue Kurve in der vorherigen Grafik.

8.2. Send_rate

Anstatt wie bei der **ack_rate** die vergangene Zeit zwischen zwei ACKs zu messen, kann auch die vergangene Zeit zwischen zwei versendeten Paketen gemessen werden. Diese Rate wird **send_rate** genannt. Hierbei werden wieder die Anzahl ACKs, die man während dieser Zeit erhalten hat, gemessen, wodurch man die **send_rate** erhält. Zur Veranschaulichung ist sie in der unteren Grafik abgebildet.

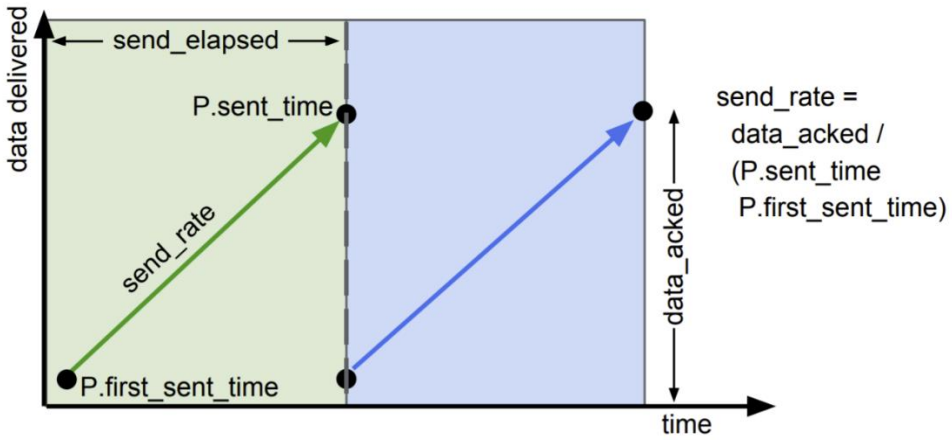

Abbildung 14 Berechnung der send_rate [16]

Diese ist aufgrund der ACK Compression, auf die im folgenden Unterkapitel genauer eingegangen wird, wichtig. Denn durch die ACK Compression treffen ACKs im kleinen Zeitabstand ein, wie in folgender Darstellung visualisiert wurde. Dadurch kann die gemessene **ack_rate** viel höher sein, als es die **ack_rate** tatsächlich ist. Durch diese Verfälschung würde der BBR Algorithmus nicht korrekt funktionieren.

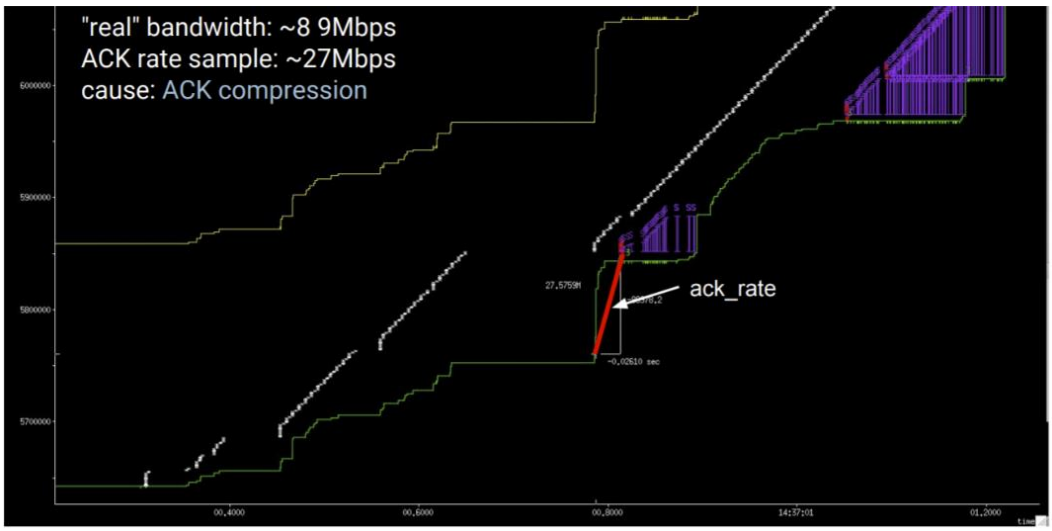

Abbildung 15 Darstellung Verfälschund der ack_rate durch ACK Compression [17]

Zur Vermeidung dieses Problems wird deshalb die **send_rate** verwendet. Physikalisch ist es nämlich unmöglich, dass die **ack_rate** schneller als die **send_rate** ist. Sollte sie es doch sein, handelt es sich um eine Verfälschung und die **ack_rate** wird durch die **send_rate** ersetzt.

8.2.1. Ack Compression

Die ACK Compression ist ein Phänomen, das bei einer realen Verbindung als unerwünschter Nebeneffekt auftritt. Dabei kommen die in einer bestimmten Rate versendeten Pakete nicht in derselben Rate beim Empfänger an. Denn einige der Pakete kommen beispielsweise durch Queuing Delays im Netz verspätetet beim Empfänger an. Somit empfängt der Sender die ACKs in sogenannten Bursts, also sehr viele auf einmal.

8.3. Delivery Rate

Die eigentliche Delivery Rate wird aus dem Minimum der **`ack_rate`** und **`send_rate`** bestimmt. Die Überlegung ist hierbei, dass es physikalisch nicht möglich ist, Daten schneller zu empfangen (**`ack_rate`**) als dass man sie sendet (**`send_rate`**). Wenn also die **`ack_rate`** grösser als **`send_rate`** ist, was wegen der ACK Compression vorkommen kann, wird sie herausgefiltert.

8.4. Application Limited Phases

Grundsätzlich kommt es vor, dass die Applikation selbst keine Daten sendet, weil es zu diesem Zeitpunkt nichts zu senden gibt. Mittels «Application Limited Phases» werden solche Intervalle herausgefiltert. Blieben sie ungefiltert enthalten, würde die Delivery Rate falsch berechnet werden und wäre dementsprechend unbrauchbar.

Diese Funktion wurde bei uns aus Zeitgründen nicht implementiert. In der Evaluation der Delivery Rate wurden deshalb immer Daten gesendet, um Effekte von «Application Limited Phases» zu vermeiden.

8.5. Delivery Rate Estimation Implementation

Für die Implementation wurde der im Draft-RFC [22] vorgeschlagene Algorithmus auf Golang übersetzt.

Die davon betroffenen Funktionen aus dem File **`selectiveArq.go`** sind: **`generateRatesample()`**, **`updateRateSample()`** und diverse Anpassungen der **`writeQueuedSegmets()`**.

Der RFC verwendet eine nicht implementierte Variable **`C.pipe()`**. Bei dieser Variable werden noch nicht bestätigte Pakete, sogenannte «in-flight» Pakete beschrieben. Sie konnte mit der vom Betreuer Dr. Thomas Bocek zur Verfügung gestellten Library **`ringBufwnd`** implementiert werden. Diese verfügt über eine **`NumOfSegments()`**-Funktion, die genau diese in-flight Pakete zurückgibt.

Der Code-Teil mit SACK musste nicht erneut nach dem Draft RFC [22] implementiert werden, da unser ARQ für jedes Packet ein ACK versendet und bereits mehrfach erhaltene ACKs schon vor Eintritt in den «Delivery Rate Estimation» Algorithmus-Teil verwirft. Implementiert wurde jedoch eine **`MinRTT`**-Funktion, die benötigt wird, wenn Pakete erneut versendet werden.

Durch das neu versenden von Paketen wegen abgelaufenem RTO werden Pakete zwischen dem Pacing-Intervall von beispielsweise 10ms «hereingeschoben». Dadurch kann die **`ack_rate`** bzw. **`send_rate`** viel schneller erscheinen als sie tatsächlich ist. Da aber **`dt`** nicht kleiner als **`MinRTT`** sein kann, können diese Effekte herausgefiltert werden, indem **`dt >= MinRtt`** sein muss. **`Dt`** stellt hierbei die vergangene Zeit zwischen zwei empfangenen ACKs dar.

9. Evaluation: Delivery Rate Estimation

Um die «Delivery Rate Estimation» zu testen, wurde anfangs versucht, das Verhalten auf dem localhost zu prüfen. Hierbei stellte sich die Simulation der Latenz als grosses Problem heraus. Die Latenz wurde versuchsweise mit einer **`time.sleep()`** Methode bei jedem **`write()`** Aufruf simuliert. War beispielsweise eine Latenz von 20ms geplant, so musste man in der **`write()`**-Methode, welche die Daten auf die UDP-Connection schreibt, einen **`time.sleep(20ms)`** einbauen.

Aber auch bei dieser Implementation gab es noch Probleme. Bevor das Paket zum Schreiben übergeben wird, wird immer der Timestamp für das jeweilige Paket zum Zeitpunkt **`t=0`** registriert. Anschliessend schläft die Goroutine für 20ms, um den Delay zu simulieren. Danach erfolgt der erste **`write()`**. Beim nächsten **`write()`** wird der Timestamp wieder vor dem Schreiben erfasst. Er liegt bei **`t=0.02`**. In einer realen Verbindung sollte der nächste Timestamp aber ohne die **`time.sleep(20ms)`** gemessen werden.

Das bedeutet, dass beim Erfassen des Timestamps zum Sendezeitpunkt, vorher die Latenz vom **`time.sleep(20ms)`** miterfasst wurde. Daraus folgt, dass die Pakete viel langsamer im Abstand von 20ms versendet werden, obwohl sie in einem kleineren Abstand versendet werden könnten. Darunter leidet die Performance. Dasselbe Problem stellt sich, wenn der **`time.sleep(20ms)`** statt im **`write()`** im **`read()`** eingebaut wird.

Daher wurde beschlossen, das Delivery Estimate auf einer Cloud-Verbindung zu testen. Dies nicht nur aus dem Grund, dass sich die Latenz kaum simulieren lässt, sondern auch, weil man dadurch alle anderen Faktoren automatisch mitberücksichtigt und somit nichts Wichtiges vergessen geht. Für das Testing auf der Cloud lässt man die entwickelte Lösung einmal auf dem Notebook und einmal auf der Cloud laufen und verbindet sie miteinander.

In der Arbeit wurde erst gegen Ende erkannt, dass durch das Setzen von manuellen Timestamps der Delay simuliert werden könnte. Konkret würde bei jedem Senden des Pakets zusätzlich zur echten Timestamp noch ein bestimmter Zeitanteil (z.B. 20ms) dazu addiert werden. Das würde das Senden theoretisch um diese 20ms verzögern, wodurch ein Delay von 20ms für den Sendeweg simuliert wird.

Zusätzlich würden beim Erhalt des ACK-Segments dem Timestamp 10ms addiert werden. Dadurch wird dem Rückweg vom Empfänger zum Sender eine Delay von 10ms hinzugefügt.

9.1. Goroutine vs Java Threads

Wir verwenden im Folgenden die Begriffe Threads und Goroutines synonym, obwohl sie genau genommen unterschiedliche Konzepte haben. Wenn also ein Thread erwähnt wird, ist er mit der Goroutine gleichzusetzen.

Es ist wichtig, dass Goroutines, anders als bei Threads in Java, nicht zwei echt parallele Threads zugewiesen bekommen. Ihnen wird vom Scheduler von Go immer ein Zeit-Slot zur Verfügung gestellt, bei dem sie Rechenpower nutzen können. Sobald eine Goroutine ihren Zeit-Slot aufgebraucht hat, muss sie ihre Rechenpower einer anderen Goroutine freigegeben. Wenn die UDP-Connection ein Segment vom Peer erhalten hat, und das Scheduling dem write Thread die Rechenpower zur Verfügung stellt, kann der read Thread sein erhaltenes Segment nicht zum Zeitpunkt des Empfangs abarbeiten. Dies führt dazu, dass Timestamps verfälscht werden, wie später zu sehen sein wird.

9.2. Write und Read Funktionen in separaten Goroutines

Wie im Log-Auszug vom **Fehler! Verweisquelle konnte nicht gefunden werden.** zu sehen ist, wird zuerst jedes zu schreibende UDP Segment geschrieben und erst nach dem Schreiben vom letzten Segment beginnt das Lesen aller für die gesendeten Segmente erhaltenen ACK-Segmente.

Dies erklärt sich durch das fehlende **`time.sleep(1ns)`** im main-Thread. Golang wechselt nur seinen Thread, wenn es dazu forciert wird. Dadurch werden die Child-Goroutines von main (read und write) zum Laufen gebracht. Dies kann beispielsweise mit dem erwähnten **`time.sleep(1ns)`** umgesetzt werden. [23]

Konkret sähe das wie folgt aus:

```
for i := 0; i < writeNTimes; i++ {
   time.Sleep(1*time.Nanosecond)
   _, err = conn.Write(toWrite)
}
```

Bei **`time.sleep(1ns)`** handelt es sich um keine wirkliche Verbesserung. Es ist nur ein Workaround. Daher wäre es bei der Implementation sinnvoller gewesen, hier mit den Golang-Channels zu arbeiten. Dies war bei der Implementation allerdings nicht bekannt, weshalb trotzdem auf **`time.sleep(1ns)`** zurückgegriffen wurde. Im Verlauf der Kapitels wird daher trotzdem weiterhin auf die Variante mit **`time.sleep(1ns)`** eingegangen.

9.2.1. Workaround: time.sleep(1ns) in main Thread

Ergänzt man den Code durch das im vorherigen Kapitel erwähnte **`time.sleep(1ns)`** und lässt es gegen die Cloud laufen, ergibt sich der Log-Auszug, der in **Fehler! Verweisquelle konnte nicht gefunden werden.** zu sehen ist. Es zeigt sich ein verändertes Bild im Vergleich zum **Fehler! Verweisquelle konnte nicht gefunden werden.**. Nun werden wie gewünschte die Reads zwischen den writes gemacht. Jedoch passiert der Wechsel zwischen den Read und write Goroutines zufällig.

Es stellt sich heraus, dass durch den **`time.sleep(1ns)`** im main Thread zwar Wechsel erfolgen, jedoch aber nicht häufig genug. Dies kann man sich so vorstellen, dass durch das **`time.sleep(1ns)`** im main Thread jedem Thread Zeit-Slots zur Verfügung stehen, diese aber zu gross sind und kleiner gemacht werden müssen. Das geschieht mit einem **`time.sleep(1ns)`** innerhalb der **`write()`**-Funktion. Dadurch erfolgt der Wechsel zwischen Goroutines häufiger. Dazu ist die letzte Zeile als Anpassung im Code-Teil notwendig: `time.Sleep(1*time.Nanosecond)`

```
func (socket *PeerSocket) write() {
   for {
      socket.multipartLock.Lock()
      for _, c := range socket.multiplex {
         c.arq.retransmitTimedOutSegments(timeNow(), c.arq.bbr)
         _, err := c.arq.writeQueuedSegments(timeNow())
         if err != nil {
            socket.errorHandler(err)
         }
      }
      socket.multipartLock.Unlock()
      time.Sleep(1*time.Nanosecond)
   }
}
```

Im read Thread ist kein **`time.sleep(1ns)`** nötig, weil der read Thread durch das Lesen auf der UDP-Connection immer blockiert. Zudem bestand in unserem Fall das Problem, dass nicht ausreichend häufig auf den read Thread gewechselt wurde.

9.2.2. Workaround: time.sleep(1ns) in write Thread

Durch die Veränderung, die im vorherigen Kapitel vorgestellt wurde, zeigt sich ein deutlich verbessertes Bild in den Logs. Diese sind im **Fehler! Verweisquelle konnte nicht gefunden werden.** aufgeführt.

Jetzt wird viel flüssiger gelesen. Die gemessenen RTTs sind deutlich kleiner und reichen von 10-20ms. Die Lösung ist jedoch noch nicht optimal. Dies zeigt sich vor allem beim Receive von zwei Paketen (SEQ 3 und 4) zum gleichen Zeitpunkt (22:53:56.0966175). Das deutet darauf hin, dass SEQ 3 schon längere Zeit in der UDP-ReadConnection anwesend ist und somit früher hätte abgeholt werden können. Weitere Verbesserungen konnten im Rahmen dieser Arbeit leider nicht vorgenommen werden. Allerdings sollte durch ein besseres Verständnis des Memory Models in Golang möglich sein, die Delivery Rate Estimation weiter zu optimieren, beispielsweise durch die bereits erwähnten Golang-Channels.

9.3. Write und Read innerhalb einer Goroutine

Da im vorherigen Kapitel die **`read()`** und **`write()`** Funktionen in separaten Goroutines nicht zu den erwünschten Ergebnissen führten, wurde versucht die **`write()`** und **`read()`** Methode in einer Methode zu vereinen.

Die erforderlichen Änderungen sind in diesem Commit[4] ersichtlich. Die Funktionen konnten ohne grossen Aufwand vereint werden. Es waren lediglich Anpassungen an den **`for`**-Schleifen bei den jeweiligen Methoden erforderlich.

9.3.1. Testen auf dem localhost

Die Resultate aus diesem Testlauf sind im **Fehler! Verweisquelle konnte nicht gefunden werden.** ersichtlich. Hier erscheinen die RTTs meistens mit 0s. Es scheint, als könnten beide Peers auf dem localhost zur gleichen Zeit senden und empfangen. Das sieht man auch an der Sende- und Empfangszeit von beispielsweise Paket SEQ 1. Dieses wird um 14:53:30.180757 gesendet und zur genau selben Zeit empfangen. Selbstverständlich vergeht zwischen dem Senden von Peer 1 bis zum Empfang bei Peer 2 und das Senden des ACKs von Peer 2 nach Peer 1 eine bestimmte Zeit. Diese scheint jedoch so minimal zu sein, dass die **`time.Now()`** Funktion von Golang ihn nicht mitberücksichtigt.

Die **`time.Now()`** Funktion greift immer auf die Uhrzeit des OS zurück. Weil das Programm auf dem Windows OS getestet wurde, ist gemäss diesem Stackoverflow Post [24] «nur» eine Präzision von 100ns möglich. Würde es auf der Linux-Umgebung getestet werden, wäre eine Präzision im ns-Bereich zu erwarten.

[4] https://gitlab.com/p2p-library-in-golang/code/-/commit/ee9843e043c8d83614dfdb8208e286312b36c670

9.3.2. Testen auf der Cloud

Interessant ist auch die Beobachtung übers Netz. Um herauszufinden, wie lange die Latenz von unserem Developer Computer zur Cloud ist, wurde als erstes ein Ping gesendet. Daraus ergeben sich folgende RTTs:

```
Pinging 34.65.198.37 with 32 bytes of data:
Reply from 34.65.198.37: bytes=32 time=8ms TTL=119
Reply from 34.65.198.37: bytes=32 time=8ms TTL=119
Reply from 34.65.198.37: bytes=32 time=8ms TTL=119
Reply from 34.65.198.37: bytes=32 time=8ms TTL=119
Reply from 34.65.198.37: bytes=32 time=14ms TTL=119
Reply from 34.65.198.37: bytes=32 time=7ms TTL=119
Reply from 34.65.198.37: bytes=32 time=9ms TTL=119
Reply from 34.65.198.37: bytes=32 time=6ms TTL=119
Reply from 34.65.198.37: bytes=32 time=8ms TTL=119
Reply from 34.65.198.37: bytes=32 time=6ms TTL=119
Reply from 34.65.198.37: bytes=32 time=8ms TTL=119
Reply from 34.65.198.37: bytes=32 time=8ms TTL=119
Reply from 34.65.198.37: bytes=32 time=8ms TTL=119
Reply from 34.65.198.37: bytes=32 time=9ms TTL=119
Reply from 34.65.198.37: bytes=32 time=7ms TTL=119
Reply from 34.65.198.37: bytes=32 time=9ms TTL=119
Reply from 34.65.198.37: bytes=32 time=8ms TTL=119
Reply from 34.65.198.37: bytes=32 time=9ms TTL=119
Reply from 34.65.198.37: bytes=32 time=10ms TTL=119
Reply from 34.65.198.37: bytes=32 time=7ms TTL=119
```

Im Schnitt kann also von einem RTT von 8ms ausgegangen werden. Wie im Auszug von **Fehler! Verweisquelle konnte nicht gefunden werden.** zu sehen ist, wird die RTT zu Beginn mit 10ms angezeigt. Selten tauchen auch RTTs von 20ms auf. Allerdings treten keine RTTs auf, die im Bereich deutlich grösser als 10ms und deutlich kleiner als 20ms, also beispielsweise 16ms lang sind.

Zudem ist ersichtlich, dass bei RTTs von 20ms zwei aufeinanderfolgende writes vorangegangen sind.

Das Problem liegt in der eigentlichen Implementation. Da der write und read jetzt in einem gemeinsamen Thread sind, und das im 5.2 erklärte Pacing zur Anwendung kommt, werden die Pakete nur im Abstand von 10ms empfangen.

Konkret sendet der Peer sein erstes Paket und versucht dabei sogleich das ACK zu empfangen. Das ACK ist aber noch nicht da, weil das versendete Packet noch auf dem Weg zum anderen Peer ist oder das ACK noch nicht beim Sender angekommen ist, sich also «in-flight» befindet. Daher versucht der Peer ein weiteres Paket zu senden. Aufgrund des Pacings muss er vorerst aber 10ms warten. Während des Wartens wird das ACK für das erste Paket bereits angekommen sein. Der Peer liest das ACK und vermerkt den Timestamp zur Berechnung des RTT erst nachdem er das zweite Paket versendet hat. Dadurch verzögert sich der berechnete RTT um mindestens 10ms. Deshalb zeigen die Resultate in **Fehler! Verweisquelle konnte nicht gefunden werden.** immer RTTs von mindestens 10ms.

Die zwei aufeinanderfolgenden writes mit 20ms RTT lassen sich mit derselben Logik erklären. Wie in den Pings ersichtlich ist, dauert eine RTT auch ausnahmsweise 14ms. In diesem Fall gibt es ein zusätzliches write, womit sich wegen des Pacings die RTT auf 20ms erhöht.

9.3.3. Ansatz: Lesen während dem Warten auf das Schreiben

Während dem Warten auf den nächsten Sendezeitpunkt, könnte mit der letzten Zeile Code im untenstehenden Beispiel gelesen werden (**`peer.read()`**). Dieser Ansatz hat während der Entwicklung teilweise funktioniert und die Zeitmessungen konnten erfolgreich durchgeführt werden. Aus Zeitgründen gegen Ende der Arbeit konnte dieser Ansatz leider nicht weiter verfolgt werden.

Diese Änderung hätte weitreichende Folgen bis hin zu notwendigen Anpassungen in der Architektur gehabt. Zudem konnten Synchronisationsprobleme zwischen main-Thread und dem zusammengefassten write/read Thread festgestellt werden. Denn das **`read()`** greift wie auch der main-Thread über **`connection.Read()`** auf den **`readBuffer`** der Connection zu. Dies führte zur Race Condition, die auf die Schnelle nicht behoben werden konnten.

```
//timeToWait:= seg.size()/arq.bbr.BBRGetPacingRate * 1000000000 //* 1000000000 =>
to Nanoseconds
timeToWait := arq.timeToWait
nextSendtime := arq.bbr.last_time_send.Add(time.Duration(timeToWait))
   for {
      if !time.Now().Before(nextSendtime) {
      break
      }
      peer.read()
   }
```

Summary / Conclusion

10. Zielerreichung

Fast alle vorgeben Ziele der Arbeit konnten erreicht werden. Allerdings gab es Probleme, welche die vollständige Entwicklung der BBR Congestion Control nicht möglich machten. Gründe dafür sind vielfältig. Zum einen war die Komplexität des Algorithmus nicht greifbar. Sie fordert neben der eigentlichen Logik auch einen tieferen mathematischen Hintergrund. Dieser mathematische Hintergrund war für uns nicht klar nachvollziehbar und bedarf einer besseren Erklärung. Des Weiteren liess der RFC die Detail-Implementation offen, was die Implementation deutlich erschwerte. Dies führte dazu, dass zuerst ein umfassendes Verständnis für den Algorithmus aufgebaut werden musste, was wegen schwierig nachvollziehbarer Mathematik eine Herausforderung war.

Ein weiteres Problem stellte die Zeitmessung für die Implementation der «Delivery Rate Estimation» dar. Dies vor allem wegen der Programmiersprache Golang und dessen Memory Model. Für eine ausreichend gute Abschätzung der Sendegeschwindigkeit ist ein besseres Verständnis des Memory Models wichtig. Das konnte im Rahmen dieser Arbeit nicht hinreichend aufgebaut werden.

Der anschliessende Wechsel auf die sequenzielle Ausführung hat die Zeitmessung deutlich verbessert, allerdings sind die Ergebnisse noch immer nicht zufriedenstellend. Der nächste im 9.3.3 vorgestellte Ansatz wäre für Fortsetzungsarbeiten interessant.

11. Ergebnis

In dieser Arbeit konnte das ARQ als Kern der von TomP2P benötigten Library erfolgreich implementieren werden. Auch die theoretischen Grundlagen für den BBR konnten weitreichend erarbeitet werden. Als Anfang einer BBR Implementation wurde die «Delivery Rate Estimation» implementiert und getestet. Die hier aufgetretenen Herausforderungen der genauen Zeitmessung und die Ansätze zur Optimierung konnten im Evaluation: Delivery Rate Estimation ausführlich aufgezeigt werden.

In einer Folgearbeit kann die «Delivery Rate Estimation» weiter verbessert und das BBR mithilfe der hier erarbeiteten Theorie implementiert werden. Die in dieser Arbeit aufgebaute Testumgebung erlaubt zudem das schnelle Realisieren von ganzheitlichen Tests für mögliche Fortsetzungsarbeiten.

Verzeichnisse

12. Glossar

ATP-go ist eine Implementation eines Protokolls in Golang, die in einer früheren Studien- und Bachelorarbeit umgesetzt wurde.

Blake2b ist ein Message Authentication Code (MAC), mit welchem die Integrität von versendeten Nachrichten sichergestellt wird. Dabei steht das «b» von Blake2b für die optimierte Version für 64-bit Plattformen.

ChaCha20 - Poly1305 ist ein kryptographisches Verfahren. Dabei wird Chacha20 für die Verschlüsselung und Poly1305 als MAC für die Integrität versendeter Nachrichten verwendet.

Congestion Control bedeutet auf Deutsch Flusskontrolle. In dieser Arbeit tritt der Begriff vor allem im Zusammenhang mit dem Sender auf. Der Sender versucht durch die Congestion Control dem Empfänger eine optimale Anzahl an Paketen über das Internet zu senden, ohne dass dabei durch zu viel auf einmal versendete Pakete, Paketverluste entstehen.

Curve25519 gilt als schnelles Verfahren zum Austausch von symmetrischen Schlüsseln**.**

GCloud ist die Google Cloud, die in dieser Arbeit für die Simulation der Latenz während des Testings der Delivery Estimation zum Einsatz kam.

Goland ist die Entwicklungsumgebung, in der diese Arbeit implementiert wurde.

Golang ist eine kompilierbare Programmiersprache, welche die Nebenläufigkeit direkt integriert hat und sich syntaktisch stark an C orientiert. Sie wurde für die Implementation des in dieser Arbeit entwickelten Protokolls eingesetzt.

in-flight bezeichnet Daten die gesendet, aber noch nicht bestätigt wurden.

Multiplex wird in dieser Arbeit vor allem im Zusammenhang mit der implementierten Lösung verwendet. Bei den vom Peer eingehenden Connections handelt es sich jeweils um multiplex Verbindungen. Somit können über eine Connection mehrere Pakete übertragen werden.

Noise-Framework ist ein Open-Source-Framework das für Verschlüsselungsprotokolle mit Ende-zu-Ende-verschlüsseltem Datenaustausch eingesetzt werden kann.

Perfect Forward Secrecy bezeichnet einen Schlüsselaustausch, bei dem verhindert wird, dass die Kommunikation entschlüsselt werden kann.

Sliding-Window beschreibt die Menge von Paketen, die der Sender senden kann, bevor eine Bestätigung als ACK zurückerwartet wird.

TomP2P ist eine Peer to Peer (P2P) Library, die eine Distributed Hash Table (DHT) für verteilte Anwendungen bereitstellt.

Window Size bezeichnet die maximale Anzahl Pakete, die der Empfänger bei ARQ empfangen kann, ohne die Pakete per ACK bestätigen zu müssen.

13. Akronyme

AES	Advanced Encryption Standard
ARQ	Automatic Repeat reQuest
BBR	Bottleneck Bandwidth and Round-trip propagation time
CCA	Congestion Control Algorithm
CI	Continuous Integration
DHT	Distributed Hash Table
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
MAC	Message Authentication Code
MSS	Maximum Segment Size
NIST	National Institute of Standards and Technology
P2P	Peer-to-Peer
RPC	Remote Procedure Call
RTO	Retransmission Time Out
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
UDP	User Datagram Protocol

14. Quellenverzeichnis

[1] T. Bocek, «TomP2P - Github,» [Online]. Available: https://github.com/tomp2p/TomP2P.

[2] IONOS, «Golang: Die einfache Programmiersprache aus dem Hause Google,» [Online]. Available: https://www.ionos.de/digitalguide/server/knowhow/golang/.

[3] BMU Verlag, «Die Programmiersprache Go,» [Online]. Available: https://bmu-verlag.de/die-programmiersprache-go/.

[4] A. C. a. S. M. Matija Čupić, «Standard Go Template - Gitlab,» [Online]. Available: https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Go.gitlab-ci.yml.

[5] R. Gane, «Gitlab CI for Go projects,» [Online]. Available: https://ronnie-gane.kiwi/blog/2019/06/18/go-gitlab/.

[6] Gitlab, «Unit test reports,» [Online]. Available: https://docs.gitlab.com/ee/ci/unit_test_reports.html.

[7] T. Bocek, «ATP-GO - Github,» [Online]. Available: https://github.com/tbocek/atp-go.

[8] skywind3000, «KCP-go - Gitlab,» [Online]. Available: https://github.com/skywind3000/kcp.

[9] Golem, «Quic ist offizieller Internet-Standard,» [Online]. Available: https://www.golem.de/news/ietf-quic-ist-offizieller-internet-standard-2105-156853.html.

[10] S. Ahmadi, «Automatic Repeat Request - ScienceDirect,» [Online]. Available: https://www.sciencedirect.com/topics/computer-science/automatic-repeat-request.

[11] P. Heinzmann, «Vorlesung: TCP (8.5.1 - 8.5.7)».

[12] Neso Academy, «Stop-and-Wait ARQ Protocol,» [Online]. Available: https://www.youtube.com/watch?v=YdkksvhkQGQ.

[13] Neso Academy, «Go-Back-N ARQ,» [Online]. Available: https://www.youtube.com/watch?v=QD3oCelHJ20.

[14] RWTH Aachen - Institute for Theoretical Information Technology, «Sicherungsschicht- ARQ,» [Online]. Available: https://www.ti.rwth-aachen.de/teaching/kommunikationsnetze/data/Folien_28.10.2009.pdf.

[15] Neso Academy, «Selective Repeat ARQ,» [Online]. Available: https://www.youtube.com/watch?v=WfIhQ3o2xow.

[16] Y. C. C. S. G. S. H. Y. V. J. Neal Cardwell, «BBR: Congestion-Based Congestion Control,» [Online]. Available: https://queue.acm.org/detail.cfm?id=3022184.

[17] N. Cardwell, «BBR Congestion Control: IETF 99 Update,» [Online]. Available: https://datatracker.ietf.org/meeting/99/materials/slides-99-iccrg-iccrg-presentation-2-00.pdf.

[18] S. S. Kumar, «How TCP segment size can affect application traffic flow,» [Online]. Available: https://medium.com/walmartglobaltech/how-tcp-segment-size-can-affect-application-traffic-flow-7bbceed5816e.

[19] Y. C. S. H. Y. V. J. N. Cardwell, «BBR Congestion Control,» [Online]. Available: https://tools.ietf.org/id/draft-cardwell-iccrg-bbr-congestion-control-00.html#detailed-algorithm.

[20] T. Perrin, «The Noise Protocol Framework,» [Online]. Available: https://noiseprotocol.org/noise.html.

[21] Flynn, «noise,» [Online]. Available: https://github.com/flynn/noise.

[22] N. C. S. H. Y. V. J. Y. Cheng, «Delivery Rate Estimation -Per-connection (C) state,» [Online]. Available: https://datatracker.ietf.org/doc/html/draft-cheng-iccrg-delivery-rate-estimation-00#section-3.1.1.

[23] U. Hiwarale, «Achieving concurrency in Go,» [Online]. Available: https://medium.com/rungo/achieving-concurrency-in-go-3f84cbf870ca.

[24] Arno, «How to use GetSystemTimeAdjustment correctly? - Stackoverflow,» [Online]. Available: https://stackoverflow.com/questions/7685762/windows-7-timing-functions-how-to-use-getsystemtimeadjustment-correctly/11743614#11743614.

15. Abbildungsverzeichnis

Anhang

A. Log-Auszug: Write und Read Funktionen

A.1 Read() und write() in separaten Goroutines

```
write, SEQNR: 0 time: 15:54:26.0672976
write, SEQNR: 1 time: 15:54:26.0782537
write, SEQNR: 2 time: 15:54:26.0892519
write, SEQNR: 3 time: 15:54:26.100252
write, SEQNR: 4 time: 15:54:26.111248
write, SEQNR: 5 time: 15:54:26.1215657
write, SEQNR: 6 time: 15:54:26.1323342
write, SEQNR: 7 time: 15:54:26.1423412
write, SEQNR: 8 time: 15:54:26.1527438
write, SEQNR: 9 time: 15:54:26.1637445
write, SEQNR: 10 time: 15:54:26.1747384
write, SEQNR: 11 time: 15:54:26.18474
write, SEQNR: 12 time: 15:54:26.194756
write, SEQNR: 13 time: 15:54:26.2057397
write, SEQNR: 14 time: 15:54:26.2167386
write, SEQNR: 15 time: 15:54:26.227738
write, SEQNR: 16 time: 15:54:26.2377399
write, SEQNR: 17 time: 15:54:26.2487377
write, SEQNR: 18 time: 15:54:26.2587417
write, SEQNR: 19 time: 15:54:26.2697468
write, SEQNR: 20 time: 15:54:26.2807414
write, SEQNR: 21 time: 15:54:26.29174
write, SEQNR: 22 time: 15:54:26.30174
write, SEQNR: 23 time: 15:54:26.3117766
write, SEQNR: 24 time: 15:54:26.3227461
write, SEQNR: 25 time: 15:54:26.3327738
write, SEQNR: 26 time: 15:54:26.3427749
write, SEQNR: 27 time: 15:54:26.3537428
write, SEQNR: 28 time: 15:54:26.3637735
write, SEQNR: 29 time: 15:54:26.3740201
write, SEQNR: 30 time: 15:54:26.3842371
receive, SEQNR: 0 time: 15:54:27.1183845
RTT: 1.0510869s
receive, SEQNR: 1 time: 15:54:27.1183845
RTT: 1.0401308s
receive, SEQNR: 2 time: 15:54:27.1183845
RTT: 1.0291326s
receive, SEQNR: 3 time: 15:54:27.1183845
RTT: 1.0181325s
receive, SEQNR: 4 time: 15:54:27.1183845
RTT: 1.0071365s
receive, SEQNR: 5 time: 15:54:27.1183845
RTT: 996.8188ms
receive, SEQNR: 6 time: 15:54:27.1183845
RTT: 986.0503ms
receive, SEQNR: 7 time: 15:54:27.1183845
RTT: 976.0433ms
receive, SEQNR: 8 time: 15:54:27.1183845
RTT: 965.6407ms
receive, SEQNR: 9 time: 15:54:27.1183845
RTT: 954.64ms
receive, SEQNR: 10 time: 15:54:27.1183845
RTT: 943.6461ms
receive, SEQNR: 11 time: 15:54:27.1183845
RTT: 933.6445ms
receive, SEQNR: 12 time: 15:54:27.1183845
RTT: 923.6285ms
receive, SEQNR: 13 time: 15:54:27.1183845
RTT: 912.6448ms
receive, SEQNR: 14 time: 15:54:27.1183845
RTT: 901.6459ms
receive, SEQNR: 15 time: 15:54:27.1183845
RTT: 890.6465ms
receive, SEQNR: 16 time: 15:54:27.1183845
RTT: 880.6446ms
receive, SEQNR: 17 time: 15:54:27.1183845
RTT: 869.6468ms
receive, SEQNR: 18 time: 15:54:27.1183845
RTT: 859.6428ms
receive, SEQNR: 19 time: 15:54:27.1183845
RTT: 848.6377ms
receive, SEQNR: 20 time: 15:54:27.1183845
RTT: 837.6431ms
receive, SEQNR: 21 time: 15:54:27.1183845
RTT: 826.6445ms
receive, SEQNR: 22 time: 15:54:27.1183845
RTT: 816.6445ms
receive, SEQNR: 23 time: 15:54:27.1183845
RTT: 806.6079ms
receive, SEQNR: 24 time: 15:54:27.1183845
RTT: 795.6384ms
receive, SEQNR: 25 time: 15:54:27.1183845
RTT: 785.6107ms
receive, SEQNR: 26 time: 15:54:27.1183845
RTT: 775.6096ms
receive, SEQNR: 27 time: 15:54:27.1183845
RTT: 764.6417ms
receive, SEQNR: 28 time: 15:54:27.1183845
RTT: 754.611ms
receive, SEQNR: 29 time: 15:54:27.1183845
RTT: 744.3644ms
receive, SEQNR: 30 time: 15:54:27.1183845
RTT: 734.1474ms
```

A.2 Read() und write() in separaten Goroutines time.sleep im main Thread

write, SEQNR: 0 time: 23:00:11.0310617
write, SEQNR: 1 time: 23:00:11.0411042
write, SEQNR: 2 time: 23:00:11.0520576
receive, SEQNR: 0 time: 23:00:11.0520576
RTT: 20.9959ms
write, SEQNR: 3 time: 23:00:11.0620577
write, SEQNR: 4 time: 23:00:11.0720699
receive, SEQNR: 1 time: 23:00:11.0720699
RTT: 30.9657ms
write, SEQNR: 5 time: 23:00:11.0830611
write, SEQNR: 6 time: 23:00:11.0930654
receive, SEQNR: 2 time: 23:00:11.0930654
RTT: 41.0078ms
write, SEQNR: 7 time: 23:00:11.103577
write, SEQNR: 8 time: 23:00:11.1145751
receive, SEQNR: 3 time: 23:00:11.1145751
RTT: 52.5174ms
write, SEQNR: 9 time: 23:00:11.124576
write, SEQNR: 10 time: 23:00:11.1345766
receive, SEQNR: 4 time: 23:00:11.1345766
RTT: 62.5067ms
write, SEQNR: 11 time: 23:00:11.1455755
write, SEQNR: 12 time: 23:00:11.1565746
receive, SEQNR: 5 time: 23:00:11.1565746
RTT: 73.5135ms
write, SEQNR: 13 time: 23:00:11.1665756
write, SEQNR: 14 time: 23:00:11.1765761
receive, SEQNR: 6 time: 23:00:11.1765761
RTT: 83.5107ms
write, SEQNR: 15 time: 23:00:11.1865786
write, SEQNR: 16 time: 23:00:11.1967264
receive, SEQNR: 7 time: 23:00:11.1967264
RTT: 93.1494ms
write, SEQNR: 17 time: 23:00:11.2070504
write, SEQNR: 18 time: 23:00:11.217779
receive, SEQNR: 8 time: 23:00:11.217779 RTT:
103.2039ms
write, SEQNR: 19 time: 23:00:11.2277829
write, SEQNR: 20 time: 23:00:11.2377856
receive, SEQNR: 9 time: 23:00:11.2377856
RTT: 113.2096ms
write, SEQNR: 21 time: 23:00:11.2487791
write, SEQNR: 22 time: 23:00:11.2587815
receive, SEQNR: 10 time: 23:00:11.2587815
RTT: 124.2049ms
write, SEQNR: 23 time: 23:00:11.2697928
write, SEQNR: 24 time: 23:00:11.2807791
receive, SEQNR: 11 time: 23:00:11.2807791
RTT: 135.2036ms
write, SEQNR: 25 time: 23:00:11.2907832
write, SEQNR: 26 time: 23:00:11.3017799
receive, SEQNR: 12 time: 23:00:11.3017799
RTT: 145.2053ms
write, SEQNR: 27 time: 23:00:11.3117812
write, SEQNR: 28 time: 23:00:11.3217847
receive, SEQNR: 13 time: 23:00:11.3217847
RTT: 155.2091ms
write, SEQNR: 29 time: 23:00:11.3327796
write, SEQNR: 30 time: 23:00:11.3427805
receive, SEQNR: 14 time: 23:00:11.3427805
RTT: 166.2044ms
receive, SEQNR: 15 time: 23:00:11.352781
RTT: 166.2024ms
receive, SEQNR: 16 time: 23:00:11.3737806
RTT: 177.0542ms
receive, SEQNR: 17 time: 23:00:11.3957817
RTT: 188.7313ms
receive, SEQNR: 18 time: 23:00:11.4177819
RTT: 200.0029ms
receive, SEQNR: 19 time: 23:00:11.4383952
RTT: 210.6123ms
receive, SEQNR: 20 time: 23:00:11.4593933
RTT: 221.6077ms
receive, SEQNR: 21 time: 23:00:11.4593933
RTT: 210.6142ms
receive, SEQNR: 22 time: 23:00:11.4803941
RTT: 221.6126ms
receive, SEQNR: 23 time: 23:00:11.5013942
RTT: 231.6014ms
receive, SEQNR: 24 time: 23:00:11.5233946
RTT: 242.6155ms
receive, SEQNR: 25 time: 23:00:11.5453938
RTT: 254.6106ms
receive, SEQNR: 26 time: 23:00:11.5663935
RTT: 264.6136ms
receive, SEQNR: 27 time: 23:00:11.5883906
RTT: 276.6094ms
receive, SEQNR: 28 time: 23:00:11.6083926
RTT: 286.6079ms
receive, SEQNR: 29 time: 23:00:11.6303904
RTT: 297.6108ms
receive, SEQNR: 30 time: 23:00:11.6513923
RTT: 308.6118ms

A.3 Read() und write() in separaten Goroutines mit time.sleep im write Thread

```
write, SEQNR: 0 time: 22:53:56.0455773
write, SEQNR: 1 time: 22:53:56.0555777
receive, SEQNR: 0 time: 22:53:56.0555777
RTT: 10.0004ms
write, SEQNR: 2 time: 22:53:56.0656203
write, SEQNR: 3 time: 22:53:56.0765776
receive, SEQNR: 1 time: 22:53:56.0765776
RTT: 20.9999ms
receive, SEQNR: 2 time: 22:53:56.0765776
RTT: 10.9573ms
write, SEQNR: 4 time: 22:53:56.0865788
write, SEQNR: 5 time: 22:53:56.0966175
receive, SEQNR: 3 time: 22:53:56.0966175
RTT: 20.0399ms
receive, SEQNR: 4 time: 22:53:56.0966175
RTT: 10.0387ms
write, SEQNR: 6 time: 22:53:56.1066206
receive, SEQNR: 5 time: 22:53:56.1066206
RTT: 10.0031ms
write, SEQNR: 7 time: 22:53:56.1166426
receive, SEQNR: 6 time: 22:53:56.1166426
RTT: 10.022ms
write, SEQNR: 8 time: 22:53:56.1275762
write, SEQNR: 9 time: 22:53:56.1376225
receive, SEQNR: 7 time: 22:53:56.1376225
RTT: 20.9799ms
receive, SEQNR: 8 time: 22:53:56.1376225
RTT: 10.0463ms
write, SEQNR: 10 time: 22:53:56.1480983
write, SEQNR: 11 time: 22:53:56.1590959
receive, SEQNR: 9 time: 22:53:56.1590959
RTT: 21.4734ms
receive, SEQNR: 10 time: 22:53:56.1590959
RTT: 10.9976ms
write, SEQNR: 12 time: 22:53:56.1690972
write, SEQNR: 13 time: 22:53:56.1790974
receive, SEQNR: 11 time: 22:53:56.1790974
RTT: 20.0015ms
receive, SEQNR: 12 time: 22:53:56.1790974
RTT: 10.0002ms
write, SEQNR: 14 time: 22:53:56.1900953
receive, SEQNR: 13 time: 22:53:56.1900953
RTT: 10.9979ms
write, SEQNR: 15 time: 22:53:56.200102
receive, SEQNR: 14 time: 22:53:56.200102
RTT: 10.0067ms
write, SEQNR: 16 time: 22:53:56.2111007
write, SEQNR: 17 time: 22:53:56.2211571
receive, SEQNR: 15 time: 22:53:56.2211571
RTT: 21.0551ms
receive, SEQNR: 16 time: 22:53:56.2211571
RTT: 10.0564ms
write, SEQNR: 18 time: 22:53:56.2320949
receive, SEQNR: 17 time: 22:53:56.2320949
RTT: 10.9378ms
write, SEQNR: 19 time: 22:53:56.2421027
write, SEQNR: 20 time: 22:53:56.2522089
receive, SEQNR: 18 time: 22:53:56.2522089
RTT: 20.114ms
receive, SEQNR: 19 time: 22:53:56.2522089
RTT: 10.1062ms
write, SEQNR: 21 time: 22:53:56.2622144
receive, SEQNR: 20 time: 22:53:56.2622144
RTT: 10.0055ms
write, SEQNR: 22 time: 22:53:56.272215
write, SEQNR: 23 time: 22:53:56.2832697
receive, SEQNR: 21 time: 22:53:56.2832697
RTT: 21.0553ms
receive, SEQNR: 22 time: 22:53:56.2832697
RTT: 11.0547ms
write, SEQNR: 24 time: 22:53:56.294214
write, SEQNR: 25 time: 22:53:56.304311
receive, SEQNR: 23 time: 22:53:56.304311
RTT: 21.0413ms
receive, SEQNR: 24 time: 22:53:56.304311
RTT: 10.097ms
write, SEQNR: 26 time: 22:53:56.3152179
receive, SEQNR: 25 time: 22:53:56.3152179
RTT: 10.9069ms
write, SEQNR: 27 time: 22:53:56.326224
write, SEQNR: 28 time: 22:53:56.3373747
receive, SEQNR: 26 time: 22:53:56.3373747
RTT: 22.1568ms
receive, SEQNR: 27 time: 22:53:56.3373747
RTT: 11.1507ms
write, SEQNR: 29 time: 22:53:56.3476574
receive, SEQNR: 28 time: 22:53:56.3476574
RTT: 10.2827ms
write, SEQNR: 30 time: 22:53:56.3583552
receive, SEQNR: 29 time: 22:53:56.3583552
RTT: 10.6978ms
receive, SEQNR: 30 time: 22:53:56.3693515
```

A.4 Read() und write() in einer Goroutine auf dem localhost

write, SEQNR: 0 time: 14:53:30.1704065
receive, SEQNR: 0 time: 14:53:30.1734423
RTT: 3.0358ms
write, SEQNR: 1 time: 14:53:30.180757
receive, SEQNR: 1 time: 14:53:30.180757 RTT:
0s
write, SEQNR: 2 time: 14:53:30.1909616
receive, SEQNR: 2 time: 14:53:30.1909616
RTT: 0s
write, SEQNR: 3 time: 14:53:30.2009631
receive, SEQNR: 3 time: 14:53:30.2009631
RTT: 0s
write, SEQNR: 4 time: 14:53:30.2109635
receive, SEQNR: 4 time: 14:53:30.2109635
RTT: 0s
write, SEQNR: 5 time: 14:53:30.2219583
receive, SEQNR: 5 time: 14:53:30.2219583
RTT: 0s
write, SEQNR: 6 time: 14:53:30.2329581
receive, SEQNR: 6 time: 14:53:30.2329581
RTT: 0s
write, SEQNR: 7 time: 14:53:30.2429609
receive, SEQNR: 7 time: 14:53:30.2429609
RTT: 0s
write, SEQNR: 8 time: 14:53:30.2539591
receive, SEQNR: 8 time: 14:53:30.2539591
RTT: 0s
write, SEQNR: 9 time: 14:53:30.2639617
receive, SEQNR: 9 time: 14:53:30.2639617
RTT: 0s
write, SEQNR: 10 time: 14:53:30.2749585
receive, SEQNR: 10 time: 14:53:30.2749585
RTT: 0s
write, SEQNR: 11 time: 14:53:30.2849587
receive, SEQNR: 11 time: 14:53:30.2849587
RTT: 0s
write, SEQNR: 12 time: 14:53:30.2959582
receive, SEQNR: 12 time: 14:53:30.2959582
RTT: 0s
write, SEQNR: 13 time: 14:53:30.3059603
receive, SEQNR: 13 time: 14:53:30.3059603
RTT: 0s
write, SEQNR: 14 time: 14:53:30.3159614
receive, SEQNR: 14 time: 14:53:30.3159614
RTT: 0s
write, SEQNR: 15 time: 14:53:30.3269613
receive, SEQNR: 15 time: 14:53:30.3269613
RTT: 0s
write, SEQNR: 16 time: 14:53:30.3371446
receive, SEQNR: 16 time: 14:53:30.3371446
RTT: 0s
write, SEQNR: 17 time: 14:53:30.3481055
receive, SEQNR: 17 time: 14:53:30.3481055
RTT: 0s
write, SEQNR: 18 time: 14:53:30.3591051
receive, SEQNR: 18 time: 14:53:30.3591051
RTT: 0s
write, SEQNR: 19 time: 14:53:30.370103
receive, SEQNR: 19 time: 14:53:30.370103
RTT: 0s
write, SEQNR: 20 time: 14:53:30.3811027
receive, SEQNR: 20 time: 14:53:30.3811027
RTT: 0s
write, SEQNR: 21 time: 14:53:30.3911028
receive, SEQNR: 21 time: 14:53:30.3911028
RTT: 0s
write, SEQNR: 22 time: 14:53:30.401107
receive, SEQNR: 22 time: 14:53:30.401107
RTT: 0s
write, SEQNR: 23 time: 14:53:30.412105
receive, SEQNR: 23 time: 14:53:30.412105
RTT: 0s
write, SEQNR: 24 time: 14:53:30.4231042
receive, SEQNR: 24 time: 14:53:30.4231042
RTT: 0s
write, SEQNR: 25 time: 14:53:30.4331042
receive, SEQNR: 25 time: 14:53:30.4331042
RTT: 0s
write, SEQNR: 26 time: 14:53:30.4431103
receive, SEQNR: 26 time: 14:53:30.4431103
RTT: 0s
write, SEQNR: 27 time: 14:53:30.4541035
receive, SEQNR: 27 time: 14:53:30.4541035
RTT: 0s
write, SEQNR: 28 time: 14:53:30.4646314
receive, SEQNR: 28 time: 14:53:30.4646314
RTT: 0s
write, SEQNR: 29 time: 14:53:30.4746343
receive, SEQNR: 29 time: 14:53:30.4746343
RTT: 0s
write, SEQNR: 30 time: 14:53:30.4856331
receive, SEQNR: 30 time: 14:53:30.4856331
RTT: 0s

A.5 Read() und write() in einer Goroutine auf dem Internet

write, SEQNR: 0 time: 20:47:18.4665587
write, SEQNR: 1 time: 20:47:18.4769375
write, SEQNR: 2 time: 20:47:18.4869635
receive, SEQNR: 0 time: 20:47:18.4869635
RTT: 20.4048ms
receive, SEQNR: 1 time: 20:47:18.4869635
RTT: 10.026ms
write, SEQNR: 3 time: 20:47:18.4970077
receive, SEQNR: 2 time: 20:47:18.4970077
RTT: 10.0442ms
write, SEQNR: 4 time: 20:47:18.5070084
receive, SEQNR: 3 time: 20:47:18.5070084
RTT: 10.0007ms
write, SEQNR: 5 time: 20:47:18.5180082
receive, SEQNR: 4 time: 20:47:18.5180082
RTT: 10.9998ms
write, SEQNR: 6 time: 20:47:18.5280082
receive, SEQNR: 5 time: 20:47:18.5280082
RTT: 10ms
write, SEQNR: 7 time: 20:47:18.5390163
receive, SEQNR: 6 time: 20:47:18.5390163
RTT: 11.0081ms
write, SEQNR: 8 time: 20:47:18.5494202
receive, SEQNR: 7 time: 20:47:18.5494202
RTT: 10.4039ms
write, SEQNR: 9 time: 20:47:18.5604315
receive, SEQNR: 8 time: 20:47:18.5604315
RTT: 11.0113ms
write, SEQNR: 10 time: 20:47:18.5714435
receive, SEQNR: 9 time: 20:47:18.5714435
RTT: 11.012ms
write, SEQNR: 11 time: 20:47:18.582389
receive, SEQNR: 10 time: 20:47:18.582389
RTT: 10.9455ms
write, SEQNR: 12 time: 20:47:18.5925401
receive, SEQNR: 11 time: 20:47:18.5925401
RTT: 10.1511ms
write, SEQNR: 13 time: 20:47:18.60259
receive, SEQNR: 12 time: 20:47:18.60259 RTT: 10.0499ms
write, SEQNR: 14 time: 20:47:18.6129758
receive, SEQNR: 13 time: 20:47:18.6129758
RTT: 10.3858ms
write, SEQNR: 15 time: 20:47:18.6239725
receive, SEQNR: 14 time: 20:47:18.6239725
RTT: 10.9967ms

write, SEQNR: 16 time: 20:47:18.6339766
receive, SEQNR: 15 time: 20:47:18.6339766
RTT: 10.0041ms
write, SEQNR: 17 time: 20:47:18.6444985
receive, SEQNR: 16 time: 20:47:18.6444985
RTT: 10.5219ms
write, SEQNR: 18 time: 20:47:18.6554858
receive, SEQNR: 17 time: 20:47:18.6554858
RTT: 10.9873ms
write, SEQNR: 19 time: 20:47:18.6654872
receive, SEQNR: 18 time: 20:47:18.6654872
RTT: 10.0014ms
write, SEQNR: 20 time: 20:47:18.6754888
receive, SEQNR: 19 time: 20:47:18.6754888
RTT: 10.0016ms
write, SEQNR: 21 time: 20:47:18.6865178
receive, SEQNR: 20 time: 20:47:18.6865178
RTT: 11.029ms
write, SEQNR: 22 time: 20:47:18.6974895
receive, SEQNR: 21 time: 20:47:18.6974895
RTT: 10.9717ms
write, SEQNR: 23 time: 20:47:18.7084866
receive, SEQNR: 22 time: 20:47:18.7084866
RTT: 10.9971ms
write, SEQNR: 24 time: 20:47:18.7184882
receive, SEQNR: 23 time: 20:47:18.7184882
RTT: 10.0016ms
write, SEQNR: 25 time: 20:47:18.7284917
receive, SEQNR: 24 time: 20:47:18.7284917
RTT: 10.0035ms
write, SEQNR: 26 time: 20:47:18.7394867
receive, SEQNR: 25 time: 20:47:18.7394867
RTT: 10.995ms
write, SEQNR: 27 time: 20:47:18.7494871
receive, SEQNR: 26 time: 20:47:18.7494871
RTT: 10.0004ms
write, SEQNR: 28 time: 20:47:18.7594877
receive, SEQNR: 27 time: 20:47:18.7594877
RTT: 10.0006ms
write, SEQNR: 29 time: 20:47:18.7694883
receive, SEQNR: 28 time: 20:47:18.7694883
RTT: 10.0006ms
write, SEQNR: 30 time: 20:47:18.7795324
receive, SEQNR: 29 time: 20:47:18.7795324
RTT: 10.0441ms
receive, SEQNR: 30 time: 20:47:18.7895356
RTT: 1.0032ms