Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Denis Lugowski

Evaluation des Congestion-Control-Algorithmus CMT/RPv2 in einer MPTCP-Umgebung im Linux Kernel

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Denis Lugowski

Evaluation des Congestion-Control-Algorithmus CMT/RPv2 in einer MPTCP-Umgebung im Linux Kernel

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr.-Ing. Martin Hübner

Eingereicht am: 05. Januar 2017

Denis Lugowski

Thema der Arbeit
Evaluation des Congestion-Control-Algorithmus CMT/RPv2 in einer MPTCP-Umgebung im Linux Kernel

Stichworte
Congestion Control, Multipath TCP, MPTCP, CMT/RPv2, Multihoming

Kurzzusammenfassung
Diese Bachelorarbeit befasst sich mit der Evaluation des Congestion-Control-Algorithmus CMT/RPv2 für Multipath TCP. Für diesen Zweck wurde der Algorithmus für den Linux Kernel implementiert. Auf einer Testumgebung mit mehrfach angebundenen Rechnern erfolgte eine Gegenüberstellung und Bewertung des CMT/RPv2 mit den bereits etablierten Algorithmen LIA, OLIA und wVegas. Dafür wurden unterschiedliche Szenarien entwickelt, die die Stärken und Schwächen der Algorithmen herausstellen sollen.
Denis Lugowski

Title of the paper
Evaluation of the Congestion Control Algorithm CMT/RPv2 in a MPTCP environment within the Linux Kernel

Keywords
Congestion Control, Multipath TCP, MPTCP, CMT/RPv2, Multihoming

Abstract
This bachelor thesis deals with the evaluation of the Congestion Control Algorithm CMT/RPv2 for Multipath TCP. For this purpose the algorithm has been implemented for the Linux Kernel. On a testbed with multiple connected computer the comparison and evaluation of the CMT/RPv2 took place with the already established algorithms LIA, OLIA and wVegas. There has been developed different scenarios to show strengths and weaknesses of these algorithms.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Das *Transmission Control Protocol* (TCP) [1] ist 35 Jahre nach der Standardisierung durch die IETF[1] bis heute eines der wichtigsten Transportprotokolle im Internet. Mit der stark ansteigenden Anzahl der im Internet angebundenen Hosts und dem damit steigenden Datenverkehr hat das Protokoll seit der Standardisierung jedoch nur wenige Änderungen erfahren. Mittlerweile ist das Vorhandensein von mehreren Netzwerkinterfaces in einem Gerät keine Seltenheit mehr. Smartphones sind in der Lage, über WLAN als auch über die Mobilfunkverbindung im Internet Daten abzurufen. Rechenzentren sind an mehrere *Internet Service Provider* (ISP) angebunden. Leider findet diese Mehrfachanbindung bei TCP keine Berücksichtigung, obwohl eine Anbindung an mehrere Netzwerke gleichzeitig möglich wäre (*multihoming*). So bleiben Ressourcen in Form von Datendurchsatz und Ausfallsicherheit ungenutzt.

Bei *Multipath TCP* (MPTCP) [2] handelt es sich um einen Ansatz, welcher die Beschränkung von TCP, nur einen Pfad nutzen zu können, aufhebt. Ein Pfad ist hier eine Folge von Links zwischen den Teilnehmern der Kommunikation, welches durch das 4-Tupel (*Source IP*, *Destination IP*, *Source Port*, *Destination Port*) definiert wird. Es ermöglicht den Aufbau und die Nutzung von mehreren Pfaden innerhalb einer MPTCP-Verbindung. Diese Pfade der MPTCP-Verbindung werden *subflows* genannt und tragen zur Ausfallsicherheit der Verbindung bei, indem die Kommunikation bei einem Ausfall eines *subflows* auf die verbleibenden *subflows* fortgeführt wird. Des weiteren kann bei einer gleichzeitigen Übertragung über alle *subflows* die Datendurchsatzrate erhöht werden. Man spricht hier von *Concurrent Multipath Transfer*, wenn *subflows* gleichzeitig für die Datenübertragung verwendet werden.

Der *Concurrent Multipath Transfer* für MPTCP hat jedoch das Problem, dass bei einem Zusammentreffen von *subflows* einer MPTCP-Verbindung auf einem gemeinsamen Link die anderen Teilnehmer auf diesem Link benachteiligt werden und nicht ihren fairen Anteil der Kapazität erhalten. Dieses Verhalten tritt auf, wenn auf jeden einzelnen *subflow* ein Congestion-Control-Algorithmus von TCP angewandt wird [3]. Aus diesem Grund wurde das *Resource Pooling Prin-*

[1] Abkürzung für Internet Engineering Task Force (www.ietf.org)

ciple [4] konzipiert, welches Grundsätze definiert, wie sich Congestion-Control-Algorithmen für MPTCP zu verhalten haben [5]. Das erste Ziel besagt, dass der Durchsatz mindestens genauso hoch wie bei TCP sein soll. Beim zweiten Ziel soll jeder *subflow* einer MPTCP-Verbindung nicht mehr Kapazität in Anspruch nehmen, als eine gewöhnliche TCP-Verbindung über diesen Pfad. Als letztes Ziel soll eine MPTCP-Verbindung überlastete *subflows* durch Umleitungen auf weniger belastete *subflows* entlasten. Diese Prinzipien werden im dritten Kapitel näher beleuchtet. Beispiele für die Umsetzung des *Resource Poolings* finden sich in den Algorithmen LIA und OLIA wieder. Der Algorithmus wVegas setzt dagegen eine Alternative des *Resource Poolings* namens *Congestion Equality Principle* um. Diese aufgeführten Algorithmen wurden bereits im Linux Kernel für MPTCP implementiert. Daneben folgt die *Congestion Control* CMT/RPv2 der Idee des *Resource Poolings*, welches bisher nur für SCTP unter FreeBSD analysiert und implementiert wurde. Eine Analyse anhand einer Implementierung für MPTCP existiert somit noch nicht.

Aus diesem Grund wird sich diese Arbeit mit der Implementierung des CMT/RPv2 und dessen Evaluierung befassen. Die methodische Vorgehensweise, um dieses Vorhaben zu realisieren, ist, das CMT/RPv2 mit den vorgestellten Congestion-Control-Algorithmen LIA, OLIA und wVegas zu vergleichen. Anhand der Eigenschaften der einzelnen Algorithmen werden Annahmen über den möglichen Verlauf des Vergleichs getroffen und es wird überprüft, inwiefern diese mit den Ergebnissen übereinstimmen. Zuletzt findet eine Bewertung der Ergebnisse anhand vorher bestimmten Kriterien statt. Der Vergleich selbst wird auf eine *multihomed* Testumgebung stattfinden, die kontrollierbare und reproduzierbare Ergebnisse ermöglicht.

1.1. Motivation

MPTCP kann die Kommunikation auf der Transportprotokoll-Ebene grundlegend verändern. Es bringt die Möglichkeit mit, die Datenübertragung hinsichtlich des Durchsatzes und der Ausfallsicherheit signifikant zu verbessern, ohne auf die Abwärtskompatibilität des weit verbreiteten TCP zu verzichten. Die dazu notwendige Mehrfachanbindung ist bei den meisten Geräten gegeben. Bereits jetzt findet MPTCP im Linux Kernel[2], in FreeBSD[3] und in allen Apple iPhones mit mindestens iOS 7[4] Anwendung.

Es existieren viele wissenschaftliche Ausarbeitungen zu den bereits genannten und implemen-

[2] http://www.multipath-tcp.org/ (02.06.2016)
[3] http://caia.swin.edu.au/urp/newtcp/mptcp/tools/v051/mptcp-readme-v0.51.txt (02.06.2016)
[4] https://support.apple.com/de-de/HT201373 (02.06.2016)

tierten Congestion-Control-Algorithmen. Jedoch steht eine Implementierung und Untersuchung des CMT/RPv2 für MPTCP noch aus. Daher ist es wichtig zu wissen, wie das CMT/RPv2 gegenüber den anderen Algorithmen abschneidet und ob eine aktive Nutzung vorstellbar wäre. Anders als in [6] wird der Vergleich der Algorithmen auf einer praxisnahen *multihomed* Testumgebung stattfinden.

2. Multipath TCP

Im Januar 2013 wurde Multipath TCP als RFC 6824 [2] der Öffentlichkeit bekannt gemacht. Das Transportprotokoll stellt eine Erweiterung zu TCP dar, welches im Gegensatz dazu die Nutzung von mehreren Pfaden gleichzeitig für die Datenübertragung erlaubt. Die Voraussetzung dafür ist, dass alle Teilnehmer der Verbindung über *multihoming* (siehe Kapitel 1) verfügen müssen. Andernfalls wird die Datenübertragung über das reguläre TCP stattfinden, was die Abwärtskompatibilität des Protokolls zu TCP gewährleistet. Um all diese Fähigkeiten erst möglich zu machen, mussten grundlegende Entscheidungen beim Aufbau von MPTCP getroffen werden.

Anders als zuerst angenommen, ersetzt MPTCP TCP nicht, sondern positioniert sich als eine Zwischenschicht innerhalb der Transportschicht unter der Socket API und über dem Standard-TCP (siehe Abbildung 2.1). Hier nimmt MPTCP die Daten der darüberliegenden Sockets, welche von den Applikationen stammen, entgegen und schickt diese über die verwalteten *subflows* (siehe Kapitel 1) hinaus.

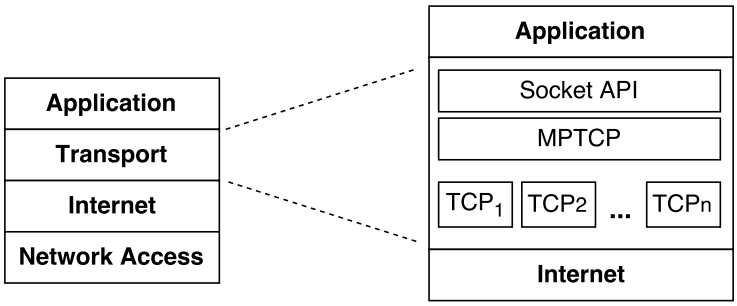


Abbildung 2.1.: Aufbau von MPTCP im TCP/IP-Modell

Applikationen, welche die Socket API nutzen, sind somit weiterhin nutzbar unter MPTCP. Dabei stellt MPTCP gegenüber den nicht-MPTCP-fähigen Applikationen die selbe Schnittstelle wie eine normale TCP-Verbindung bereit und erlaubt weiterhin die Kommunikation über einen Pfad. Um die Funktionen von MPTCP nutzbar zu machen, ist jedoch der Zugriff auf die Er-

weiterung der Socket API durch MPTCP und somit eine Anpassung der Applikation notwendig.

Gegenüber der Internet-Schicht sieht eine MPTCP-Verbindung aus wie eine oder mehrere TCP-Verbindungen. Der Hintergrund dafür ist, dass *Middleboxes* den Datenverkehr filtern. Datenpakete, welche ein für die *Middleboxes* abweichendes Paketformat nutzen, werden oft verworfen [7]. Die Verwendung des selben Paketformats trägt also zu einer höheren Akzeptanz bei den *Middleboxes* bei. Damit die Verbindung für den anderen Teilnehmer als MPTCP-Verbindung erkenntlich gemacht werden kann, muss der TCP-Header um weitere Optionen ergänzt werden.

2.1. Protokollablauf

Eine Verbindung bei TCP wird charakterisiert durch ihren Aufbau, der eigentlichen Kommunikation und den Abbau. Dieser Verlauf findet sich auch bei MPTCP wieder. Beim initialen Verbindungsaufbau für MPTCP wird zusätzlich eine Option mitgegeben, mit der verhandelt wird, ob der Kommunikationsteilnehmer bereit ist, mit MPTCP zu kommunizieren oder ob ein *fallback* auf TCP stattfindet. Wenn der Verbindungsaufbau mit MPTCP geglückt ist, können zur Erhöhung der Ausfallsicherheit und Datenübertragungsrate zusätzliche *subflows* aufgebaut werden. Der Aufbau dieser ist dabei, bis auf die Übertragung zusätzlicher Sicherheitsinformationen zur Einordnung und Authentifikation des *subflows*, analog zum initialen Verbindungsaufbau [8].

Das Besondere an der Datenübertragung mit MPTCP ist, dass ein Datenstrom eines Senders auf mehrere *subflows* aufgeteilt werden kann. Diese Aufteilung birgt jedoch die Gefahr, die Daten nicht mehr wie sonst in TCP üblich in einer gesicherten Reihenfolge übertragen zu können, da die *subflows* unterschiedliche Paketlaufzeiten aufweisen können. Um die Datenpakete nach dem Eintreffen beim Empfänger in der richtigen Reihenfolge zusammenzusetzen, hat man sich für zwei Arten von Sequenznummerbändern entschieden. Die *Data Sequence Number* bestimmt die Reihenfolge der übertragenen Daten über die verschiedenen *subflows* auf der Ebene der MPTCP-Verbindung. So wird vermieden, dass durch Überholungen von schnelleren *subflows* die Daten in der falschen Reihenfolge an die Applikation weitergegeben werden. Auf der Subflow-Ebene agiert die *Subflow Sequence Number*, welche genauso wie die Sequenznummern in TCP die Daten innerhalb des *subflows* auf die richtige Reihenfolge und Paketverluste hin überprüft.

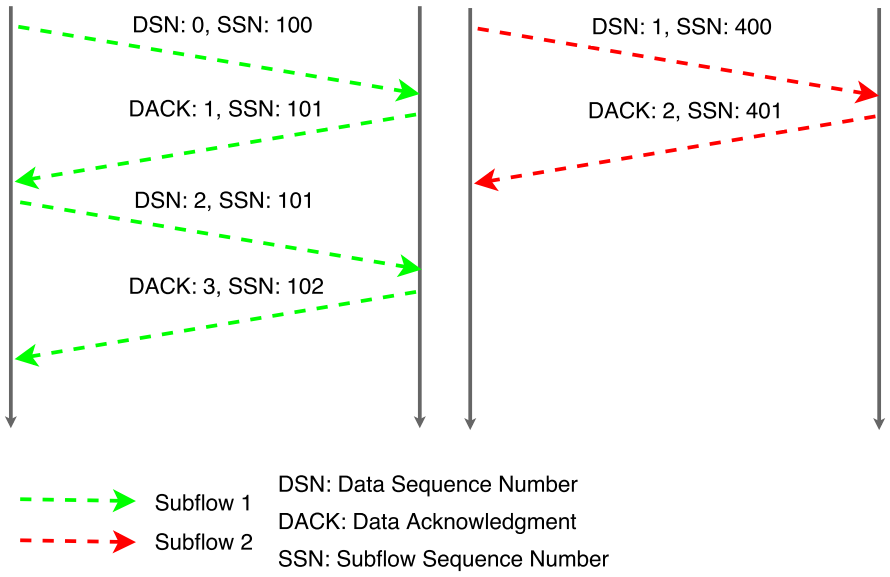


Abbildung 2.2.: Beispiel zu den Sequenznummerbändern

In der Abbildung 2.2 ist ein Beispiel zu den Sequenznummerbändern zu sehen. Es existieren zwei *subflows* welche parallel für die Datenübertragung genutzt werden. Zuerst wird ein Paket mit der *Data Sequence Number* (DSN) 0 und der *Subflow Sequence Number* (SSN) 100 gesendet. Der Empfänger bestätigt das Paket und gibt die zu erwartende DSN und SSN mit. Kurz danach wird auf dem zweiten subflow das nächste Datenpaket versendet. Die SSN ist hierbei eine ganz andere als im ersten *subflow* und absolut unabhängig von den anderer *subflows*. Am Ende wird letzte Paket über den ersten *subflow* mit der eigenen SSN übertragen. Die eingetroffenen Datenpakete werden nach ihrer DSN sortiert und anschließend der Applikation des Anwenders übergeben.

2.2. Bottleneck-Problem

Wenn eine TCP-Verbindung zur Übertragung von Daten genutzt wird, so erwartet man die bestmögliche Datenübertragungsrate dafür. Gleichzeitig muss gewährleistet werden, dass das darunterliegende Netzwerk mit der Menge an Daten zurechtkommt und keine Überlastsituationen auftreten. Diese Situationen machen sich durch den Verlust von Paketen bemerkbar. Der Grund für den Verlust sind die in einer Verbindung dazwischenliegenden Router, die mit der Verarbeitung der Daten nicht schnell genug sind und bereits so viele Pakete in ihre Buffer zwischengespeichert haben, dass keine weiteren mehr aufgenommen werden können. Es kommt

dann zum Verlust von Paketen. Abhilfe schafft hier die *Congestion Control* [9]. Die *Congestion Control* definiert für den Sender Mechanismen (*slow start* und *congestion avoidance* [10]), die eine ungefähre Abschätzung der zur Verfügung stehenden Kapazität einer TCP-Verbindung vornehmen und darauf aufbauend die Senderate anpassen. Dieser Vorgang geht während der gesamten Datenübertragung vonstatten, um sich auf die ständig ändernden Bedingungen im Netzwerk einstellen zu können. Treffen mehrere TCP-Verbindungen auf einen Link zusammen, so sorgt die *Congestion Control* dafür, dass keine Verbindung mehr Bandbreite erhält als ihr fairerweise zustehen würde. Bei zwei Verbindungen auf einem Link wäre die Aufteilung der Kapazität, im Kontext des Internets und der damit verbundenen *TCP Friendly Rate Control* [11], demnach bei 50:50.

Im Unterschied zu TCP verwendet MPTCP mehrere *subflows* zur Datenübertragung. Jeder dieser *subflows* sieht für alle Außenstehenden der Kommunikation aus wie eine TCP-Verbindung. Der naive Ansatz, um Überlastsituationen für MPTCP vorzubeugen, besteht darin, die *Congestion Control* für TCP auf jeden einzelnen *subflow* anzuwenden.

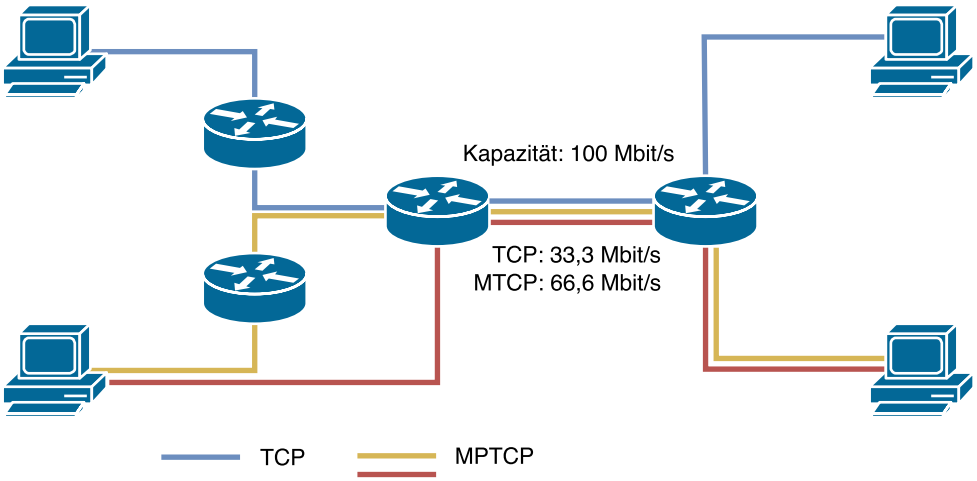


Abbildung 2.3.: Beispiel eines Bottleneck-Problems

Die Abbildung 2.3 veranschaulicht das daraus resultierende Problem anhand eines Beispiels. Es existiert eine TCP- und eine MPTCP-Verbindung. Die Verbindung mit MPTCP verwaltet zwei *subflows* zur Datenübertragung. In beiden Verbindungen treffen die Pfade auf einen Link mit einer Kapazität von 100 Mbit/s zusammen und müssen diesen miteinander teilen. Dieser geteilte Link wird *Shared Bottleneck* oder auch nur *Bottleneck* genannt. In der Abbildung wird deutlich, dass die für die Bereitstellung unterschiedlicher Pfade zuständige *Path Management Method* in

MPTCP dabei nicht nur auf vollkommen disjunkte Pfade beschränkt ist. Die Pfade müssen nur soweit disjunkt sein, um MPTCP eine erhöhte Datendurchsatzrate und Widerstandsfähigkeit liefern zu können [7]. Der Einsatz von *Congestion Control* auf jeden einzelnen Pfad bewirkt, dass die *subflows* der MPTCP-Verbindung wie eigenständige TCP-Verbindungen betrachtet werden. Jeder Verbindungspfad würde demnach 33,3 Mbit/s erhalten. Da jedoch die *subflows* im Kontext der MPTCP-Verbindung stehen, resultiert daraus eine unfaire Bandbreitenverteilung von 66,6 Mbit/s für die MPTCP-Verbindung und 33,3 Mbit/s für die TCP-Verbindung. Es steht außer Frage, dass diese unfaire Bandbreitenverteilung beseitigt werden muss, damit ein Einsatz von MPTCP in der Praxis Erwägung findet.

3. Coupled Congestion Control

Der Ansatz, die *Congestion Control* von TCP auf jeden einzelnen *subflow* anzuwenden, hat gezeigt, dass es bei einem Zusammentreffen der Pfade auf einen Link zum Bottleneck-Problem kommt. Die *subflows* wurden gehandhabt wie vollkommen unabhängige TCP-Verbindungen, obwohl sie im Kontext der MPTCP-Verbindung standen. Diesem Problem widmet sich die *Coupled Congestion Control*. Unter der *Coupled Congestion Control* versteht man Algorithmen, welche sich an Regeln oder Prinzipien halten, die die Fairness an einem *Shared Bottleneck* gewährleisten. Im nachfolgenden Abschnitt wird das *Resource Pooling Principle* [4] vorgestellt, welches bereits von einigen Congestion-Control-Algorithmen für MPTCP Unterstützung findet.

3.1. Das Resource Pooling Principle

Das *Resource Pooling Principle* sieht alle *subflows* als Ressourcen. Anstatt diese unabhängig voneinander zu betrachten und anzusteuern, werden sie miteinander gekoppelt und zu einer gemeinsamen Ressource gebündelt. Daraus resultieren die Vorteile einer erhöhten Widerstandsfähigkeit gegen den Ausfall von Komponenten, eine bessere Möglichkeit, auf ansteigenden Datenverkehr zu reagieren und eine maximierte Ausnutzung der Kapazität [12].

Um das Resource Pooling sicherzustellen, wurden drei Grundsätze definiert [5]:

1. *(Improve throughput) A multipath flow should perform at least as well as a single-path flow would on the best of the paths to it.*
2. *(Do not harm) A multipath flow should not take up any more capacity on any of its paths than if it was a single path flow using only that route.*
3. *(Balance congestion) A multipath flow should move as much traffic as possible off its most-congested paths, subject to meeting the first two goals.*

Die Regeln 1 und 2 sagen im Grunde aus, dass die Gesamtheit aller MPTCP-Subflows (*multipath flow*) mindestens so viel Durchsatz wie eine gewöhnliche TCP-Verbindung auf dem besten Pfad haben muss, aber auf allen Pfaden einzeln nicht mehr Bandbreite als eine TCP-Verbindung

in Anspruch nehmen darf. Diese Regeln in Verbindung miteinander beseitigen bereits das vorher behandelte Bottleneck-Problem. Es wird verhindert, dass die *subflows* einer MPTCP-Verbindung wie einzelne TCP-Verbindungen angesteuert werden. Damit entspricht MPTCP den Vorgaben der *TCP Friendly Rate Control* [11], wodurch die Kapazität eines Links zwischen den Verbindungen fair aufgeteilt wird.

Das eigentliche *Resource Pooling* findet sich in der dritten Regel wieder. Die Kapazität aller Links wird unter allen *subflows* geteilt. Bei Auftreten einer Überlastsituation auf einem *subflow* wird so viel Datenverkehr wie möglich auf weniger belastete *subflows* umgeleitet. Auf diese Weise haben alle *subflows* nach einiger Zeit die selbe Verlustrate, weil die Verluste auf den vorher belasteten Pfaden aufgrund der Umleitung zurückgehen, während die Verluste auf den weniger belasteten Pfaden sich erhöhen. Damit wird ein Lastausgleich über alle Pfade hinweg erzielt.

3.1.1. Statistical Multiplexing und Packet Switching

Das *Resource Pooling* wirkt sich auch indirekt darauf aus, auf welchen *subflows* die Daten verschickt werden. Die nachfolgende Abbildung macht dies deutlich:

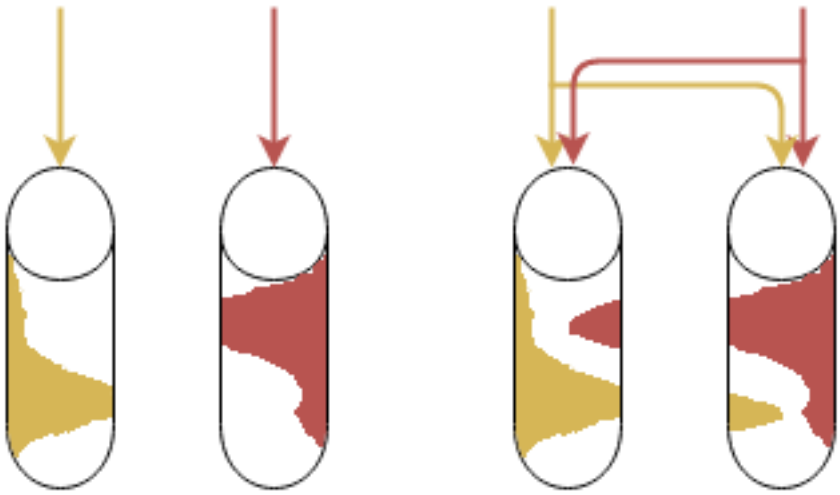


Abbildung 3.1.: Circuit Switching (links), Packet Switching (rechts)

Links sind zwei voneinander unabhängige *subflows* abgebildet. Ohne *Resource Pooling* bleibt die ungenutzte Kapazität auf anderen Pfaden vollkommen unberücksichtigt, obwohl zu der Zeit der Lastspitze auf dem anderen *subflow* noch genug Kapazität frei wäre, um diese aufzufangen.

Diese Form der Übertragung mit unabhängigen Ressourcen wird auch *circuit switching* genannt. Dagegen ist rechts eine Datenübertragung mit *Resource Pooling* und dem damit verbundenen *Packet Switching* abgebildet. Das *Packet Switching* für MPTCP erlaubt der Kommunikation bei Bedarf, einen Teil der Datenübertragung eines *subflows* auf einen anderen *subflow* mit noch frei verfügbarer Kapazität umzuleiten. So wird über mehrere *subflows* hinweg ein statistisches Multiplexing erzielt, was bedeutet, dass bei Bedarf der Datenverkehr zwischen den *subflows* geteilt wird. Diese Form der Datenübertragung gewährleistet eine maximale Auslastung der Kapazität eines Links und vermindert bei einer Überlast auf einem *subflow* die Last.

3.2. Konkrete Implementierungen des Resource Pooling Principles

Das *Resource Pooling Principle* existiert seit dem Jahr 2008 und hat bereits in einigen Congestion-Control-Algorithmen Anwendung gefunden. Dabei funktionieren alle Congestion-Control-Algorithmen nicht gleich, sondern unterscheiden sich hinsichtlich ihrer Art den Datendurchsatz zu erhöhen und darin welches Signal als Anzeichen für eine Überlastung im Netzwerk gewertet wird. Jede dieser Eigenschaften wird für die vorgestellten Algorithmen durchleuchtet und ihre Vor- und Nachteile abgewogen.

3.2.1. Loss-Based Congestion Control

Diese Art der *Congestion Control* ist am häufigsten im öffentlichen Internet zu finden. Hier reagiert der Algorithmus erst mit einer Reduzierung des *Congestion Windows* und der damit verbundenen Senderate, sobald ein Paketverlust in der Datenübertragung identifiziert wurde. Das *Congestion Window* ist dabei eine Anzahl an TCP-Segmenten, die der Sender auf einmal senden kann, ohne auf ein *Acknowledgment* warten zu müssen. Der Paketverlust kann ein Anzeichen dafür sein, dass auf dem Verbindungspfad mindestens ein Router mit der Vielzahl an empfangenen Datenpaketen überfordert ist. Es sind in kurzer Zeit mehr Datenpakete eingetroffen, als der Router verarbeiten kann. So haben sich die Pakete soweit in dessen Buffer angestaut, bis dieser übergelaufen ist. In diesem Fall kann der Router keine weiteren Pakete zur Weiterverarbeitung empfangen und wird diese dann verwerfen.

Es bleibt hier festzuhalten, dass die Reduzierung der Senderate erst stattfindet, wenn die Überlast im Netzwerk bereits aufgetreten ist.

3.2.1.1. Linked Increases Algorithm (LIA)

Die LIA-Congestion-Control [3] ist ein loss-based Algorithmus, welcher nur in der Congestion-Avoidance-Phase der *Congestion Control* angewandt wird. Die *Congestion Avoidance* lässt sich in zwei Phasen unterteilen: Das *additive increase* ist für den linearen Anstieg des *Congestion Windows* zuständig und das *multiplicative decrease* reduziert das Sendefenster um einen bestimmten Faktor bei Auftreten eines Paketverlusts. LIA findet genau gesehen nur in der Phase des *additive increase* statt. Tritt ein Paketverlust bei LIA ein, so wird das Standardverhalten der *Congestion Control* New Reno (im nachfolgenden nur Reno genannt) für TCP beim *multiplicative decrease* beibehalten (Halbierung des *Congestion Windows*). Weiter übernimmt LIA die Slow-Start-Phase von Reno für sich. Der Algorithmus selbst erreicht keine volle Unterstützung des *Resource Poolings*, da die dritte Regel nicht ganz erfüllt werden kann. Im weiteren Verlauf dieser Arbeit wird auf dieses Problem genauer eingegangen.

Die Erhöhung des *Congestion Windows* für einen *subflow* i beim *additive increase* wird bei LIA bei jedem eintreffenden *Acknowledgment* durch folgende Formel bestimmt:

$$min\Big(\frac{alpha * bytes_{acked} * MSS_i}{cwnd_{total}}, \frac{bytes_{acked} * MSS_i}{cwnd_i}\Big) \tag{3.2.1}$$

Sei $cwnd_i$ das *Congestion Window* eines *subflows* i. So ist

$$\sum_i cwnd_i = cwnd_{total} \tag{3.2.2}$$

Die MSS_i gibt hier die *Maximum Segment Size* auf einem *subflow* i an. Das ist die maximale Anzahl an Nutzdaten, die innerhalb eines TCP-Segments versendet werden können.
Das $bytes_{acked}$ wurde im RFC3465 zum *Appropriate Byte Counting* erstmals eingeführt [13]. Anstatt das *Congestion Window* nach Eintreffen einer bestimmten Anzahl an *Acknowledgments* zu erhöhen, wird beim *Appropriate Byte Counting* das *Congestion Window* auf Basis der Anzahl eingetroffenen *acknowledged* Bytes erhöht. Diese Anzahl an Bytes wird dann in $bytes_{acked}$ abgelegt.
Mit der Variable $alpha$ wird die Aggressivität einer MPTCP-Verbindung beschrieben. $alpha$ wird dabei so berechnet, dass für die gesamte MPTCP-Verbindung die erste Regel des *Resource Pooling Principles* (MPTCP muss mindestens den Durchsatz von TCP auf dem besten Pfad haben) eingehalten wird.

Die Formel (3.2.1) gibt an, um welchen Wert das $cwnd_i$ bei Erhalt eines *Acknowledgments*

erhöht wird. Dazu enthält es zwei Ausdrücke, von denen das Minimum genommen wird. Der erste Ausdruck berechnet die Erhöhung des Fensters für den *subflow*, während der zweite die Erhöhung bei einer gewöhnlichen TCP-Verbindung ermittelt. Durch das Minimum beider Ausdrücke ist sichergestellt, dass eine MPTCP-Verbindung auf einem Pfad nicht mehr Durchsatz erhält, als eine TCP-Verbindung auf dem selben Pfad. Damit folgt es dem Gedanken der zweiten Regel des *Resource Poolings*.

Angenommen, die *Round Trip Time* (RTT) ist für alle *subflows* gleich, so wächst das $cwnd_{total}$ nach Vereinfachung des ersten Ausdrucks der Formel (3.2.1) pro RTT um ungefähr

$$cwnd_{total} \mathrel{+}= alpha * MSS \tag{3.2.3}$$

Die einzelnen *subflows* würden demnach folgenden Zuwachs des $cwnd_i$ nach jeder RTT zählen:

$$cwnd_i \mathrel{+}= \frac{alpha * cwnd_i}{cwnd_{total}} \tag{3.2.4}$$

Zur Berechnung der Variablen $alpha$ wird folgende Formel verwendet:

$$alpha = cwnd_{total} * \frac{MAX(\frac{cwnd_i}{rtt_i^2})}{(\sum_i \frac{cwnd_i}{rtt_i})^2} \tag{3.2.5}$$

Die Variable rtt_i gibt die RTT für einen *subflow* i an. Die Operation MAX liefert den größten berechneten Wert von allen *subflows*.

Wie bereits vorher kurz erwähnt, erreicht LIA keine vollständige Unterstützung des *Resource Pooling Principles*, weil es im Konflikt mit der dritten Regel steht (so viel Datenverkehr wie möglich auf weniger belastete Pfade umzuleiten). Mithilfe der Lösung von Kelly und Voice [14] kann eine vollständige Unterstützung erreicht werden. Jedoch würde dies auf Kosten einer unzureichenden Erfassung frei werdender Kapazität gehen und unnötige Wechsel zwischen *subflows* bei der Datenübertragung durch das Vorhandensein von ähnlichen Paketverlustraten auf allen *subflows* verursachen [15]. Das Verhalten des ständigen Wechsels zwischen den *subflows* wird auch als *flappy* bezeichnet.

3.2.1.2. Opportunistic Linked-Increases Algorithm (OLIA)

Genauso wie LIA basiert die OLIA-Congestion-Control [16] auf dem Verlust von Datenpaketen als Anzeichen für eine Überlast im Netzwerk. Bei einem Paketverlust wird auch das Standardverhalten von Reno übernommen. Ebenso wirkt sich dieser Algorithmus nur auf die Phase des *additive increase* innerhalb der *Congestion Avoidance* aus. Dort nimmt OLIA die Schwäche von LIA in Angriff, keine vollständige Unterstützung des *Resource Pooling Principles* zu bieten. Als Grundlage für diese Verbesserung wurde die Lösung von Kelly und Voice genommen. Diese findet sich im ersten Term der Berechnung des *Congestion Windows* nach jedem eintreffenden *Acknowledgment* wieder:

$$cwnd_i \mathrel{+}= \frac{\frac{cwnd_i}{rtt_i^2}}{(\sum_j \frac{cwnd_j}{rtt_j})^2} * \frac{alpha_i}{cwnd_i} \tag{3.2.6}$$

Am Ende des Kapitels zur LIA-Congestion-Control wurde auf die Nachteile eingegangen, welche eine Umsetzung der Lösung von Kelly und Voice mit sich bringen würde (*flappiness*, mangelhafte Erfassung frei werdender Kapazität). Bei OLIA tritt der zweite Term mit $alpha_i$ in der Formel genau diesen Problemen entgegen. Das bedeutet, dass OLIA das *Resource Pooling Principle* vollständig unterstützt und dabei keine Nachteile dafür in Kauf nehmen muss [17].

Im Linux Kernel findet die OLIA-Congestion-Control bereits Unterstützung. In einem Vergleich zwischen OLIA und LIA [17] zeigt sich, dass OLIA den Datenverkehr durch die vollständige Unterstützung des *Resource Pooling Principles* besser auf weniger belastete *subflows* umleiten kann. Desweiteren hat sich gezeigt, dass bei einem Upgrade einer TCP-Verbindung auf eine MPTCP-Verbindung mit LIA Nachteile hinsichtlich des Datendurchsatzes sowohl für die neue MPTCP-Verbindung als auch für andere MPTCP- und TCP-Verbindungen entstehen. Diese Beeinträchtigungen sind vor allem auf die mangelnde Unterstützung der dritten Regel des *Resource Pooling Principles* zurückzuführen.

3.2.2. Delay-based Congestion Control

Die delay-based *Congestion Control* unterscheidet sich von der loss-based dahingehend, dass als Signal für eine Überlast des Netzwerks nicht der eingetretene Paketverlust genutzt wird. Stattdessen wird auf eine zunehmende Erhöhung der RTT oder des *One-Way-Delays* mit einer Anpassung reagiert. Das Anzeichen der steigenden Paketlaufzeiten deutet darauf hin, dass die Buffer der in der Kommunikation dazwischenliegenden Routern volllaufen und sich deshalb die Verarbeitung der Pakete verzögert. Eine Beibehaltung der Senderate würde dazu führen,

dass die Buffer überlaufen und damit Pakete an den Routern verworfen werden müssten. Dies wird durch eine vorherige Anpassung des *Congestion Windows* vermieden. Dadurch fällt die starke Verringerung des *Congestion Windows* durch die *Fast Recovery* mit dem anschließenden Anstieg weg, welche durch einen Paketverlust sonst üblich auftritt.

Die Einsatzgebiete von delay-based Congestion-Control-Algorithmen beschränken sich auf verzögerungssensitive Anwendungen (z.B. Voice-Over-IP), die bei Auftreten einer Verzögerung stark beeinträchtigt werden und den Hintergrunddatenverkehr (z.B. Updates, Datei-Synchronisierungen), welcher den Verkehr von im Vordergrund laufenden Anwendungen nicht behindern soll [18].

3.2.2.1. Weighted Vegas (wVegas)

Als erster delay-based Congestion-Control-Algorithmus ist wVegas [19] im MPTCP Linux Kernel implementiert worden. Wie im Namen angedeutet, handelt es sich hier um eine abgeleitete Version des TCP Vegas [20], welche für MPTCP einige Änderungen erfahren hat. Durch die Fähigkeit, die Paketumlaufzeit als Signal für eine Überlast im Netzwerk zu nutzen, soll wVegas den Datenverkehr besser auf weniger belastete *subflows* umleiten als die loss-based Pendants.

Im Gegensatz zu den bisher vorgestellten Algorithmen für *Congestion Control* folgt wVegas nicht dem *Resource Pooling Principle*, sondern dem *Congestion Equality Principle*. Das *Congestion Equality Principle* besagt:

> *Wenn jeder flow danach strebt, das Ausmaß von Überlastung auszugleichen, die es mithilfe von Verlagerung des Verkehrs auf allen verfügbaren Pfaden erkennt, dann werden die Netzwerkressourcen fair und effizient zwischen allen flows geteilt.*

Dieser Grundsatz folgt nur anders formuliert dem selben Gedanken wie die dritte Regel des *Resource Pooling Principles*, nämlich so viel Datenverkehr von überlasteten *subflows* wie möglich auf weniger belastete umzuleiten. Abseits von diesem Prinzip hat wVegas auch eine Lösung für das Bottleneck-Problem, welche wie das *Ressource Pooling Principle* die Fairness gegenüber anderen Verbindungen auf einem *Shared Bottleneck* gewährleistet.

Die Funktionsweise von wVegas lässt sich wie folgt zusammenfassen:

- Auf jedem Pfad wird der Algorithmus von TCP Vegas angewandt.

- Jeder *subflow* besitzt eine Alpha-Variable. Die Summe aller Alphas ist dabei unabhängig von der Anzahl an *subflows* immer die selbe und stellt die Fairness innerhalb des Algorithmus sicher.
- Die Alpha-Variable wird für jeden *subflow* individuell angepasst und wirkt sich auf die Größe des *Congestion Windows* aus. Die Anpassungen werden solange vorgenommen, bis die Verlustrate auf allen *subflows* die selbe ist und sie damit ausgeglichen sind.

Die Fähigkeit, frühzeitig auf eine mögliche Überlast zu reagieren, geht bei wVegas wie auch bei TCP Vegas mit folgenden Nachteilen einher: Das Erkennen einer gestiegenen Paketumlaufzeit erfordert eine hohe Genauigkeit bei der Messung dieser. Falsche Zeiten können zu falschen Annahmen in der Berechnung eines neuen *Congestion Windows* führen und damit den Algorithmus stark beeinträchtigen. Aus diesem Grund benötigt wVegas sogenannte *high-resolution timer* [21] zur zuverlässigen Funktionsweise.
Treffen delay-based und loss-based Algorithmen auf einem Link zusammen, so stellt sich heraus, dass die delay-based *Congestion Control* aufgrund der Früherkennung einer Überlastung weniger aggressiv bei der Belegung der Kapazität vorgeht. Die loss-based *Congestion Control* erhöht dagegen das *Congestion Window* so weit, bis eine Überlast im Netzwerk vorliegt und damit Pakete verloren gegangen sind.
In Netzwerken mit einem hohen Bandbreiten-Delay-Produkt verlässt Vegas die Slow-Start-Phase zu früh und verliert viel Zeit, bis das *Congestion Window* in der *Congestion Avoidance* langsam die maxmimale Größe erreicht hat [22]. Dadurch leidet die Effizienz des Algorithmus.

3.3. Concurrent Multipath Transfer / Resource Pooling v2 (CMT/RPv2)

Das CMT/RPv2 [6] ist ein loss-based Congestion-Control-Algorithmus, welcher im Rahmen dieser Bachelorarbeit für den Linux Kernel implementiert und unter bestimmten Testbedingungen mit den anderen vorgestellten Algorithmen verglichen und beurteilt werden soll. Unter FreeBSD existiert bereits für das CMT/RPv2 und dessen Vorgänger CMT/RPv1 eine einsatzfähige Implementierung. Im Vergleich zum CMT/RPv1 kann das CMT/RPv2 mit ungleichen Pfaden, welche unterschiedliche RTT, Bandbreiten oder Paketverlustraten aufweisen, umgehen.

3.3.1. Funktionsweise

Die bisher vorgestellten loss-based Algorithmen sahen nur eine Anpassung des *additive increase* innerhalb der *Congestion Avoidance* vor. Beim CMT/RPv2 hingegen sind auch das *multiplicative*

decrease bei einem Paketverlust und die Slow-Start-Phase von Änderungen betroffen. Zur Berechnung der Erhöhung des *Congestion Windows* wird ein *increase factor* ($\hat{i}_p$) definiert, welcher das Verhältnis der Bandbreite eines *subflows* zu der Gesamtbandbreite bestimmt.

$$\hat{i}_p = \frac{\frac{cwnd_p}{rtt_p}}{\sum_i \frac{cwnd_i}{rtt_i}} \tag{3.3.1}$$

Die Erhöhung des *Congestion Windows* für einen *subflow* p wird folgendermaßen berechnet:

$$cwnd_p = cwnd_p + \begin{cases} \lceil \hat{i}_p * min\{alpha, MSS_p\} \rceil & \text{Bei Slow Start} \\ \lceil \hat{i}_p * MSS_p \rceil & \text{Bei Congestion Avoidance} \end{cases} \tag{3.3.2}$$

Die Variable *alpha* zeigt hier die Anzahl der neuen *acknowledged* Bytes an.

Bei einem erkannten Paketverlust muss die Senderate verringert und der *Slow Start Threshold* angepasst werden. Der *Slow Start Threshold* steht hier für einen Schwellenwert, bis zu dem das *Congestion Window* mit dem *Slow Start* erhöht werden darf. Wird der Wert vom Sendefenster erreicht oder überschritten, darf es nur noch mit der *Congestion Avoidance* erhöht werden. Zur Berechnung eines neuen *Slow Start Threshold* wird ein *decrease factor* $\hat{d}_p$ benötigt.

$$\hat{d_p} = max\{\frac{1}{2}, \frac{1}{2} * \frac{\sum_i \frac{cwnd_i}{rtt_i}}{\frac{cwnd_p}{rtt_p}}\} \tag{3.3.3}$$

Der *Slow Start Threshold* s_p und das *Congestion Window* erfahren folgende Anpassungen bei einem Paketverlust auf einem Pfad p:

$$s_p = max\{cwnd_p - \lceil \hat{d}_p * cwnd_p \rceil\} \tag{3.3.4}$$

$$cwnd_p = \begin{cases} s_p & \text{Bei Fast RTX} \\ MSS_p & \text{Bei Timer-based RTX} \end{cases} \tag{3.3.5}$$

3.3.2. Implementierung im Linux Kernel

Bei dem Linux Kernel handelt es sich um eine Abstraktionsschicht zwischen Hardware und der darauf laufenden Software. Diese ist ein Bestandteil zahlreicher Betriebssysteme (wie z.B. Ubuntu[1], Android[2] usw.) und steht allen Personen zur Einsicht, Bearbeitung und sogar dem Weitervertrieb unter der GNU General Public License[3] frei zur Verfügung [23]. Durch seine weite Verbreitung und dem öffentlichen Zugang ist der Linux Kernel die ideale Plattform, um Erfahrungen mit neuen Algorithmen, Komponenten oder Ähnlichem zu sammeln. Aus diesem Grund bot sich eine Umsetzung des CMT/RPv2 im Linux Kernel nach dessen Implementierung in FreeBSD an.

Die Integration einer neuen *Congestion Control* in den Linux Kernel wird durch die Unterstützung des Konzepts der *Pluggable Congestion Control* [24] erleichtert. Dieses Konzept ermöglicht über eine Schnittstelle die Übergabe eines selbst definierten Verhaltens beim *Additive Increase*, *Multiplicative Deacrease* und dem *Slow Start*. Über die vordefinierte Struktur *tcp_congestion_ops* (zu finden in */include/linux/tcp.h*) werden *Pointer* zu den Funktionen der eigenen *Congestion Control* übergeben, die je nach dem eingetroffenem Ereignis oder den aktuell befindlichen Zustand aufgerufen werden.

```
1 struct tcp_congestion_ops {
2         ...
3         /* return slow start threshold (required) */
4         u32 (*ssthresh)(struct sock *sk);
5         /* do new cwnd calculation (required) */
6         void (*cong_avoid)(struct sock *sk, u32 ack, u32 acked);
7         ...
8 };
```

Listing 3.1: Auszug der Struktur *tcp_congestion_ops*

Dabei müssen zwingend Funktionen für den *Slow Start Threshold* und die *Congestion Avoidance* übergeben werden (siehe Listing 3.1). Bei einem aufgetretenen Paketverlust wird zur Behandlung die Funktion des *Slow Start Threshold* aufgerufen und jedes eintreffende *Acknowledgment* bewirkt den Aufruf der *Congestion Avoidance*. Hier muss unbedingt beachtet werden, dass das *Congestion Window* in der *Congestion Avoidance* nur bei einer vollständigen Übertragung des Sendefensters erhöht werden darf.

[1] https://www.ubuntu.com (11.11.2016)
[2] https://www.android.com (11.11.2016)
[3] https://www.gnu.org/gnu/linux-and-gnu.en.html (30.11.2016)

Die Registrierung der *Congestion Control* erfolgt über die Funktion *tcp_register_congestion_control*, welche die Struktur *tcp_congestion_ops* als Parameter entgegennimmt. Diese Funktion wird letztendlich dem Makro *module_init* übergeben, welche die *Congestion Control* entweder zur Startzeit des Systems (vorausgesetzt, es ist fest im Kernel kompiliert worden) oder beim Laden des Moduls (vorausgesetzt, es wurde als Modul kompiliert) registriert.

Während der Programmierung des Algorithmus kommt man mit der C-Struktur *tcp_sock*, auch unter */include/linux/tcp.h* zu finden, sehr häufig in Kontakt. Diese Struktur stellt das Mittel dar, um direkt Einfluss auf das Verhalten der *Congestion Control* zu nehmen. Um die Struktur überhaupt nutzen zu können, muss diese zuvor mithilfe der Funktion *tcp_sk()* von der Struktur *sock* auf *tcp_sock* gecastet werden. Im Vergleich zur TCP-Implementierung wurde für MPTCP die Struktur um weitere Attribute ergänzt, die genaue Informationen zum MPTCP-Socket, Scheduling und vielem mehr beinhalten. Die wichtigsten Attribute dieser Struktur sind hier zusammengefasst:

```
1 struct tcp_sock {
2         u32     srtt_us;      /* RTT in Mikrosekunden */
3         u32     snd_ssthresh; /* Slow Start Threshold */
4         u32     snd_cwnd;     /* Send Congestion Window */
5         u32     sacked_out;   /* Doppelte Pakete */
6         ...
7 }
```

Listing 3.2: Auszug der Struktur *tcp_sock*

Die oben aufgelisteten Attribute sind auch in der Standardimplementierung von TCP zu finden. Jedoch beziehen sich diese bei MPTCP anstatt auf die Gesamtheit aller *subflows* nur auf einen einzelnen. Dadurch ist es möglich, jeden *subflow* unabhängig von den anderen anzusteuern.

Anders als möglicherweise zuerst angenommen, ist eine direkte Umsetzung der im Kapitel 3.3.1 vorgestellten Formeln erstens nicht möglich und zweitens nicht immer sinnvoll. Das liegt zum einen an der Besonderheit des Linux Kernels, Operationen, welche nur den Kernel betreffen, im *Kernel Space* und alle anderen Operationen im *User Space* durchzuführen. Dabei handelt es sich um gesonderte Bereiche im Kernel, welche die Ausführung von privilegierten Anwendungen, wie beispielsweise dem eigentlichen Kernel selbst, den Hardware-Treibern, Modulen und anderen Erweiterungen von der Ausführung von nicht privilegierten trennt. Mit nicht privilegierten Anwendungen sind jene gemeint, die oft vom Nutzer selbst ausgeführt werden (z.B. Browser, Texteditoren usw.) und denen man, um den Speicher und die Hardware

zu schützen, einen direkten Zugriff auf den *Kernel Space* verweigern möchte. Aus diesem Grund stehen diesen Anwendungen sogenannte *System Calls* zur Verfügung, welche für die Zeit einer Operation vom *User Space* in den *Kernel Space* umschalten und so einen größeren Funktionsumfang, wie z.B. das Öffnen von Dateien oder das Erstellen von neuen Prozessen, ermöglichen [25].

Da das CMT/RPv2 als *Congestion Control* standardmäßig im *Kernel Space* ausgeführt wird [24], bestehen erst einmal keine Bedenken zu möglichen Einschränkungen. Jedoch bringt der *Kernel Space* den Nachteil mit sich, dass keine direkte Ausführung von Fließkomma-Operationen möglich ist. Das An- und Ausschalten der *Floating Point Unit* und die dazugehörige manuelle Sicherung der einzelnen Register erschweren die Handhabung [26]. Daraus folgt auch eine Verlängerung der Ausführungszeit, die in der Abarbeitung von Netzwerkpaketen in schnell angebundenen Netzwerken kritische Folgen haben kann. Um diesen Nachteilen zu entgehen, haben sich im Linux Kernel Bitverschiebungen im Festkomma-Format um einen festen Wert als die bessere Wahl herausgestellt.
Bei einer Division eines Zählers durch einen größeren Nenner resultiert immer eine Fließkommazahl, die im Kernel ohne Zuschaltung der *Floating Point Unit* nicht dargestellt werden kann. Werden die Bits des Zählers dagegen um einen festen Wert nach links verschoben, sodass dieser größer als der Nenner ist, so kann die Fließkommazahl indirekt als Ganzzahl abgebildet werden. Mit dieser Ganzzahl können die Berechnungen wie mit einer Fließkommazahl durchgeführt werden. Um das echte Ergebnis aus der Berechnung zu erhalten, muss dieses nur noch um die vorher nach links verschobene Anzahl an Bits zurückgeschoben werden.
Die Methode der Bitverschiebung findet beim CMT/RPv2 bei der Berechnung des *increase factor* (3.3.1) ihren Einsatz. Der Grund dafür ist der häufig deutlich kleinere Wert des *cwnd* im Vergleich zur RTT. Bei einer normalen Division würden die entscheidenden Nachkommastellen verloren gehen und damit die Erhöhung des *cwnd* ausbleiben. Durch die Verwendung eines 32-Bit-Datentyps für das *cwnd* bietet sich eine Verschiebung auf den nächstgrößeren Datentyp um 32 Bit an. Die Berechnung der Gesamtbandbreite sieht dann wie folgt aus:

```
1 mptcp_for_each_sk(mpcb, sub_sk) {
2         struct tcp_sock *sub_tp = tcp_sk(sub_sk);
3         total_bandwidth += div64_u64(mptcp_rpv2_scale
4         (sub_tp->snd_cwnd, rpv2_scale), sub_tp->srtt_us);
5 }
```

Listing 3.3: Berechnung der Gesamtbandbreite im Code des CMT/RPv2

Hier werden in einer For-Each-Schleife alle *subflows* nacheinander durchlaufen und ihr *cwnd* mit *rpv2_scale* um 32 Bit nach links verschoben. Anschließend folgt die 64-Bit-Division mit

der RTT in Mikrosekunden. Der daraus resultierende Wert repräsentiert die Fließkommazahl, welche durch die Bitverschiebung mit 2^{32} multipliziert wurde.

Die Berechnung der Gesamtbandbreite ist die einzige Stelle bei der Bestimmung des *increase factor*, wo auf die Bitverschiebung zurückgegriffen wird, obwohl laut der Formel (3.3.1) eine zweite Berechnung der Bandbreite im Zähler für den aktuellen *subflow* stattfindet. Diese wird durch eine Umformung der Gleichung aufgelöst. Dazu wird der Faktor des *increase factor* aus der Formel (3.3.2) mit hinzugenommen.

$$\frac{\frac{cwnd_p}{rtt_p}}{\sum_i \frac{cwnd_i}{rtt_i}} * \begin{cases} min\{alpha, MSS_p\} & \text{Bei Slow Start} \\ MSS_p & \text{Bei Congestion Avoidance} \end{cases} \tag{3.3.6}$$

Anschließend wird der Ausdruck soweit vereinfacht, bis diese Gleichung entsteht:

$$\frac{cwnd_p * factor}{rtt_p * \sum_i \frac{cwnd_i}{rtt_i}} \tag{3.3.7}$$

Der Wert dieser Gleichung repräsentiert nun die Anzahl an Bytes, um die das $cwnd_p$ erhöht wird. Die Variable $factor$ steht hier für den Faktor, der in (3.3.6) hinzugenommen wurde.

Die in [6] vorgestellten Formeln sind für eine byte-weise Erhöhung des *cwnd* aufgestellt worden. Jedoch ist es im Linux Kernel nur möglich, ganze Pakete auf das Sendefenster zu addieren. Die Umstellung von einer byte-basierenden Erhöhung zu einer paket-basierenden erfordert bei der Implementierung einen Zwischenpuffer. Dieser Zwischenpuffer speichert die Anzahl an Bytes, um die das *cwnd* erhöht werden soll (3.3.7), zwischen. Sobald der Zwischenpuffer größer gleich einer MSS ist, wird das Fenster um ein Paket erhöht.

Im Falle eines Paketverlusts wird die Berechnung des *decrease factor* ((3.3.3) zweiter Term) auch in einer vereinfachten Form durchgeführt. Hierfür wird genauso wie beim *increase factor* zuvor der Faktor aus der Gleichung (3.3.4) miteinbezogen. Die daraus resultierende Gleichung stellt die Anzahl an Bytes dar, um die das $cwnd_p$ verringert werden soll:

$$\frac{1}{2} * \frac{\sum_i \frac{cwnd_i}{rtt_i}}{\frac{cwnd_p}{rtt_p}} * cwnd_p \tag{3.3.8}$$

Nach einer Reihe von Umformungen und Vereinfachungen ist die Komplexität der neuen Gleichung deutlich gesunken, was der Implementierung sehr zugutekommt:

$$\frac{\sum_i \frac{cwnd_i}{rtt_i} * rtt_p}{2} \tag{3.3.9}$$

Somit konnte im Vergleich zu den ursprünglichen Gleichungen des *increase* und *decrease factor* mithilfe der Umformungen und Zusammenführungen jeweils eine Division eingespart werden.

Die Entwicklung des CMT/RPv2 wurde zusammen mit ersten Tests lokal auf einem Rechner durchgeführt. Erst später folgten Tests und die endgültigen Messungen auf der im vierten Kapitel vorgestellten Testumgebung. Der lokale Testaufbau bestand aus zwei virtuellen Maschinen, die mit Ubuntu betrieben wurden. Beiden Maschinen wurden über das Einstellungsmenü von VirtualBox ein zweites Netzwerkinterface hinzugefügt, um die MPTCP-Fähigkeiten im Rahmen des Aufbaus testen zu können.

Während der Programmierung im Linux Kernel sind Methoden zum Debugging des Quellcodes von großer Bedeutung. Ohne diese ist die Entwicklung am Linux Kernel nicht nur sehr zeitaufwändig, sondern auch unflexibel, da die Programmierung nur von Zeile zu Zeile geschehen kann. Aus diesem Grund haben sich zahlreiche Möglichkeiten herausgebildet, die den Prozess des Debuggings beschleunigen und vereinfachen. Ein kleiner Auszug dieser Werkzeuge zum Debugging sind *GDB*[4], *Kernel Crash Dump*[5], *ftrace*[6] und das *printk*[7]. Diese Methoden wurden in dieser Arbeit einzeln auf ihre Eignung und Handhabbarkeit getestet. Letzten Endes hat sich das *printk* in Verbindung mit dem später vorgestellten *netconsole* als die einfachste und zuverlässigste Variante erwiesen.
Das *printk* im Kernel funktioniert bis auf kleinere Unterschiede genauso wie das *printf* in der Programmiersprache C. Anstelle der Konsole wird beim *printk* die Meldung in dem Kernel-Ring-Buffer-Log hinterlegt, welcher mit dem Kommandozeilen-Befehl *dmesg* eingesehen werden kann. Mit einer zusätzlichen Angabe eines Loglevels kann die Meldung auch auf der Konsole angezeigt werden, wenn der angegebene Level höher als der Level der Kernelvariable *console_loglevel* ist. Bei der Programmierung wurde für die Ausgabe auf das *pr_err_ratelimited* zurückgegriffen. Dieser Befehl vereint das *printk* mit der Angabe des Loglevels *Error* und einer

[4] http://www.elinux.org/Debugging_The_Linux_Kernel_Using_Gdb (24.07.2016)
[5] https://help.ubuntu.com/12.04/serverguide/kernel-crash-dump.html (24.07.2016)
[6] https://lwn.net/Articles/365835/ (24.07.2016)
[7] http://elinux.org/Debugging_by_printing (12.07.2016)

Limitierung bei vielen aufeinanderfolgenden Ausgaben, was der Übersichtlichkeit und der Performance zugutekommt.

Gerät der Linux Kernel durch eine fehlerhafte Programmierung in einen undefinierten Zustand, welcher vom System nicht selbst behandelt werden kann, so tritt ein *Kernel Panic* auf, um das System vor Schäden zu bewahren. Der *Kernel Panic* gibt auf der Konsole einen Auszug der Fehlermeldung aus und erlaubt keine weitere Interaktionen mit dem System. Die Fehlermeldung enthält eine Auflistung der zuletzt aufgerufenen Funktionen, den Inhalt der CPU-Register, des Stacks und vieles mehr. Jedoch wird auf dem Bildschirm nur ein kleiner Teil dieser Informationen ausgegeben, was das Auffinden der eigentlichen Ursache des Problems erschwert. Selbst vorher ausgegebene Debug-Ausgaben mit *printk* sind nicht einsehbar. Das Scrollen nach oben oder Umleiten der Nachricht in eine Textdatei ist nicht möglich, da nach dem Zeitpunkt des Absturzes keine Interrupts mehr vom System verarbeitet werden [27]. Deshalb ist nach einem Neustart auch in den Log-Dateien kein Hinweis auf einen zuvor aufgetretenen *Kernel Panic* zu finden.

Die Lösung des Problems war das Modul *netconsole*[8]. Es ermöglicht das Senden von neu eingetroffenen Meldungen des Kernel-Ring-Buffer-Logs über das Netzwerk. Für die Nutzung von *netconsole* ist auf der Empfangsseite ein Server notwendig, der beispielsweise mit *netcat* auf einen bestimmten Port auf eingehende Nachrichten wartet. Der Sender verbindet sich nach dem Laden des Moduls und der Angabe der Adresse und des Ports mit dem Empfänger. Danach kann selbst nach einem Absturz des Sendersystems die Meldung des *Kernel Panics* einschließlich der verursachenden Funktion auf dem Empfänger eingesehen werden.

[8] https://wiki.ubuntu.com/Kernel/Netconsole (28.07.2016)

4. Vergleich der Coupled Congestion Control

Nach erfolgter Implementierung des CMT/RPv2 stellt sich die Frage, wie es im Vergleich zu den anderen vorgestellten Algorithmen, unter der Berücksichtigung der in Kapitel 4.4 vorgestellten Kriterien, abschneidet. Dazu ist eine für MPTCP geeignete Testumgebung notwendig. Es müssen Testszenarien festgelegt werden, unter denen die Vergleiche stattfinden und die näheren Umstände der Messungen beschrieben werden. Nach erfolgter Messung sind Kriterien von großer Bedeutung, um das Ergebnis richtig einordnen zu können. In den folgenden Abschnitten werden diese Punkte definiert.

4.1. Die Testumgebung

Eine Eigenschaft von MPTCP besteht darin, Datenübertragungen über mehrere sowohl logische als auch physikalische Datenpfade gleichzeitig durchzuführen. Um Algorithmen für die *Congestion Control* im vollem Umfang testen zu können, ist es notwendig, dass mindestens zwei physikalische Pfade zum Empfänger hinführen. Diese müssen, wie schon im Kapitel 2.2 erwähnt, nicht vollkommen disjunkt sein. Die von der Hochschule für angewandte Wissenschaften Hamburg zur Verfügung gestellte Testumgebung erfüllt diese Bedingung und ermöglicht damit Tests mit MPTCP. Die Testumgebung besteht aus vier Computern der Firma Dell, welche mit jeweils drei Netzwerkkarten ausgerüstet worden sind. Weitere Spezifikationen zu den Computern sind aus der nachstehenden Auflistung zu entnehmen:

- **Name**: North, East, South, West
- **CPU**: Intel Core i5-4690 (4 x 3.50GHz)
- **RAM**: 4GB
- **Netzwerkkarten Medientyp**: 1000baseT

Die Anordnung und Verbindung der einzelnen Rechner ist im ersten Setup ringförmig. Entsprechend nach ihrer Position wurden die Rechner jeweils nach einer Himmelsrichtung benannt (North, East, South, West). In der folgenden Abbildung ist die Anordnung und Verbindung der Netzwerkinterfaces gut ersichtlich:

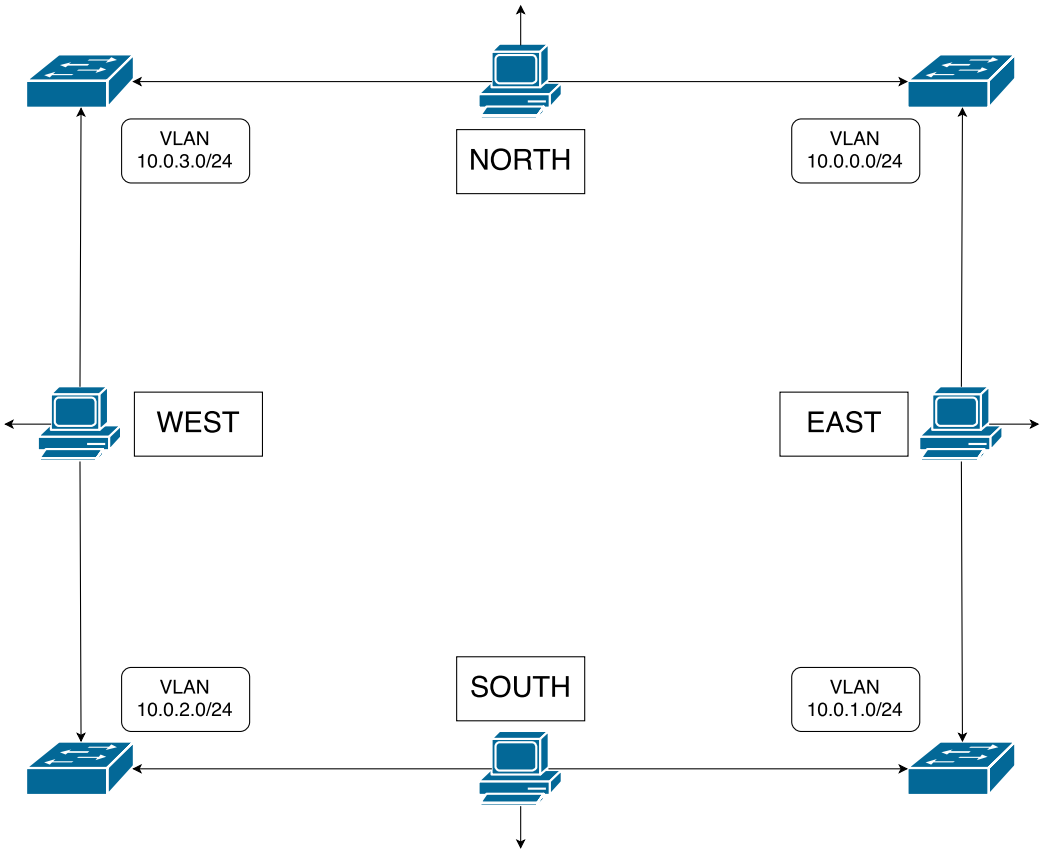


Abbildung 4.1.: Testumgebung mit *Half-Bottleneck*

Das Netzwerkinterface eth0 verbindet die Rechner mit dem Internet und ermöglicht einen Zugriff von außen. Über die Interfaces eth1 und eth2 sind die Rechner über jeweils einen Port eines gemeinsamen Switches mit ihren Nachbarrechnern verbunden. Damit sind die Rechner über genau zwei unabhängige Pfade zu erreichen. Der Switch unterteilt das Netzwerk hierbei in die vier Subnetze 10.0.0.0/24, 10.0.1.0/24, 10.0.2.0/24 und 10.0.3.0/24, welche einem *Virtual Local Area Networks* (VLAN) zugeordnet werden. Das heißt, jeder Rechner ist an genau zwei VLANs angeschlossen. Obwohl die Rechner physikalisch miteinander verbunden sind, ist es nicht möglich, dass zwei Rechner von verschiedenen VLANs aus direkt miteinander kommunizieren.

Für diese Kommunikation wird ein Router vorausgesetzt. Aus diesem Grund sind alle Rechner der Testumgebung auch als Router konfiguriert worden und besitzen eine eigene statisch angelegte Routing-Tabelle, in der hinterlegt ist, wohin ein Paket anhand der Zieladresse zu senden ist. Diese Form des Routings wird auch *Destination-based Routing* genannt [28].

Der Aufbau der Testumgebung in Abbildung 4.1 entspricht dem eines *Half-Bottlenecks*. Im Unterschied zum in 4.2 abgebildeten *Bottleneck* treffen nicht alle *Flows* des *Half-Bottlenecks* während der Datenübertragung auf einen einzelnen Link zusammen und sind gezwungen diesen gemeinsam zu teilen. Stattdessen wird ein weiterer Link zur Verfügung gestellt, über den eine MPTCP-Verbindung einen *subflow* erstellen kann. Damit besteht mit einer weiteren TCP-Verbindung das Szenario, dass eine TCP-Verbindung und ein *MPTCP-Subflow* einen Link teilen und ein weiterer *subflow* frei auf dem alternativen Pfad sendet. Es stehen somit nicht alle *subflows* einer MPTCP-Verbindung in Konkurrenz mit der TCP-Verbindung.

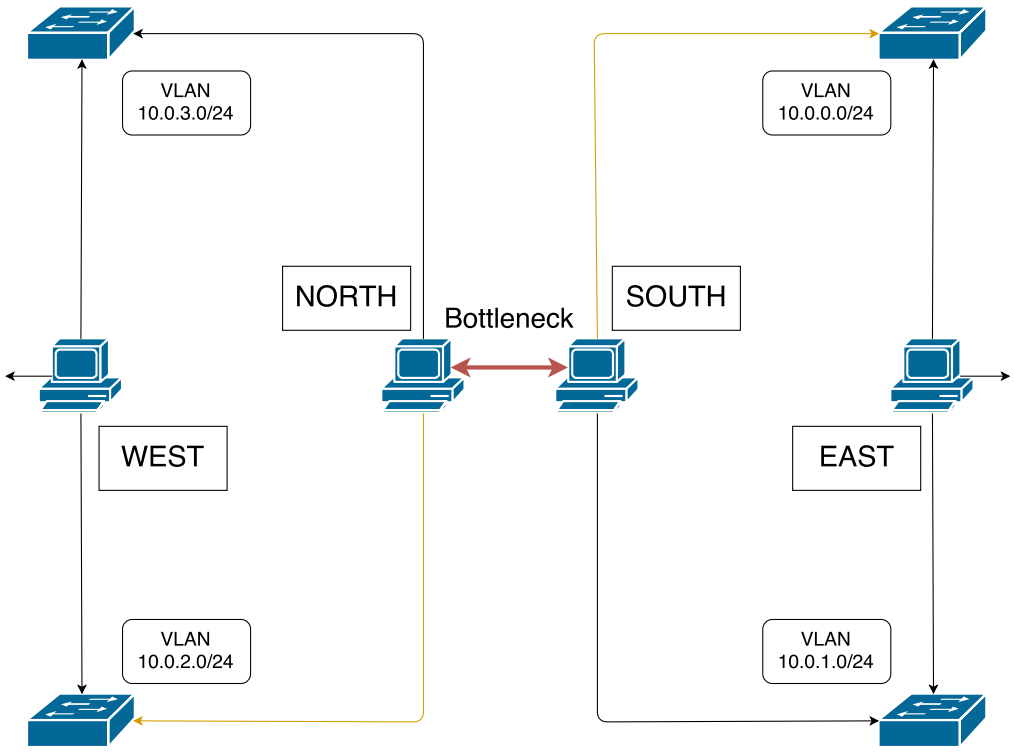


Abbildung 4.2.: Testumgebung mit *Bottleneck*

Die alternative Anordnung, wie in 4.2 abgebildet, wird zusätzlich um einen weiteren Link zwischen den Rechnern North und South ergänzt. Weiter werden in diesem Setup die Eth2-Interfaces von North und South an ein anderes VLAN angeschlossen (gelbe Links). Eine Umleitung von Paketen nur über die außenliegenden Links ist damit nicht mehr möglich. Besonders wichtig ist hier, dass die *Flows* zwischen North und South in die selbe Richtung senden. Hätte man beispielsweise im Aufbau des *Half-Bottlenecks* einen zusätzlichen Link zwischen North und South erstellt und über die Routingtabelle eine Umleitung über die Außen unterbunden, so würden East oder West die Pakete einer MPTCP-Verbindung über South und North verschicken. Damit würde die Übertragung zwischen North und South in beide Richtungen gleichzeitig (*Full-Duplex*) verlaufen und die Verbindungen könnten sich nicht, wie sonst in einem *Bottleneck* üblich, gegenseitig beeinträchtigen. Dieser Aufbau hätte somit nicht in einem *Bottleneck* resultiert. Dagegen wird mit der Verlegung der Links zu den Eth2-Interfaces von North und South garantiert, dass East nur South und West nur North direkt erreichen kann und die Übertragung unidirektional erfolgt. Dadurch wird künstlich ein *Bottleneck* kreiert, welches sich ideal dafür eignet, zu überprüfen, ob das CMT/RPv2 tatsächlich auch in der Praxis zusammen mit anderen *Flows* auf dem Link fair ist und nach den Prinzipien des *Resource Poolings* agiert.

Die Testumgebung besitzt die große Flexibilität, dass auf allen Rechnern sowohl FreeBSD als auch Ubuntu 14.04 installiert ist. Senderseitig ist das CMT/RPv2 für Linux implementiert worden, weshalb dort Ubuntu 14.04 zum Einsatz kommt. Die Empfängerseite wird ebenso mit Ubuntu betrieben. Für die Routingtätigkeiten zwischen den Kommunikationsteilnehmern wird dagegen auf FreeBSD gesetzt. In den Messungen wird die Senderseite dem Rechner East und die Empfangsseite West zugeordnet, während North und South das Routing durchführen.

Um unterschiedliche Szenarien in der Testumgebung abbilden zu können, ist ein Mechanismus notwendig, um den Datenverkehr zu manipulieren. Dies geschieht in Form von dummynet [29] auf den Rechnern mit FreeBSD. Dummynet ist ein Bestandteil der IP-Firewall (*ipfw*) [30] von FreeBSD und wird über diese konfiguriert. Mithilfe von dummynet ist es möglich, künstlich die Übertragungsdauer, -rate und den Verlust von Datenpaketen, als auch das Management der Queues in den Routern zu beeinflussen. Eine Regel, um beispielsweise die Datenübertragungsrate zu limitieren, sieht folgendermaßen aus:

```
1 ipfw add pipe 1 ip from any to any
2 ipfw pipe 1 config bw 10Mbit/s
```

Listing 4.1: Beispiel einer Dummynet-Regel

Der erste Befehl kreiert eine *pipe*, welche auf alle Pakete mit IPv4 oder IPv6 für alle Quell- und Zieladressen angewandt wird. Eine *pipe* stellt einen emulierten Link dar, welcher künstlich das Verhalten der eingestellten Konfigurationen annimmt. Die nachfolgende Anweisung setzt die Übertragungsrate für diese *pipe* auf 10 Mbit/s. Netzwerkverbindungen, die nun über einen Rechner mit dieser eingestellten *pipe* geroutet werden, erfahren eine Drosselung der Bandbreite auf den angegebenen Wert. Damit ermöglicht es dummynet einfach, zur Laufzeit die Bedingungen des Netzwerks von Grund auf neu zu definieren und zu verändern. Mithilfe von *Ping* und dem Messwerkzeug *NetPerfMeter* (siehe 4.3) konnte die Funktionstüchtigkeit der Regeln für die Verzögerung und die Bandbreitenbeschränkung erfolgreich getestet werden.

4.2. Testszenarien

Die Auswahl einer Menge an geeigneten Testszenarien zum Vergleich des CMT/RPv2 erweist sich durch die Vielzahl an zu testenden Möglichkeiten als schwierig. Dies ist zum einen auf die beliebige Paarung an Congestion-Control-Algorithmen und zum anderen auf die zahlreichen Möglichkeiten von Dummynet zurückzuführen. Um in einem geeigneten Zeitrahmen die Messungen abschließen und auswerten zu können, ist es notwendig, genau abzuwägen, welche Paarungen und Konfigurationen für die Auswertung von größerem Interesse sind. Folgende Paarungen wurden für die Messungen zusammengestellt:

- **MPTCP-Verbindung gegen TCP-Verbindung**: Hier werden alle vorgestellten Algorithmen einzeln mit dem Standard TCP-Congestion-Control-Algorithmus Reno für den Linux Kernel getestet. Die Auswahl auf Reno als Haupttestfall wurde vor dem Hintergrund gefällt, dass Reno eines der verbreitetsten Transportprotokolle ist und vielfach Anwendung findet. Da alle Algorithmen dem *Resource Pooling Principle* oder im Fall von wVegas einem ähnlichen Prinzip folgen, müssen die *subflows* gegenüber Reno fair sein. Im Bottleneck-Aufbau der Testumgebung heißt das, dass auf dem *Bottleneck* eine Verteilung der Bandbreite von 50:50 erfolgen muss. Eine Abweichung von dieser Verteilung würde auf eine mangelhafte Unterstützung der implementierten Prinzipien hindeuten. Es zeigt sich außerdem hier, ob die Umsetzung des CMT/RPv2 für den Linux Kernel erfolgreich vonstatten ging oder sich noch Fehler in der Programmierung während der Messung bemerkbar machen.
- **MPTCP-Verbindung gegen sich selbst**: Die Algorithmen werden in der Testumgebung gegen sich selbst getestet. Damit konkurrieren zwei MPTCP-Verbindungen mit jeweils zwei *subflows* um die zur Verfügung stehenden Kapazität. Auch hier ist eine Bandbreitenverteilung von 50:50 im Falle des *Bottlenecks* zu erwarten.

- **TCP-Verbindung gegen zwei MPTCP-Verbindungen**: In diesem Testfall stehen das TCP Reno, das CMT/RPv2 und OLIA in Konkurrenz zueinander. Drei Verbindungen auf einen *Bottleneck* soll hier einen Extremfall darstellen und zeigen, ob unter diesen Umständen noch die Fairness auf dem *Bottleneck* aufrecht erhalten werden kann.

Nachdem die Paarungen der Algorithmen festgelegt wurden, steht noch die Auswahl geeigneter Parameter für dummynet aus:

- **Verzögerung**: Jede Messung mit den oben vorgestellten Paarungen wird mit drei unterschiedlichen Verzögerungen durchgeführt. Diese betragen weniger als 1ms, für keine zusätzliche Verzögerung, 5ms, welches ungefähr der Verzögerung in einem LAN entspricht und 20ms für die Verzögerung in einem WAN. Dabei ist die reale Verzögerung der Übertragung nicht inbegriffen. Damit soll untersucht werden, wie stark sich die zunehmende Verzögerung auf die Ausnutzung der Kapazität auswirkt.

- **Kapazität**: Es werden sowohl symmetrische als auch asymmetrische Kapazitäten getestet. Der Raum dafür erstreckt sich in den Messungen von 10 Mbit/s auf 100 Mbit/s in 10er-Schritten. Damit können über verschiedene Kapazitäten hinweg die Bandbreiten der Verbindungen überprüft und Probleme im Anstiegverhalten ausfindig gemacht werden. Im synchronen Fall werden die Kapazitäten von beiden Pfaden auf den selben Wert gesetzt während im asynchronen Fall ein Pfad auf 10 Mbit/s limitiert und der andere schrittweise von 10 Mbit/s auf 100 Mbit/s erhöht wird, um die Asymmetrie immer größer werden zu lassen.

- **Queue-Management**: Dummynet erlaubt es beim Queue-Management in den Routern zwischen mehreren Arten zu wählen. In den Testszenarien werden die folgenden zwei berücksichtigt: Die Tail-Drop-Technik und das *active queue management* RED (Random Early Detection) [31]. Ersteres verwirft eintreffende Pakete am Router, sobald die Kapazität der Queue ausgeschöpft ist. Daraus resultieren nach [31] folgende Nachteile:
  - Die Router erhalten einen vollen oder fast vollen Buffer über eine längere Zeitdauer. Trifft ein burst-artiger Zustrom an Paketen ein, so besitzen diese nicht genügend Kapazitäten, um die Spitze aufzufangen. Es kommt über mehrere Verbindungen hinweg zu Paketverlusten und damit zu einer Synchronisation der Übertragungsrate. Diese Synchronisation mehrerer Verbindungen gleichzeitig führt zu einer ungenügenden Ausnutzung der Kapazität, da die betroffenen Verbindungen gleichzeitig ihr *Congestion Window* reduzieren.

- Zusätzlich als Folge einer Synchronisierung einiger Verbindungen durch gemeinsame Paketverluste, welche durch einen vollen Buffer verursacht wurde, entsteht in einigen Situationen ein Monopol einiger Verbindungen auf die Kapazität des Buffers. Diese Situation wird auch als *Lock-Out* bezeichnet.

Der RED-Algorithmus soll diesen Nachteilen entgegenwirken, indem es eintreffende Pakete am Router bereits vor Erreichen der Kapazitätsgrenze verwirft. Mit zunehmendem Grad der Füllung des Buffers erhöht sich die Wahrscheinlichkeit, dass ein Paket verworfen wird. Das soll bewirken, dass die Verbindungen vor Eintreten einer möglichen Überlast ihre Übertragungsrate verringern, um diese zu verhindern. Dieser Ansatz wird *active queue management* genannt.

Die Messungen der Paarungen werden einzeln mit der jeweiligen Dummynet-Konfiguration durchgeführt. Es wird sowohl im Bottleneck- als auch im Half-Bottleneck-Aufbau gemessen. Dies hat den Sinn, dass nur im Bottleneck-Fall eine klare Aussage darüber getroffen werden kann, wie sich die Bandbreite innerhalb dessen aufteilt und ob die ersten beiden Prinzipien des *Resource Poolings* funktionieren. Im Half-Bottleneck-Aufbau dagegen können keine klaren Annahmen zur Verteilung getroffen werden. Dafür erlaubt der Aufbau, das verbliebene dritte Prinzip des *Resource Poolings* auf seine Funktionsweise zu überprüfen. Am Ende der Messungen kann mit dieser Vorgehensweise genau nachgewiesen werden, wie gut die Prinzipien umgesetzt wurden.

4.3. Messung und Darstellung

Die Messung stellt ein zentrales Thema in dieser Arbeit dar. Sie dient zur Verifikation der Funktionstüchtigkeit des CMT/RPv2 und erlaubt Vergleiche mit anderen Algorithmen im selben Testszenario. Das verwendete Werkzeug zur Messung ist der *Network Performance Meter* (kurz: *NetPerfMeter*) [1]. Mithilfe des *NetPerfMeters* wird eine Verbindung zu einem Endpunkt erstellt, über den Testdaten übertragen werden. Gleichzeitig findet eine Messung der resultierenden Übertragungsrate statt. Die Übertragung kann über die Transportprotokolle TCP inklusive MPTCP, UDP, SCTP und DCCP stattfinden. Es ist auch möglich, mehrere Messungen mit unterschiedlichen Transportprotokollen durchzuführen. Am Ende einer Messung können die Ergebnisse in eine Vektor- oder Skalar-Datei geschrieben werden. Diese unterscheiden sich darin, dass die Skalar-Datei pro Messgröße nur einen Wert über die gesamte Dauer der Messung hinweg speichert, während es in der Vektor-Datei mehrere Werte über die Zeit sind.

[1] https://www.uni-due.de/ be0001/netperfmeter/

Um aus den Messungen aussagekräftige Diagramme erstellen zu können, wird aus der Skalar-Datei, die beim *NetPerfMeter* verschiedene Übertragungsgrößen und -zeiten dokumentiert, nur die Bitrate benötigt. Jedes Testszenario wird 10 Mal mit unterschiedlichen Bandbreiten gemessen, welche sich von 10 Mbit/s auf 100 Mbit/s erstrecken. Somit steht für jede Verbindung ein Wert für die Bitrate pro eingestellter Kapazität bereit. Diese Darstellungsart hat den Vorteil, dass auf den ersten Blick die Bandbreitenverteilung ersichtlich ist und so eine Beurteilung zur Fairness oder Kapazitätsausnutzung einer Verbindung leichter fällt.

Die Dauer einer Messung beträgt fünf Minuten, um den Einfluss kurzzeitiger Schwankungen in der Übertragungsrate zu vermindern. Zusätzlich wird jede Messung fünf Mal durchgeführt. Die Gesamtdauer einer Messung eines Testszenarios mit einer Dummynet-Konfiguration beläuft sich damit auf fünf Stunden. Mithilfe eines eigenen Shell-Skripts werden die Werte für die Bitrate nach der Messung aus den Skalar-Dateien extrahiert und aus diesen fünf Werten der Median bestimmt, welcher im Diagramm abgebildet wird. Um einen Eindruck über die Streuung der einzelnen Werte zu bekommen, wird zusätzlich die Standardabweichung dieser berechnet.

Da Messungen ständig unter dem Einfluss bestimmter Größen stehen, die Abweichungen vom wahren Ergebnis hervorrufen, ist ein Maß für die Zuverlässigkeit der Ergebnisse sinnvoll. Für diese Fälle bietet sich die Berechnung eines Konfidenzintervalls an. Mit den fünf Messungen eines Messpunkts und der anschließenden Bestimmung des Medians ist nur eine kleine Stichprobe von unendlich vielen Messungen abgedeckt worden. Der Erwartungswert kann trotzdem stark vom ausgewählten Median abweichen. Das Konfidenzintervall hilft hier, einen Intervall mit einem vorher festgelegten Konfidenzniveau zu berechnen. Dieses Intervall beinhaltet mit der Wahrscheinlichkeit des Konfidenzniveaus den wahren Median der Messung. Die Intervalle für die Diagramme wurden mit einem Konfidenzniveau von 95% berechnet. Das bedeutet, dass zu 95% das abgebildete Intervall den wahren Wert der Medians beherbergt.

Zur Erstellung der Diagramme wird das Programm Gnuplot[2] verwendet. In einem eigenen Skript (siehe CD im Anhang) nimmt Gnuplot die Punkte mit den Abweichungen und den Konfidenzintervallen entgegen und stellt diese gemeinsam dar.

[2] http://gnuplot.info/

4.4. Vergleichskriterien

Nach der Vorstellung der Testszenarien und des Ablaufs der Messung erfolgt die Suche nach geeigneten Kriterien. Diese müssen ein aussagekräftiges Merkmal besitzen, damit die Ergebnisse der Messungen bewertet und miteinander verglichen werden können. Letztendlich haben sich genau zwei Kriterien hevorgetan, die den Anforderungen genügen und einen großen Bereich an zur Verfügung stehenden Merkmalen in sich vereinen:

- **Fairness**: Ein gewisser Teil dieser Arbeit lag darin, dass Bottleneck-Problem (2.2) zu beschreiben und eine Lösung dafür vorzustellen. Mithilfe des zweiten Setups der Testumgebung ist es möglich, ein solches *Bottleneck* nachzubilden. Da bei einem *Bottleneck* unabhängig von der Anzahl der *subflows* alle Verbindungen einen fairen Anteil der Kapazität des *Bottlenecks* haben müssen, eignet sich dieser Aufbau ideal, um zu überprüfen, ob die Congestion-Control-Algorithmen in der Praxis wirklich nach den Regeln des *Resource Poolings* agieren. Das bedeutet, im Idealfall besitzen n Verbindungen einen Anteil von $\frac{1}{n}$ der zur Verfügung stehenden Kapazität. Zeigt sich dagegen eine Abweichung vom Soll-Zustand, so verstößt der Algorithmus gegen die Vorgaben der *TCP Friendly Rate Control* [11].

- **Bandbreite**: Bei aller Fairness muss gleichzeitig gewährleistet werden, dass immer die optimale Durchsatzrate für die Datenübertragung anliegt. Ein Nichtausnutzen frei verfügbarer Kapazität resultiert in einer geringen Effizienz und ein zu starker Anstieg des *Congestion Windows* führt zu schnellen Überläufen mit einhergehenden Paketverlusten aufseiten der Router und der Verdrängung anderer Verbindungen auf dem selben Pfad.

5. Ergebnisse der Messungen

Dieses Kapitel präsentiert und diskutiert die Ergebnisse der Messungen. Im Einzelnen wird betrachtet, wie die Implementierung des CMT/RPv2 im Vergleich zu den anderen Congestion-Control-Algorithmen abgeschnitten hat, ob es bei den Messungen Abweichungen von der Erwartungshaltung gab und untersucht, wo die Ursache bei möglichen Abweichungen liegt. Zuletzt findet eine Bewertung statt, in der die Leistung der Algorithmen eingeordnet wird. Diagramme, die keine neuen Erkenntnisgewinne bringen, werden im Anhang hinterlegt.

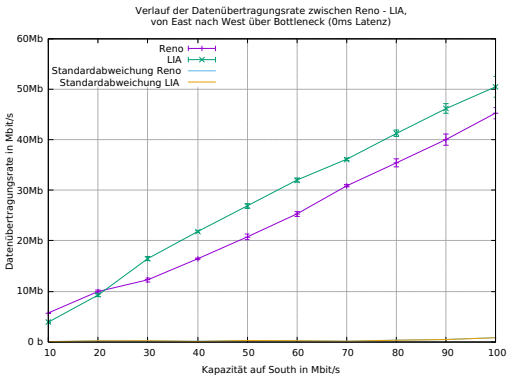
5.1. Ergebnisvorstellung

Die folgenden Messungen wurden auf dem Bottleneck-Aufbau der Testumgebung durchgeführt. Es soll überprüft werden, wie sich die MPTCP-Algorithmen am *Bottleneck* verhalten und ob die ersten beiden Prinzipien des *Resource Poolings* ihre Anwendung finden.

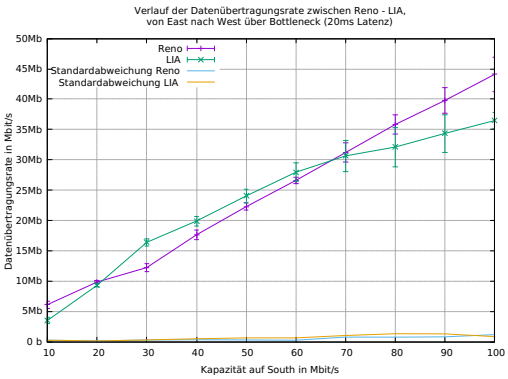
Zu beachten ist, dass die Diagramme der Messungen unterschiedlichen Skalierungen aufweisen. Neben der Diskussion der Unterschiede zwischen den Messergebnissen am Ende eines Testszenarios, sind zusätzlich im Anhang Tabellen hinterlegt worden, die an bestimmten Kapazitäten die Übertragungsrate auflisten, um diese besser gegenüber stellen zu können.

Eine MPTCP-Verbindung gegen eine TCP-Verbindung

Reno gegen LIA



(a) Reno gegen LIA ohne zusätzliche Latenz

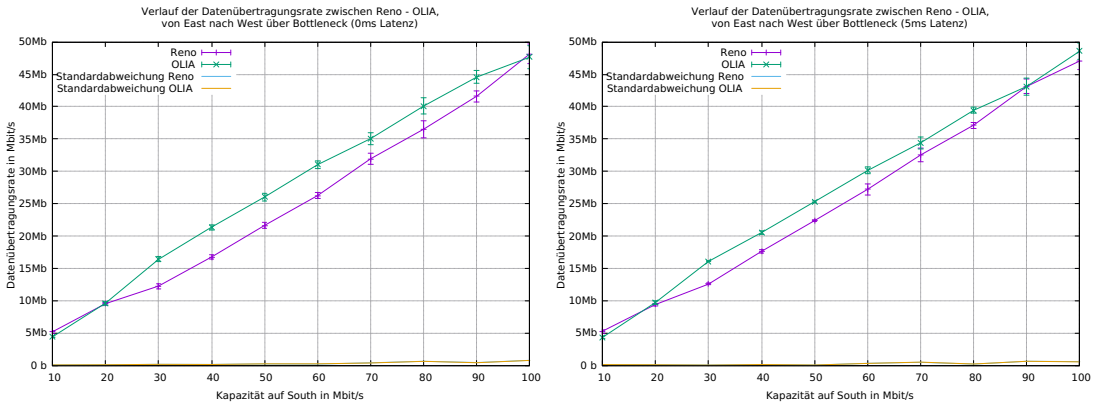


(b) Reno gegen LIA mit 20ms Latenz

Abbildung 5.1.: Reno gegen LIA auf dem *Bottleneck*

Im Bottleneck-Fall ist die *Congestion Control* LIA in 5.1a gegenüber Reno ab 20 Mbit/s leicht überlegen und nimmt bis zu 2 Mbit/s mehr in Anspruch, als dem Algorithmus eigentlich zustehen dürfte. Diese zusätzliche Bandbreite bei LIA fehlt der TCP-Verbindung, um den gerechten Anteil der Kapazität zu erhalten. Die Abweichung während der gesamten Messung ist sehr klein, sodass diese kaum Einfluss auf das Ergebnis hat. Eine Latenz von 20ms bewirkt, dass ab 70 Mbit/s eine Umkehr der Verhältnisse zu beobachten ist. Während Reno quasi linear weiter etwas weniger als die ihm zustehende Kapazität einnimmt, ist bei LIA eine deutliche Reduzierung zu sehen. Dieses Verhalten ist auf die Zunahme des Bandbreiten-Delay-Produkts und der fehlenden Adaption von LIA auf ein größeres *Congestion Window* zurückzuführen. Mit einer eingestellten Latenz von 5ms sind kaum Unterschiede zu einer Latenz von kleiner als 1ms wahrzunehmen, weshalb die Abbildung hier ausgelassen wurde.

Reno gegen OLIA



(a) Reno gegen OLIA ohne zusätzliche Latenz (b) Reno gegen OLIA mit 5ms Latenz

Abbildung 5.2.: Reno gegen OLIA auf dem *Bottleneck*

In den Abbildungen 5.2 ähnelt der Verlauf der Bandbreitenverteilung dem von LIA. Interessant ist hier die letzte Messung bei 100 Mbit/s in 5.2a. Hier treffen beide Algorithmen aufeinander und teilen sich fair jeweils knapp 50 Mbit/s. Es stellt sich hier die Frage, ob dieser Zustand auch bei 110 Mbit/s beibehalten werden würde oder ob wie in Abbildung 5.1b eine Reduzierung zu beobachten wäre. Eine Erhöhung der Latenz auf 5ms zeigt, dass sich die Verhältnisse der Bandbreite verbessern. OLIA nimmt weniger Bandbreite in Anspruch, welche dann an Reno übergeht.

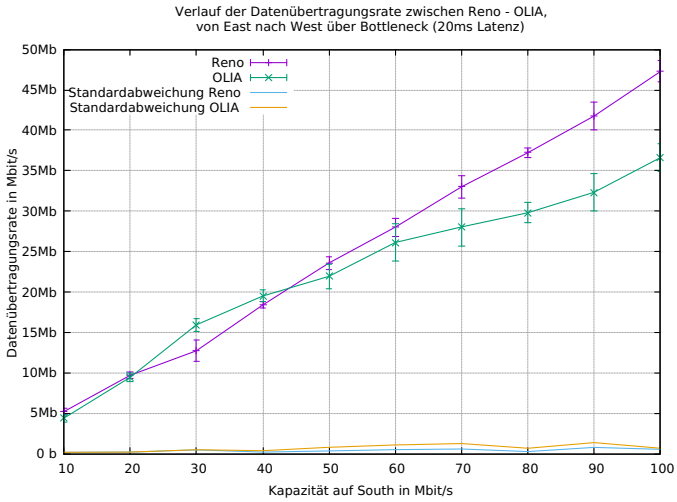
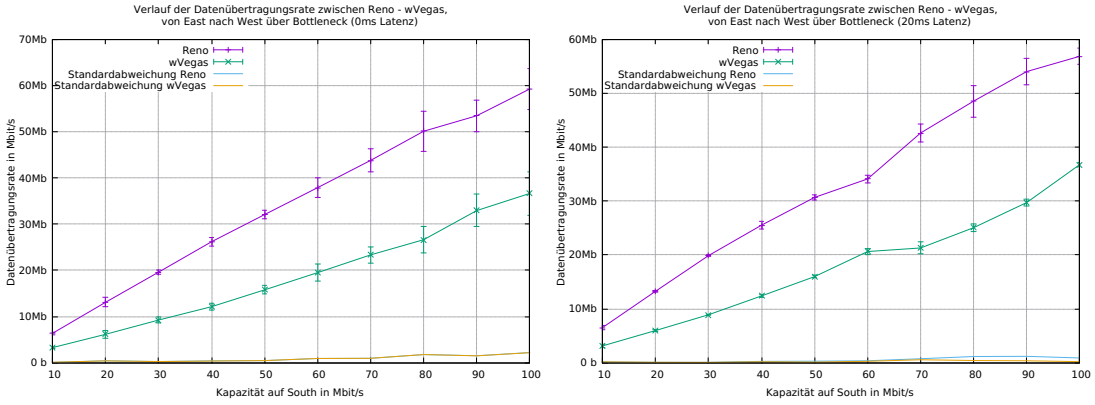


Abbildung 5.3.: Reno gegen OLIA mit 20ms Latenz

Bei einer Latenz von 20ms zeigt sich eine deutliche Verschlechterung von OLIA. Ab ungefähr 44 Mbit/s ist eine deutliche Abnahme der Steigungsrate zu beobachten, welche sich mit zunehmender Kapazität erhöht. Dieses Bild ist ähnlich zu Abbildung 5.1b, nur mit dem Unterschied, dass OLIA bei einer geringeren Kapazität bereits die Sendeleistung reduziert.

Reno gegen wVegas

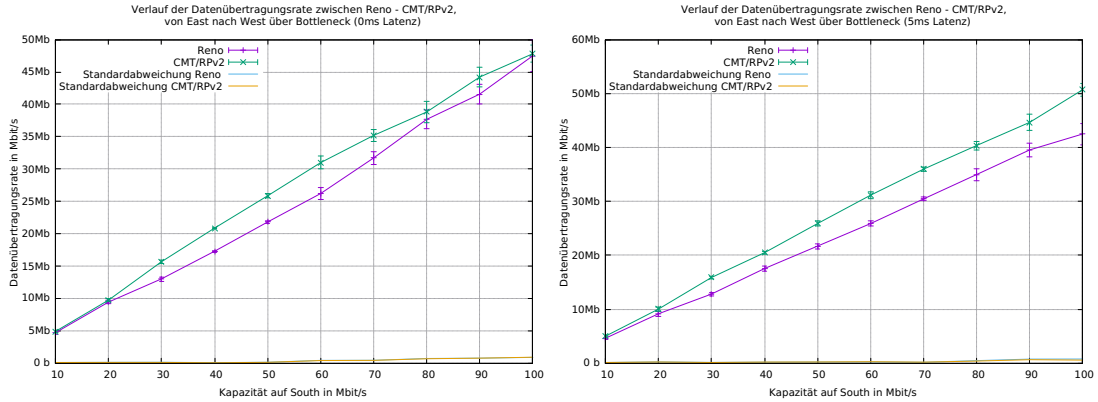


(a) Reno gegen wVegas ohne zusätzliche Latenz (b) Reno gegen wVegas mit 20ms Latenz

Abbildung 5.4.: Reno gegen wVegas auf dem *Bottleneck*

Das wVegas ist wie in 3.2.2.1 vorgestellt eine delay-based *Congestion Control*. Auch wurde der Fall betrachtet, bei dem eine loss-based und eine delay-based *Congestion Control* aufeinandertreffen. Diese Paarung resultiert darin, dass die delay-based *Congestion Control* durch ihre Überlastungsfrüherkennung das *Congestion Window* wesentlich früher reduziert, was auch hier in den Diagrammen zu beobachten ist. Bei allen getesteten Latenzen unterliegt die Bandbreite von wVegas deutlich der von TCP Reno.

Reno gegen CMT/RPv2



(a) Reno gegen CMT/RPv2 ohne zusätzliche Latenz (b) Reno gegen CMT/RPv2 mit 5ms Latenz

Abbildung 5.5.: Reno gegen CMT/RPv2 auf dem *Bottleneck*

Im ersten Testszenario des CMT/RPv2 zeigt sich bei einer Latenz von <1ms kein Unterschied zur Messung der loss-based Algorithmen LIA und OLIA bei der selben Konfiguration. Auch das CMT/RPv2 nimmt etwas mehr Bandbreite Anspruch als die TCP-Verbindung. Jedoch ist wie bei den anderen Messungen auch die Tendenz zur Fairness zu beobachten. Bei 5ms bleibt das Verhältnis der Bandbreite weiter stabil.

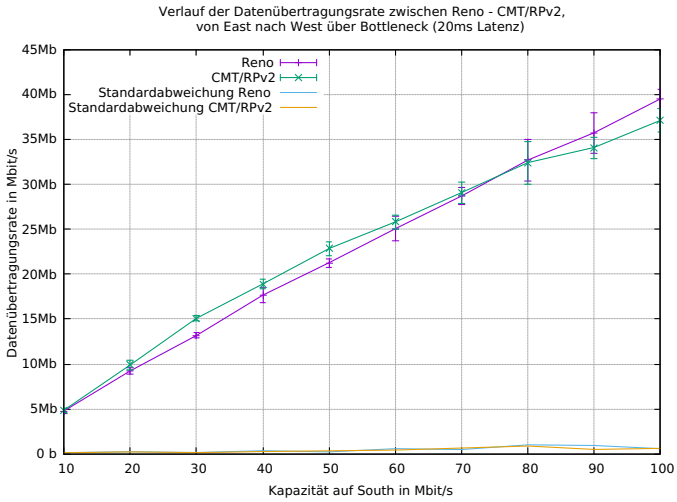


Abbildung 5.6.: Reno gegen CMT/RPv2 mit 20ms Latenz

Das CMT/RPv2 präsentiert bei der höchsten Latenz einen ruhigen Bandbreitenverlauf. Jedoch ist die Effizienz beider Verbindungen stark beeinträchtigt. Während das CMT/RPv2 bei einer Latenz von 5ms noch einen maximalen Durchsatz von 50 Mbit/s erzielen konnte, so sind es bei 20ms nur noch 37 Mbit/s. Die TCP-Verbindung bleibt weitgehend gleich. Nur gegen Ende macht sich auch bei Reno das höhere Bandbreiten-Delay-Produkt durch einen verringerten Durchsatz bemerkbar.

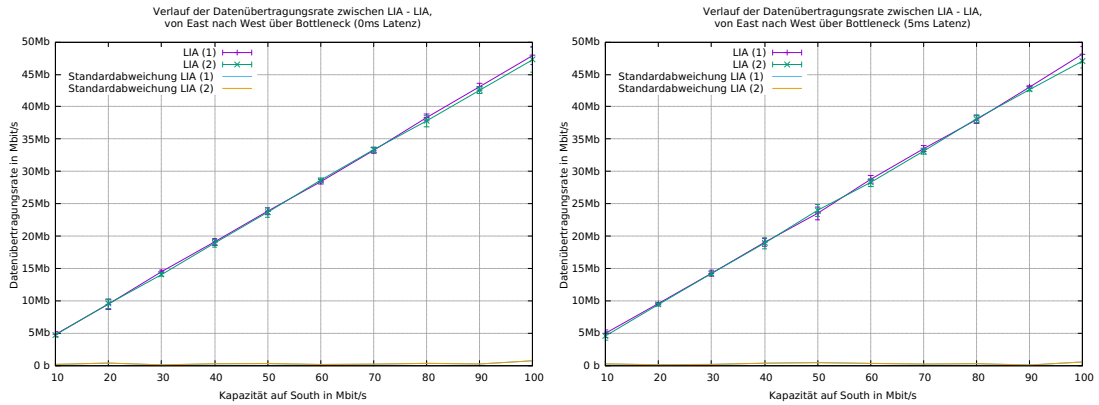
Auffällig bei den vorgestellten Messungen ist, dass die Algorithmen trotz Umsetzung der ersten beiden Prinzipien des *Resource Poolings*, welche das Bottleneck-Problem behandeln, keine exakte Fairness auf diesem erzielen, sondern durchweg etwas mehr Bandbreite in Anspruch nehmen (das wVegas ist hier ausgenommen). Die Ursache für diese Abweichung ist im Falle von LIA das statistische Multiplexing [32] (siehe Kapitel 3.1.1). Dieses beschreibt die Nutzung eines gemeinsamen Mediums durch mehrere Verbindungen. Teilen sich viele Verbindungen ein Medium (hohes statistisches Multiplexing), so hat eine MPTCP-Verbindung wenig Einfluss auf auftretende Paketverluste und nimmt einen Kapazitätsanteil von $\frac{1}{n}$ bei n Verbindungen ein. Verursacht dagegen eine MPTCP-Verbindung Paketverluste bei wenigen Verbindungen auf dem Medium (niedriges statistisches Multiplexing), so wird der Durchsatz der MPTCP-Verbindung strikt höher sein als bei anderen Verbindungen. Mit nur einer zusätzlichen TCP-Verbindung besteht in diesen Testszenarien ein niedriges statistisches Multiplexing, weshalb letztendlich eine etwas höhere Bandbreite im Vergleich zur TCP-Verbindung resultiert. Die Algorithmen

OLIA und CMT/RPv2 verfügen dagegen ebenso wie LIA über eine höhere Bandbreite, obwohl dieses Verhalten in den Ausarbeitungen ([6], [16]) nicht beschrieben wurde. Die Ursachen dafür können vielfältig sein, dem wird in dieser Arbeit jedoch nicht weiter nachgegangen.

Insbesondere bei einem hohen Bandbreiten-Delay-Produkt ist aufgefallen, dass die Steigung der Durchsatzrate der MPTCP-Verbindungen nachlässt. Ein hohes Bandbreiten-Delay-Produkt erfordert ein ausreichend großes *Congestion Window*, um die Kapazität des Verbindungspfades optimal ausnutzen zu können. Nur die TCP-Verbindung kann trotz der Zunahme der Kapazität bei einer Latenz von 20ms beinahe linear die Steigung der eigenen Bandbreite fortführen.

Eine MPTCP-Verbindung gegen eine andere MPTCP-Verbindung

LIA gegen LIA



(a) LIA gegen LIA ohne zusätzliche Latenz

(b) LIA gegen LIA mit 5ms Latenz

Abbildung 5.7.: LIA gegen LIA auf dem *Bottleneck*

Sogleich im ersten Testszenario mit zwei MPTCP-Verbindungen zeigt LIA ein Musterbeispiel für die Fairness am *Bottleneck* gegenüber sich selbst. Bei einer Latenz von <1ms als auch 5ms ist das Verhältnis der Bandbreite bei annähernd 50:50. Damit wird die gesamte Kapazität des *Bottlenecks* mit Ausnahme kleinerer Abweichungen von der eingestellten Kapazität durch die Protokolleffizienz effektiv ausgenutzt.

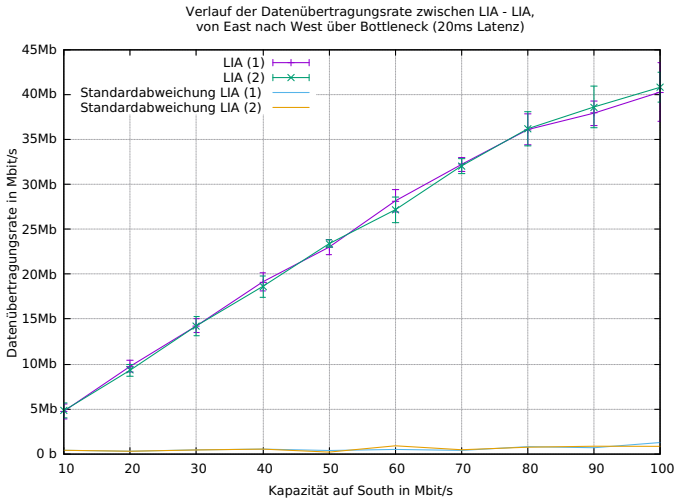
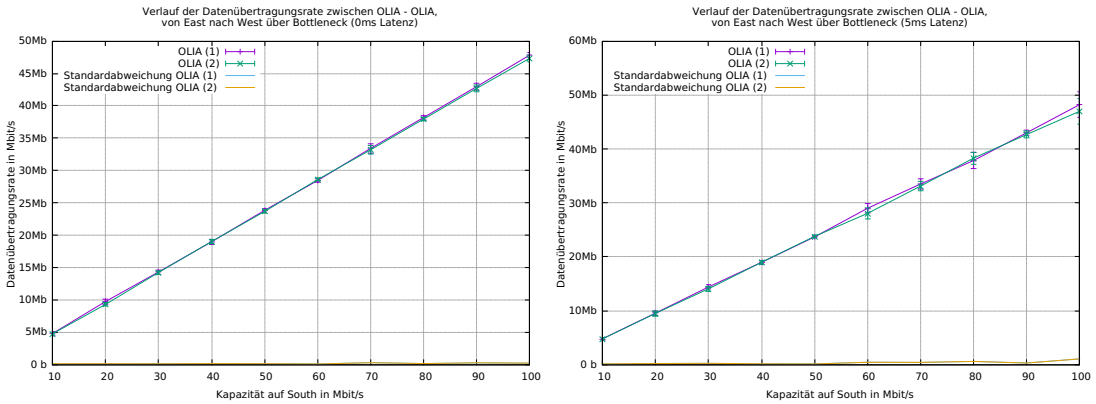


Abbildung 5.8.: LIA gegen LIA mit 20ms Latenz

Bei einer Latenz von 20ms sind beide Verbindungen über alle Kapazitäten hinweg weiter sehr nahe beieinander. Es fällt auf, dass ab ca. 80 Mbit/s die Kapazität nicht mehr so effektiv ausgenutzt wird. Jedoch ist die Ausnutzung der Kapazität bei einem höheren Bandbreiten-Delay-Produkt wesentlich besser als zuvor in allen Messungen mit der TCP-Verbindung und der Latenz von 20ms.

OLIA gegen OLIA



(a) OLIA gegen OLIA ohne zusätzliche Latenz

(b) OLIA gegen OLIA mit 5ms Latenz

Abbildung 5.9.: OLIA gegen OLIA auf dem *Bottleneck*

Das Szenario OLIA gegen OLIA auf dem *Bottleneck* unterscheidet sich nicht von der vorherigen Messung. Auch hier wird die Kapazität abgesehen von den Abweichungen mustergültig aufgeteilt. Die Abbildung mit 20ms Latenz wurde ausgelassen, da es bis auf kleinere Abweichungen identisch mit der LIA-Messung in Abbildung 5.8 ist.

wVegas gegen wVegas

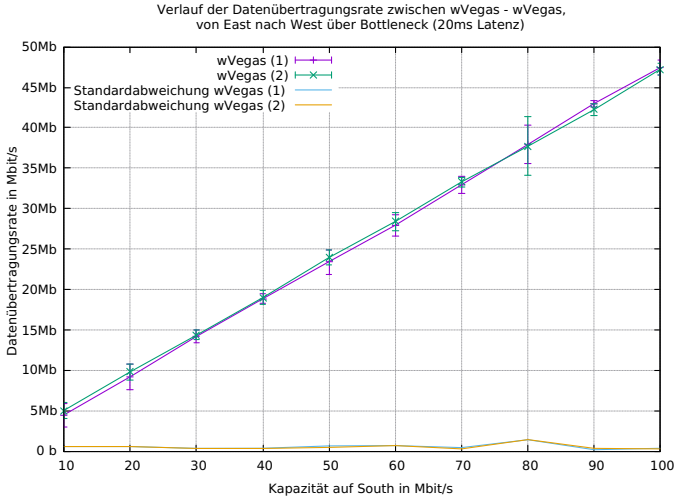
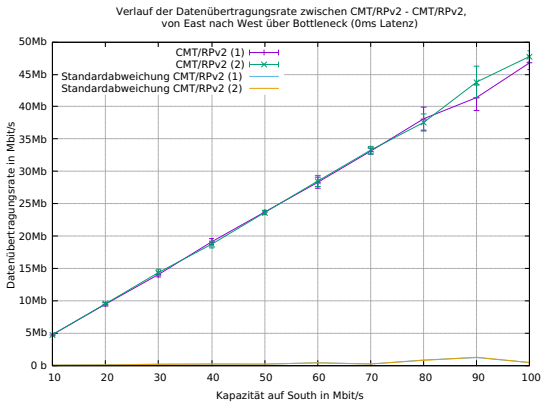


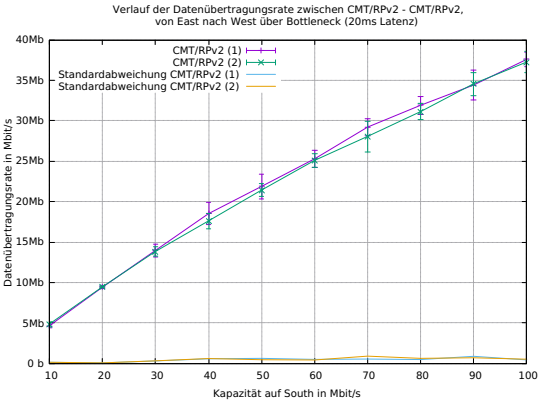
Abbildung 5.10.: wVegas gegen wVegas mit 20ms Latenz

Die Messung mit <1ms und 5ms zeigt auch hier keine Unterschiede, weshalb diese hier nicht abgebildet werden. Besonders ist dagegen das Ergebnis bei 20ms Latenz. Der Verlauf der Bandbreite beider Verbindungen ist trotz des zunehmenden Bandbreiten-Delay-Produkts durchgehend linear. Die in Abbildung 5.8 erstmals erwähnte Reduzierung der Zunahme der Sendeleistung ab einer höheren Kapazität ist hier nicht zu beobachten. Dieser Verlauf ist darauf zurückzuführen, dass beide delay-based Congestion-Control-Algorithmen, ohne Störung einer loss-based Congestion Control, ihre Sendeleistung über die Überlastungsfrüherkennung regulieren können. Daraus resultieren weniger Paketverluste, die sonst immer eine starke Reduzierung des *Congestion Windows* mit sich bringen würden.

CMT/RPv2 gegen CMT/RPv2



(a) CMT/RPv2 gegen CMT/RPv2 ohne zusätzliche Latenz



(b) CMT/RPv2 gegen CMT/RPv2 mit 20ms Latenz

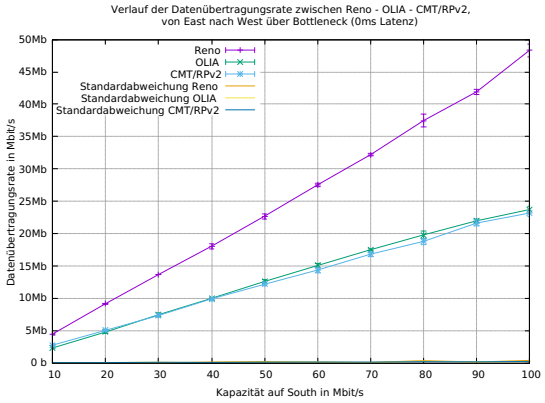
Abbildung 5.11.: CMT/RPv2 gegen CMT/RPv2 auf dem *Bottleneck*

Die Paarung CMT/RPv2 gegen sich selbst ist bei einer Latenz von <1ms und 5ms identisch zu den Messungen von LIA und OLIA zuvor. Liegt am *Bottleneck* eine Latenz von 20ms an, so sieht der Verlauf auf den ersten Blick genauso aus wie die vorherigen loss-based Algorithmen in diesem Szenario. Es zeigt sich jedoch, dass bereits ab 30 Mbit/s das CMT/RPv2 um bis zu 5 Mbit/s (bei einer Kapazität von 70 Mbit/s) unter den Messungen von LIA und OLIA gegen sich selbst liegt. Insbesondere die Effizienz weist bei einer Kapazität von 100Mbit/s nur eine Ausnutzung von 75% auf. Dieses Resultat ist darauf zurückzuführen, dass das CMT/RPv2 bei einem hohen Bandbreiten-Delay-Produkt das *Congestion Window* nicht ausreichend erhöhen oder die neu hinzugekommene Kapazität nicht erkennen kann.

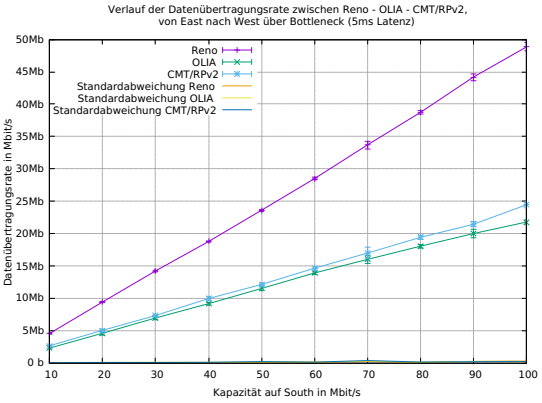
Es bleibt festzuhalten, dass die Congestion-Control-Algorithmen für MPTCP gegen sich selbst in allen Messungen durchweg fair zueinander sind. Nur mit der Latenz von 20ms gab es, mit Ausnahme des wVegas, leichte Einbrüche, welche auf Schwierigkeiten der Algorithmen mit einem höheren Bandbreiten-Delay-Produkt hindeuten.

Zwei MPTCP-Verbindungen gegen eine TCP-Verbindung

Reno gegen OLIA und CMT/RPv2



(a) Reno gegen OLIA und CMT/RPv2 ohne zusätzliche Latenz



(b) Reno gegen OLIA und CMT/RPv2 mit 5ms Latenz

Abbildung 5.12.: Reno gegen OLIA und CMT/RPv2 auf dem *Bottleneck*

Im extremen Szenario zwischen zwei MPTCP-Verbindungen und einer TCP-Verbindung fällt auf, dass beide MPTCP-Verbindungen zusammen nur die Hälfte der Bandbreite der TCP-Verbindung in einem Verhältnis von fast 50:50 beanspruchen. Bei insgesamt drei Flows mit fünf Verbindungen wäre ein Anteil von $\frac{1}{3}$ der zur Verfügung stehenden Kapazität zu erwarten. Es stellt sich als schwierig heraus, die genaue Ursache des Problems auszumachen, da die Congestion-Control-Algorithmen alleine mit TCP Reno fast eine faire Aufteilung von 50:50 erreicht haben. Bei einer zunehmenden Latenz nimmt die Bandbreite von OLIA ab, während das CMT/RPv2 stets stabil bleibt.

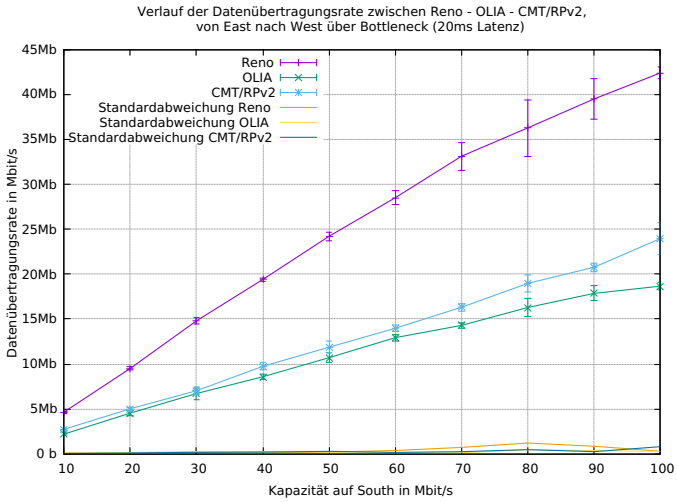


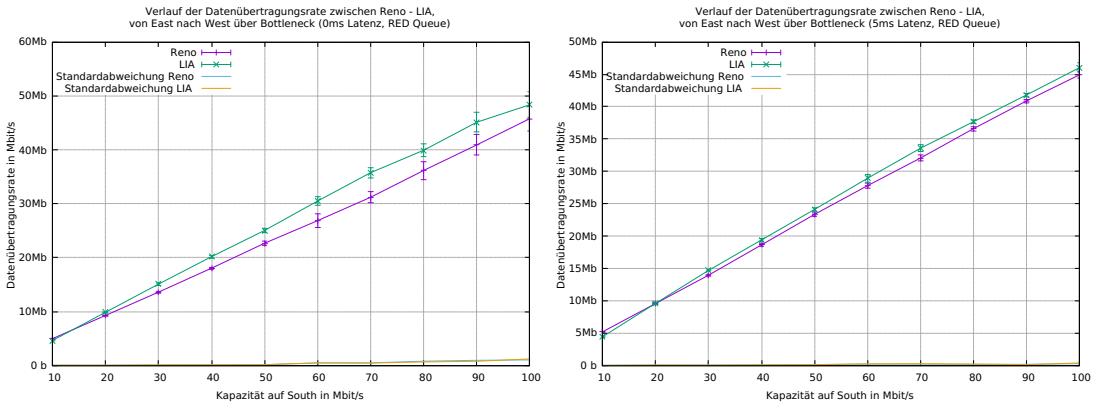
Abbildung 5.13.: Reno gegen OLIA und CMT/RPv2 mit 20ms Latenz

Im Fall mit einer Latenz von 20ms wird deutlich, wie stark sich die Zunahme des Bandbreiten-Delay-Produkts auf alle Verbindungen auswirkt. Der Durchsatz verringert sich hierbei für jede Verbindung um wenige Mbit/s, weshalb die Gesamteffizienz stärker abnimmt. Davon nahezu unberührt ist das CMT/RPv2, wodurch sich die Lücke zwischen den MPTCP-Algorithmen vergrößert.

Eine TCP-Verbindung gegen eine MPTCP-Verbindung mit der RED-Queue

Die folgenden Diagramme bilden weiterhin die Messungen auf dem *Bottleneck* ab. Das Queue-Management wurde hier auf den Algorithmus RED umgestellt, welcher auf einer Queue-Größe von 50 Paketen operiert. Diese Größe entspricht nach [30] dem eines typischen Ethernet-Geräts.

Reno gegen LIA



(a) Reno gegen LIA ohne zusätzliche Latenz

(b) Reno gegen LIA mit 5ms Latenz

Abbildung 5.14.: Reno gegen LIA auf dem *Bottleneck* mit der RED-Queue

Sowohl bei einer Latenz von <1ms als auch bei 5ms, sind die Bandbreitenverläufe beider Verbindungen im Vergleich zum ersten Testszenario mit der Tail-Drop-Queue (5.1) deutlich näher beieinander und entsprechen damit mehr der erwarteten Verteilung auf dem *Bottleneck*. Auffällig ist, dass sich bei 5ms die Verteilung trotz des höheren Bandbreiten-Delay-Produkts deutlich verbessert und beinahe das ideale Verhältnis widerspiegelt.

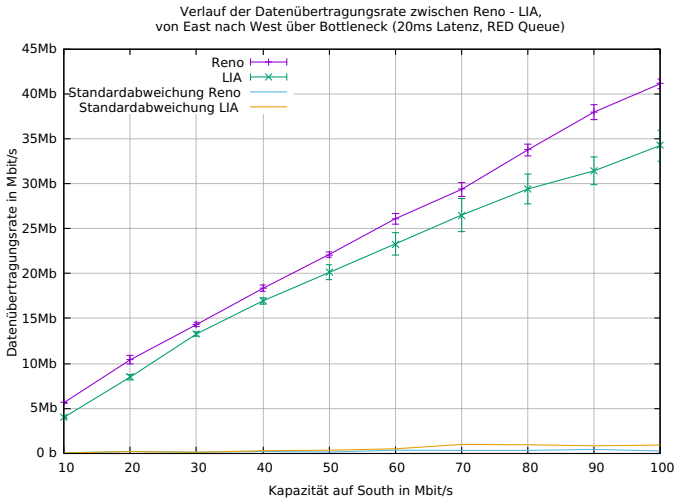
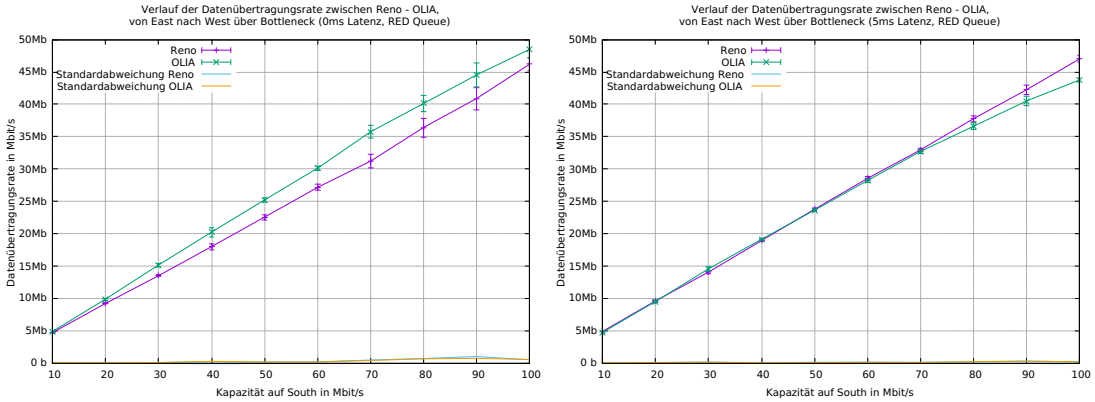


Abbildung 5.15.: Reno gegen LIA mit 20ms Latenz

Die Abbildung 5.15 zeigt, dass LIA über die gesamte Messung hinweg Reno unterliegt. Der Verlauf ist insgesamt stabiler und es kommt zu keinen Umbrüchen wie in 5.1b, jedoch fällt besonders bei einer höheren Latenz auf, dass die Effizienz bei zunehmender Kapazität im Vergleich zur Tail-Drop-Messung schlechter ist. Dies hängt mit dem Merkmal des RED-Algorithmus zusammen, Pakete vor Erreichen der Kapazitätsgrenze zu verwerfen. Damit wird den Algorithmen die Möglichkeit verwehrt, die volle Kapazität am Link ausschöpfen zu können.

Reno gegen OLIA



(a) Reno gegen OLIA ohne zusätzliche Latenz (b) Reno gegen OLIA mit 5ms Latenz

Abbildung 5.16.: Reno gegen OLIA auf dem *Bottleneck* mit der RED-Queue

Im direkten Vergleich zur Messung 5.2a ist zu beobachten, dass bis 60 Mbit/s die Graphen von Reno und OLIA bei der RED-Queue deutlich näher beieinander liegen. Erst danach zeigt sich fast ein identisches Bild wie bei der Tail-Drop-Messung. Nur bei 100 Mbit/s ist auffällig, dass sich bei der RED-Queue-Messung die Tendenz zeigt, weiter linear zu verlaufen, während in 5.2a ein Schnittpunkt zu verzeichnen ist.
5.16b präsentiert bis 70 Mbit/s das erwartete Verhältnis zwischen Reno und OLIA im Gegensatz zur Messung ohne den RED-Algorithmus (5.2b). Bemerkenswert ist die Tendenz von OLIA bei einer höheren Kapazität. Dort ist eine verringerte Steigung der Senderate zu beobachten, die nicht mit einer Erhöhung der Bandbreite bei Reno einhergeht. Das Diagramm 5.2b zeigt dagegen beim Tail-Drop-Algorithmus, dass OLIA weiter mehr Bandbreite als Reno einnimmt.

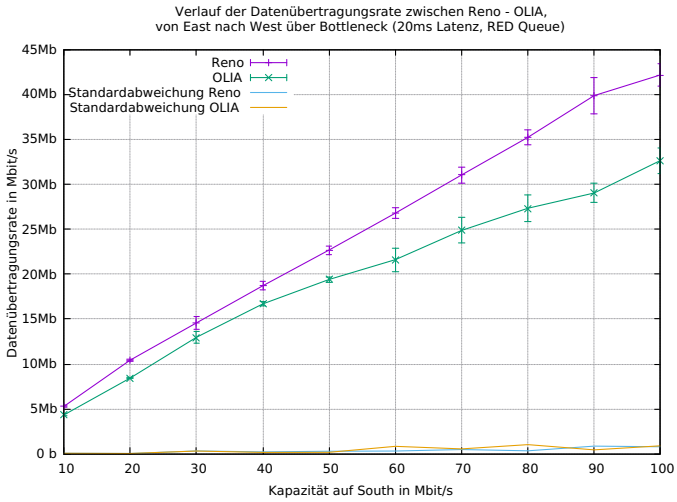
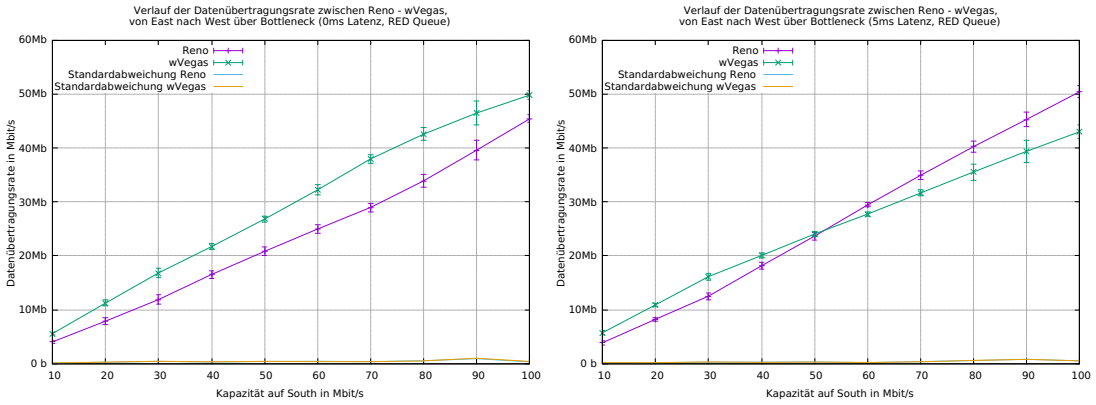


Abbildung 5.17.: Reno gegen OLIA mit 20ms Latenz

In diesem Testszenario mit einer Latenz von 20ms ist erkenntlich, dass OLIA konsequent unter Reno in der Bandbreitenverteilung liegt. Wie auch zuvor bei LIA im 20ms-Latenz-Fall (5.15) ist eine verschlechterte Effizienz bei beiden Verbindungen zu beobachten.

Reno gegen wVegas



(a) Reno gegen wVegas ohne zusätzliche Latenz

(b) Reno gegen wVegas mit 5ms Latenz

Abbildung 5.18.: Reno gegen wVegas auf dem *Bottleneck* mit der RED-Queue

Die größten Auswirkungen des Wechsels des Queue-Managements hat das wVegas zu verzeichnen. Lag die Bandbreite des wVegas vorher deutlich unter Reno (5.4), so nimmt diese nun bis 100

Mbit/s etwas über 50% der zur Verfügung stehenden Kapazität in Anspruch. Dementsprechend weniger als die 50% nimmt die TCP-Verbindung ein. Erhöht sich die Latenz auf 5ms, so agiert wVegas ab etwas über 50 Mbit/s defensiver, aber immer noch deutlich performanter als im Tail-Drop-Fall und stellt Reno die nötige Bandbreite bereit. Begründen kann man die Umkehr der Bandbreitenverhältnisse zur Tail-Drop-Messung in 5.18 damit, dass die TCP-Verbindung einen Paketverlust als ein Zeichen von Überlast deutet und dementsprechend die Senderate stark reduziert, obwohl die RED-Queue am Router theoretisch Kapazität frei hätte. Das wVegas kann mit der RED-Queue wesentlich besser arbeiten, da es nicht Paketverluste als Überlast deutet, sondern rein nach dem *One-Way-Delay* oder der RTT agiert. Dies ermöglicht der delay-based *Congestion Control* eine deutlich bessere Ausnutzung der Kapazität.

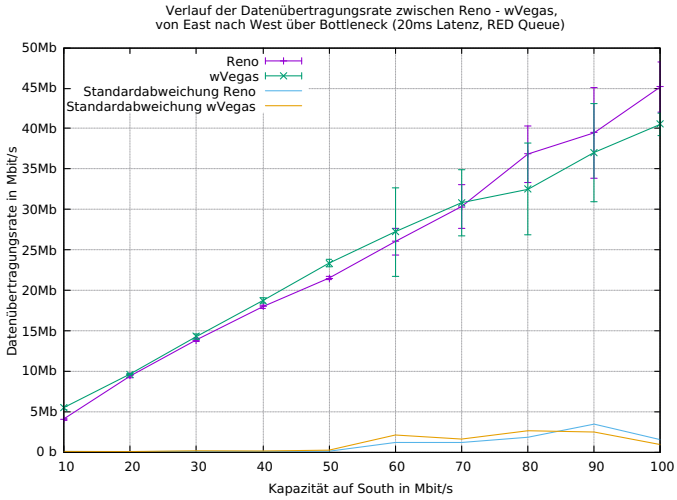
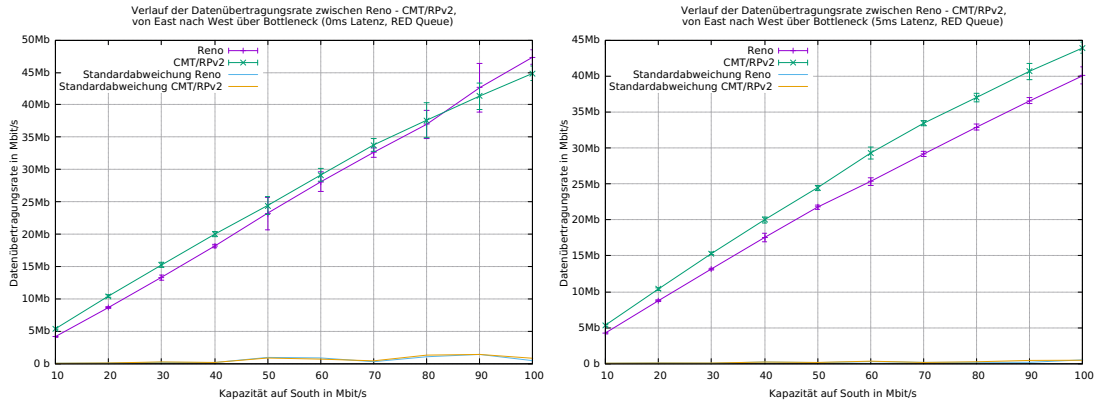


Abbildung 5.19.: Reno gegen wVegas mit 20ms Latenz

Mit einer weiteren Erhöhung der Latenz könnte womöglich mit einem früher beginnenden Schnittpunkt, der die Unterlegenheit des wVegas einleitet, gerechnet werden. Jedoch zeigt sich, dass das wVegas bis 70 Mbit/s gleichauf mit Reno ist. Erst danach geht eine Verringerung der Steigungsrate des Sendefensters bei wVegas mit einer Erhöhung der selben Rate bei Reno einher.

Reno gegen CMT/RPv2



(a) Reno gegen CMT/RPv2 ohne zusätzliche Latenz (b) Reno gegen CMT/RPv2 mit 5ms Latenz

Abbildung 5.20.: Reno gegen CMT/RPv2 auf dem *Bottleneck* mit der RED-Queue

Das Szenario ohne zusätzliche Latenz zeigt ein ähnliches Bild wie die vorherigen Messungen von LIA und OLIA. Im Vergleich zur Messung in 5.5a ist bei der RED-Queue-Messung ein Schnittpunkt bei ungefähr 83 Mbit/s zu erkennen, bei dem das CMT/RPv2 bis zum Ende etwas an Effizenz verliert. Die Erhöhung der Latenz auf 5ms bewirkt, dass das CMT/RPv2 bis ca. 60 Mbit/s nahe an seinem Bandbreitenanteil agiert. Die TCP-Verbindung liefert weniger gute Ergebnisse und verbleibt fast durchgängig unter dem Anteil von 50%.

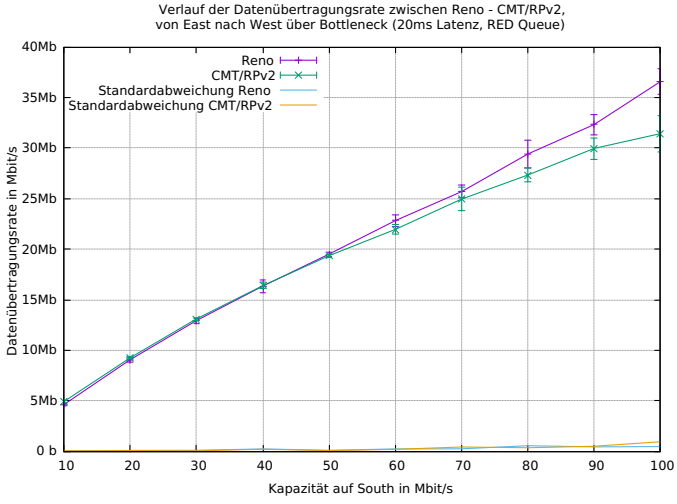


Abbildung 5.21.: Reno gegen CMT/RPv2 mit 20ms Latenz

Eine weitere Erhöhung der Latenz auf 20ms bringt eine Verschlechterung der Gesamtleistung beider Verbindungen im Vergleich zur selben Messung mit dem Tail-Drop-Algorithmus (5.6) und der vorherigen 5ms-Latenz-Messung in Abbildung 5.20b mit. Verglichen mit den anderen Algorithmen im selben Testszenario, weist die Messung mit dem CMT/RPv2 eine deutliche Durchsatzschwäche auf.

Eine TCP-Verbindung gegen eine MPTCP-Verbindung im Half-Bottleneck-Aufbau

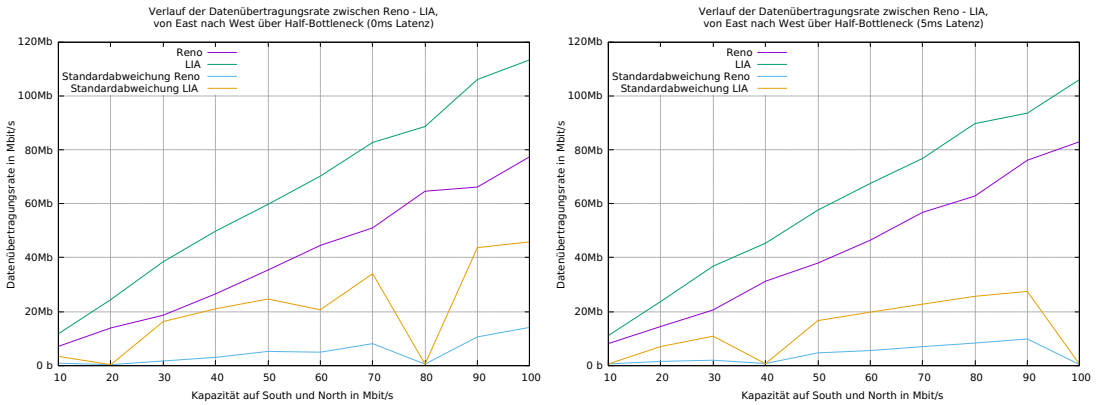
Alle nachfolgenden Messungen der Ergebnisvorstellung wurden ausschließlich auf dem *Half-Bottleneck* durchgeführt. Davon erfolgte der erste Teil auf symmetrischen und der zweite Teil auf asymmetrischen Kapazitäten auf den FreeBSD-Rechnern North und South. Der Hintergrund dieses Aufbaus besteht darin, die Funktionsweise der dritten Regel des *Resource Pooling Principles* (*Balance Congestion*) zu überprüfen. Wenn die TCP-Verbindung mehr als 50% der Kapazität in Anspruch nimmt und die MPTCP-Verbindung dennoch eine insgesamt höhere Bandbreite erzielt, ist dies ein Anzeichen für eine Umleitung des Datenverkehrs, um die Anzahl an Überlastsituationen zu reduzieren.

In den symmetrischen Testszenarien werden die Kapazitäten sowohl auf South als auch auf North schrittweise um 10 Mbit/s erhöht. Die daraus resultierenden Messungen unterliegen stärkeren Schwankungen als auf dem *Bottleneck*. Mit zwei voneinander unabhängigen Pfaden können sich zu unterschiedlichen Zeitpunkten Paketverluste durch die Konkurrenz mit der TCP-Verbindung und mit sich selbst im Hinblick auf die Kapazität ereignen, die eine erhöhte RTT mit sich bringen und so die Entscheidung des RTT-based Subflow-Schedulings [33] beeinflussen. Ein ständiger Wechsel zwischen den *subflows* verringert die Effizienz und kann Bandbreitenschwankungen verursachen. Da die RTT bei beiden Pfaden die selbe ist, existiert kein bevorzugter Pfad in diesem Szenario, was den Effekt der Schwankungen wahrscheinlich begünstigen kann. Aufgrund dessen werden die Bedingungen der Messungen beeinträchtigt, was dazu führt, dass die Messungen weit voneinander streuen und die Aussagekraft des Konfidenzintervalls vermindert wird. Aus diesem Grund wird in den Messungen auf dem *Half-Bottleneck* das Konfidenzintervall nicht abgebildet.

Genauso wie den Konfidenzintervall betrifft die verminderte Aussagekraft zum Teil auch die Messungen selbst, welche stellenweise sehr hohe Standardabweichungen aufweisen und keine genaue Aussage über ihre Richtigkeit erlauben. Vereinzelt kam es dadurch in einigen Messungen zu starken Einbrüchen im Bandbreitenverlauf. Die Messpunkte, die in der Messung

Einbrüche darstellten, wurden ein weiteres Mal wiederholt. Fügte sich der neue Messpunkt besser in den Bandbreitenverlauf ein, so wurde dieser gegenüber dem alten in der Darstellung der Diagramme bevorzugt. Im Folgenden wird versucht, trotz der Abweichungen Unterschiede zwischen den Messungen auszumachen.

Reno gegen LIA



(a) Reno gegen LIA ohne zusätzliche Latenz

(b) Reno gegen LIA mit 5ms Latenz

Abbildung 5.22.: Reno gegen LIA auf dem *Half-Bottleneck*

Das erste Szenario auf dem *Half-Bottleneck* weicht etwas von den bisher bekannten Diagrammen ab. Die Standardabweichung ist bei diesen Messungen stark erhöht, was der großen Streuung der Messungen zueinander geschuldet ist. Über die gesamte Messung hinweg nimmt LIA deutlich mehr Bandbreite in Anspruch, was auf den zusätzlichen Pfad der Testumgebung zurückzuführen ist. Durch die Umleitung der Daten auf dem freien Pfad wird der TCP-Verbindung ermöglicht, mehr als 50% der auf dem südlichen Pfad verfügbaren Kapazität auszunutzen. Der Bandbreitenverlauf ist bei beiden Verbindungen weitestgehend gleichmäßig und weist keine gröberen Ausreißer auf. Steigt die Latenz auf 5ms an, so gibt es kaum Unterschiede im Anstiegverhalten. Die Übertragunsrate von LIA nimmt insgesamt leicht ab, was Reno befähigt, etwas mehr Kapazität zu beanspruchen.

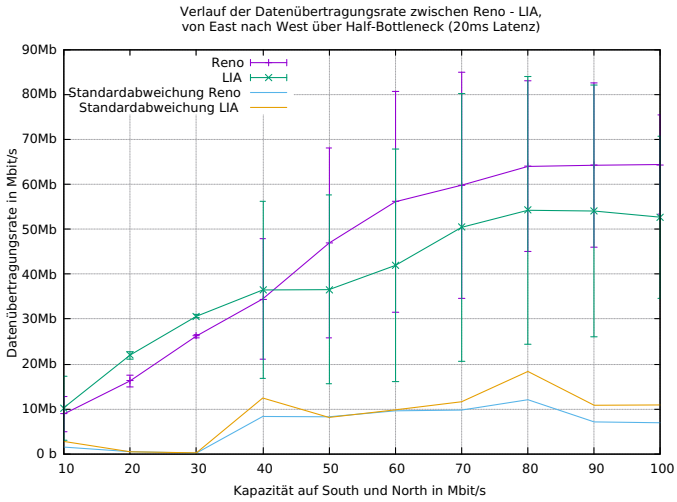


Abbildung 5.23.: Reno gegen LIA mit 20ms Latenz zeigt sehr große Konfidenzintervalle

Wenn mit einer Latenz von 20ms gemessen wird, so fällt ab 40 Mbit/s ein Umbruch zwischen Reno und LIA auf. Im weiteren Bandbreitenverlauf ist eine Stagnation trotz höherer Kapazitäten sichtbar. Als Ursache kommt zuerst ein Flaschenhals bei dummynet selbst in Betracht, da bei den kleineren Latenzen dieses Verhalten nicht zu beobachten war. Die maximale Datenrate, die dummynet bei einer Latenz von 20ms noch bewerkstelligen kann, lässt sich folgenderweise berechnen:

$$50\ Packets * 1500\ Byte * 8 = 600.000\ bit \tag{5.1.1}$$

Bei einer eingestellten Standardkapazität der Queue von 50 Paketen und einer *Maximum Transmission Unit* (MTU) von 1500 Byte kann die Queue insgesamt 600 kbit fassen.

$$\frac{600\ kbit}{20\ ms} = 30\ Mbit/s \tag{5.1.2}$$

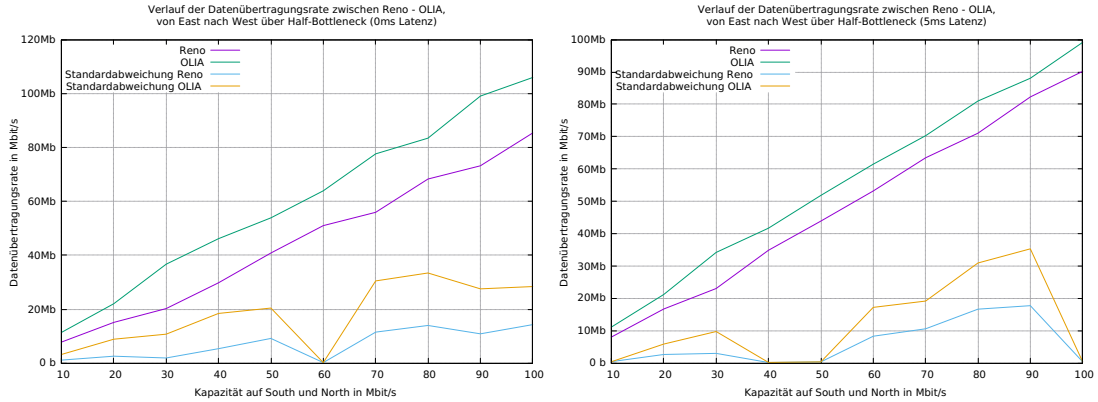
Da die 600 kbit für 20ms verzögert werden müssen, ergibt sich daraus eine maximal zu bewältigende Datenrate von 30 Mbit/s für dummynet.

Das in den Messungen maximal mögliche Bandbreiten-Delay-Produkt für jeden Datenpfad beträgt:

$$100\ Mbit/s * 20\ ms = 2\ Mbit/s \tag{5.1.3}$$

Es wird hiermit deutlich, dass die Kapazität von dummynet für die Verzögerung absolut ausreichend ist und somit nicht die Ursache für die Bandbreitenbeschränkung darstellt. Zum Zeitpunkt dieser Ausarbeitung kann jedoch der genaue Grund dieses Problems nicht geklärt werden. Die dadurch entstehende Verfälschung des Messergebnisses führt dazu, dass alle Messergebnisse für den 20ms-Latenz-Fall mit Ausnahme des asynchronen Falls nicht weiter berücksichtigt und im Anhang hinterlegt werden.

Reno gegen OLIA

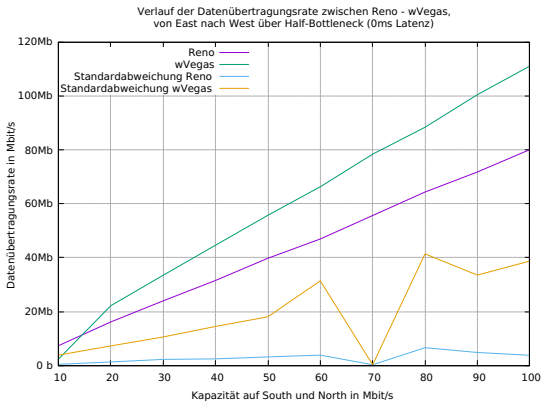


(a) Reno gegen OLIA ohne zusätzliche Latenz (b) Reno gegen OLIA mit 5ms Latenz

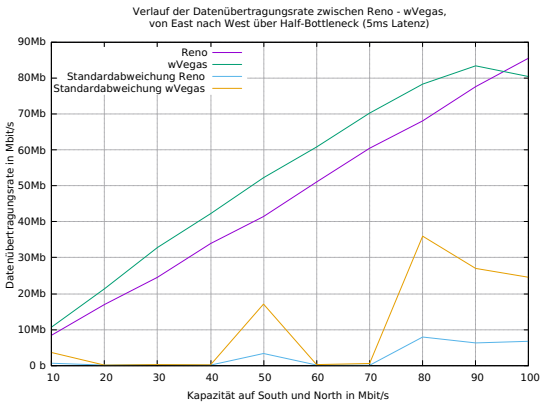
Abbildung 5.24.: Reno gegen OLIA auf dem *Half-Bottleneck*

OLIA weist im Vergleich zur vorherigen Messung in Abbildung 5.22a eine etwas schlechtere Kapazitätsausnutzung aus. Die fehlende Verdrängung von OLIA durch die unzureichende Erhöhung des eigenen *Congestion Windows* führt dazu, dass Reno mehr der zur Verfügung stehenden Kapazität für sich beanspruchen kann. In 5.24b ist zu beobachten, dass die TCP-Verbindung an etwas mehr Bandbreite hinzugewinnt. Diese neu hinzugewonnenen Anteile gehen jedoch auf Kosten der Bandbreite von OLIA. Es findet hier eine Annäherung der Bandbreiten beider Verbindungen statt.

Reno gegen wVegas



(a) Reno gegen wVegas ohne zusätzliche Latenz

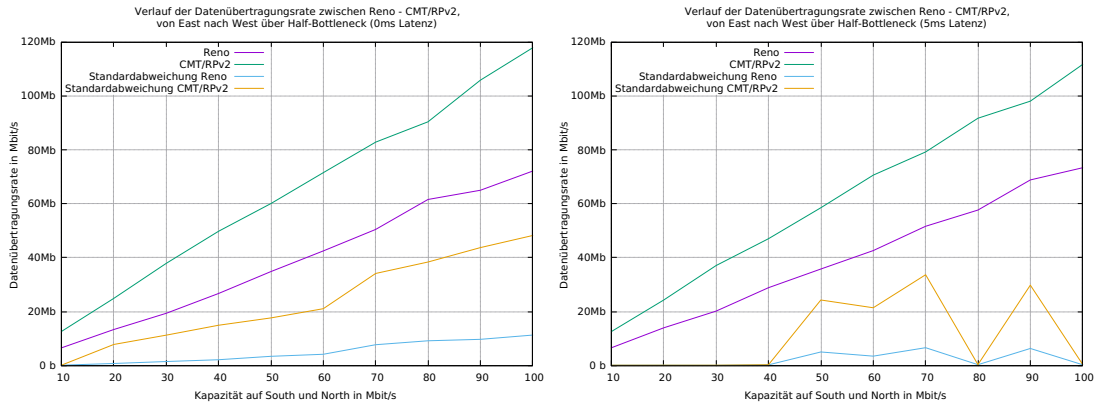


(b) Reno gegen wVegas mit 5ms Latenz

Abbildung 5.25.: Reno gegen wVegas auf dem *Half-Bottleneck*

Die Messung des wVegas ist ohne zusätzlicher Latenz ziemlich stabil und nimmt ein lineares Anstiegverhalten an. Das selbe trifft auch auf die TCP-Verbindung zu, welche durchgängig wVegas unterliegt. Dieser Bandbreitenvorteil des wVegas wird nur durch die Umleitung des Datenverkehrs auf dem freien Pfad ermöglicht, da Reno im direkten Konflikt mit dem wVegas bei einer Kapazität von 100 Mbit/s genau 80 Mbit/s für sich beansprucht. Die nächsthöhere Latenz bewirkt einen großen Abfall der maximalen Bandbreite des wVegas. Lagen vorher 111 Mbit/s bei einer Kapazität von 100 Mbit/s auf beiden Pfaden an, so sind es jetzt nur noch 80 Mbit/s. Die TCP-Verbindung kann nur wenig davon profitieren, weshalb die gesamte Effizienz stark abgenommen hat.

Reno gegen CMT/RPv2



(a) Reno gegen CMT/RPv2 ohne zusätzliche Latenz (b) Reno gegen CMT/RPv2 mit 5ms Latenz

Abbildung 5.26.: Reno gegen CMT/RPv2 auf dem *Half-Bottleneck*

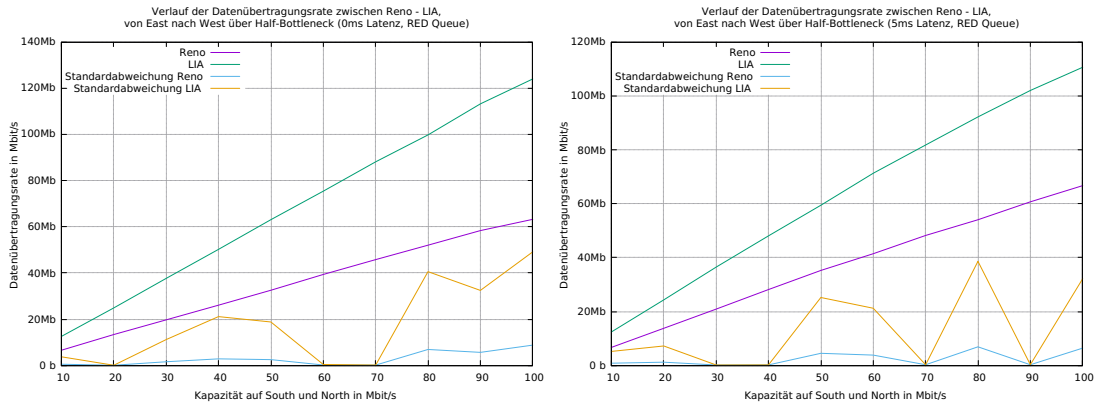
Das CMT/RPv2 zeigt ein ähnliches Bild wie OLIA zuvor (5.24). Bei näherer Betrachtung ist zu sehen, dass das CMT/RPv2 die TCP-Verbindung deutlich mehr beeinträchtigt und diese dadurch etwas weniger Kapazität in Anspruch nehmen kann. Dafür profitiert das CMT/RPv2 von der verminderten Kapazitätsausnutzung der TCP-Verbindung. Das selbe Bild zeigt sich auch bei einer Latenz von 5ms. Die Effizienz ist fast die selbe wie ohne zusätzliche Latenz. Nur bei 100 Mbit/s ist aufseiten des CMT/RPv2 eine marginal schlechtere Leistung zu erkennen.

Die ersten Messungen auf dem Half-Bottleneck-Aufbau der Testumgebung unterscheiden sich deutlich von denen im *Bottleneck*. Das liegt neben den am Anfang angesprochenen Schwankungen zusätzlich daran, dass abseits von geteilten Datenpfaden die ersten beiden Prinzipien des *Resource Poolings* für MPTCP-Verbindungen keine Anwendung finden, da kein gemeinsamer *Bottleneck* vorliegt. Stattdessen tritt hier das dritte Prinzip des *Resource Poolings* in Kraft und bestimmt, wieviel Bandbreite auf weniger belastete Pfade umgeleitet wird. In allen Fällen ist zu sehen, dass das dritte Prinzip des *Resource Poolings* Anwendung findet. Die TCP-Verbindung nimmt auf dem südlichen Pfad oftmals deutlich mehr als 50% der zur Verfügung stehenden Kapazität in Anspruch. Dennoch sind die MPTCP-Verbindungen der TCP-Verbindung deutlich überlegen, was dafür spricht, dass diese den Großteil ihrer Datenübertragung auf dem freien Übertragungspfad umleiten, um so die Paketverlustrate auf dem gemeinsamen Pfad zu reduzieren.

Eine TCP-Verbindung gegen eine MPTCP-Verbindung mit der RED-Queue im Half-Bottleneck-Aufbau

Im Unterschied zu dem vorherigen Testszenario wurden die Messungen hier mit dem Algorithmus für das Queue-Management RED durchgeführt.

Reno gegen LIA

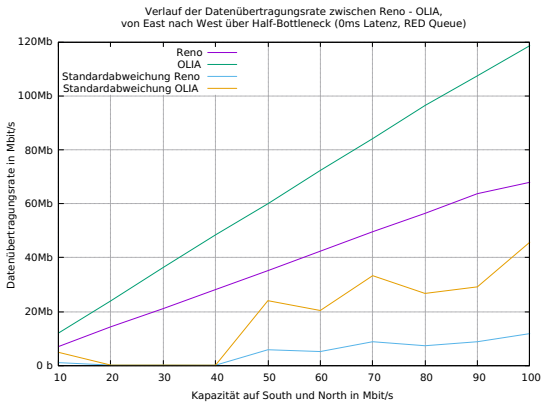


(a) Reno gegen LIA ohne zusätzliche Latenz (b) Reno gegen LIA mit 5ms Latenz

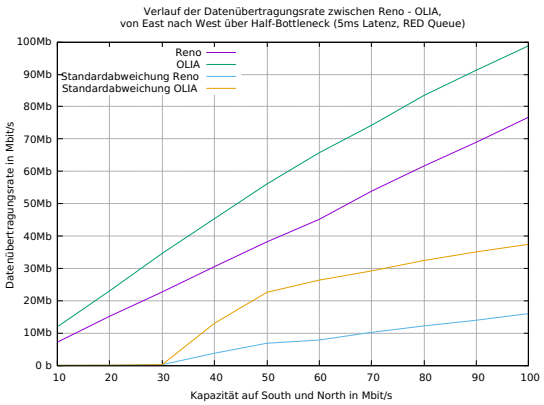
Abbildung 5.27.: Reno gegen LIA auf dem *Half-Bottleneck* mit der RED-Queue

Im Vergleich zur Messung ohne RED-Queue (5.22) fällt auf, dass die Bandbreitenverläufe mit der RED-Queue sehr linear sind und keine sprunghaften Steigungen aufweisen. Die Übertragungsrate bei LIA ist mit bis zu 11 Mbit/s bei einer Kapazität von 80 Mbit/s besser als bei der Tail-Drop-Queue. Dafür büßt die TCP-Verbindung auf der anderen Seite die bei LIA neu hinzugewonnene Durchsatzrate ein, was am Ende auf fast die selbe Effizienz der Kapazität für beide Szenarien hinausläuft. Im Fall mit der Latenz von 5ms ist eine etwas geringere Erhöhung aufseiten von LIA, verglichen mit dem Testszenario ohne RED-Queue (5.22), zu beobachten. Hier beträgt die Differenz bei 100 Mbit/s nur noch 4 Mbit/s. Die TCP-Verbindung nimmt dagegen bei einer Kapazität von 100 Mbit/s um 17 Mbit/s ab, was letztendlich in einer schlechteren Effizienz für die RED-Queue resultiert.

Reno gegen OLIA



(a) Reno gegen OLIA ohne zusätzliche Latenz

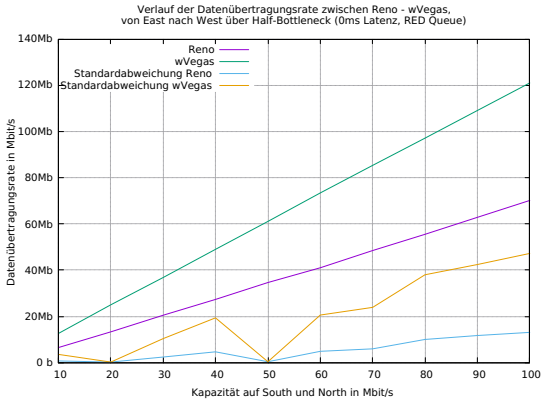


(b) Reno gegen OLIA mit 5ms Latenz

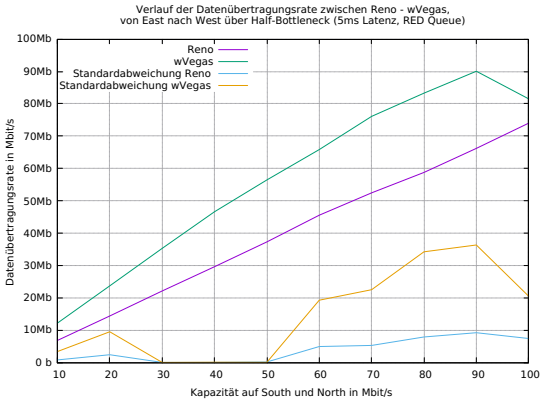
Abbildung 5.28.: Reno gegen OLIA auf dem *Half-Bottleneck* mit der RED-Queue

Gegenüber der Messung mit der Tail-Drop-Queue verzeichnet OLIA eine höhere Bandbreite, die jedoch, wie bei LIA zuvor auch, auf Kosten der Bandbreite von Reno geht. In Abbildung 5.28b verliert OLIA im Vergleich zur Abbildung 5.28a bis zu 20 Mbit/s bei einer Kapazität von 100 Mbit/s pro Pfad. Von diesen 20 Mbit/s gehen jedoch nur 9 Mbit/s an die TCP-Verbindung über, was in einer entsprechend geringen Effizienz resultiert.

Reno gegen wVegas



(a) Reno gegen wVegas ohne zusätzliche Latenz

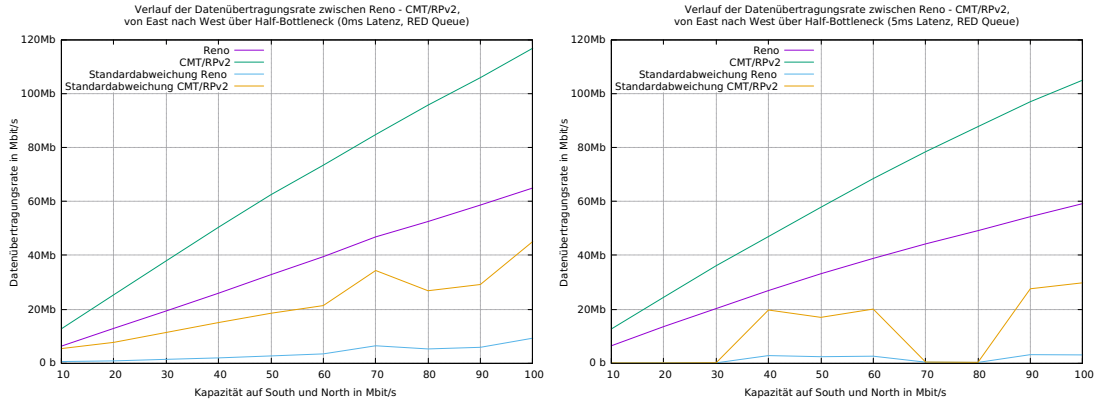


(b) Reno gegen wVegas mit 5ms Latenz

Abbildung 5.29.: Reno gegen wVegas auf dem *Half-Bottleneck* mit der RED-Queue

Abbildung 5.29a zeigt einen linearen Verlauf beider Verbindungen mit einer durchweg gleichbleibenden Steigung. Die Ausnutzung der Kapazität konnte verglichen mit der Messung aus der Tail-Drop-Queue bei wVegas um bis zu 10 Mbit/s gesteigert werden, die allerdings nun auf der anderen Seite der TCP-Verbindung fehlen. Wie auch in 5.25b mit der Tail-Drop-Queue, ist in 5.29b ab 90 Mbit/s ein Fallen der Durchsatzrate zu beobachten. Es scheint an dieser Stelle ein generelles Problem mit wVegas im Half-Bottleneck-Aufbau zu sein.

Reno gegen CMT/RPv2



(a) Reno gegen CMT/RPv2 ohne zusätzliche Latenz (b) Reno gegen CMT/RPv2 mit 5ms Latenz

Abbildung 5.30.: Reno gegen CMT/RPv2 auf dem *Half-Bottleneck* mit der RED-Queue

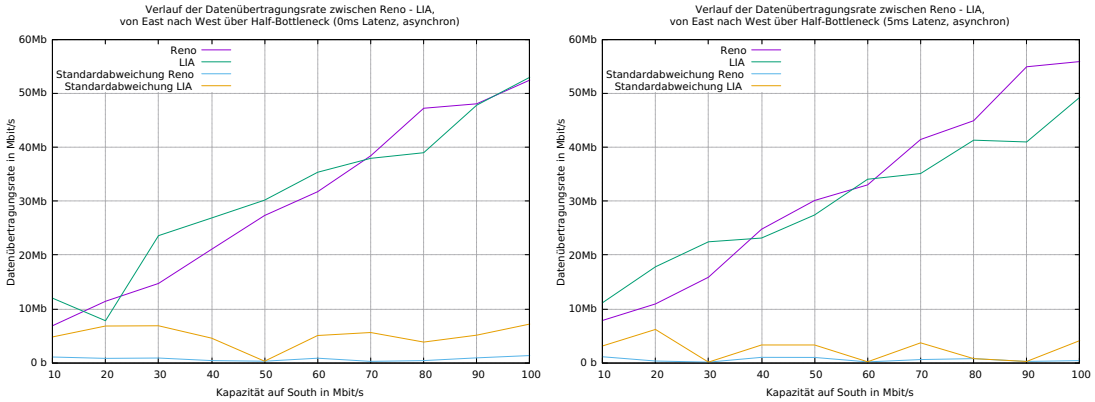
Das CMT/RPv2 kann in Abbildung 5.30a verglichen zur Messung mit der Tail-Drop-Queue nicht an Durchsatz hinzugewinnen. Nur die TCP-Verbindung verliert etwas an Bandbreite, weshalb die Effizienz etwas geringer ausfällt. Sobald eine Latenz von 5ms anliegt, beginnt auch das CMT/RPv2, an Durchsatz zu verlieren, den die TCP-Verbindung jedoch nicht auffangen kann.

Abgesehen von den Bandbreiten fällt bei der RED-Queue das jegliche Fehlen von Schwankungen in den Messungen auf. Die Bandbreitenverläufe sind bei allen Algorithmen sehr linear und zeigen, abgesehen von wVegas in 5.18b, kaum Änderungen der Steigung auf. Durch die Eigenschaft der RED-Queue, Pakete bereits vereinzelt vor einem drohendem Überlauf des Buffers zu verwerfen, scheinen die Messungen eine Stabilisierung zu erfahren. Es kommt somit keine abrupte Synchronisation aller Verbindungen über gemeinsame Paketverluste zustande, wie es sonst bei der Tail-Drop-Queue der Fall wäre.

Eine TCP-Verbindung gegen eine MPTCP-Verbindung mit asynchronen Kapazitäten im Half-Bottleneck-Aufbau

Nachdem auf dem *Half-Bottleneck* zuvor die Kapazitäten sowohl auf North als auch auf South synchron erhöht wurden, erfolgt hier das Szenario, auf dem dies asynchron geschieht. Dafür wird auf dem Rechner North eine feste Kapazitätsgrenze von 10 Mbit/s eingestellt. Der Rechner South hingegen durchläuft für die Dauer der Messung alle Kapazitäten von 10 Mbit/s bis 100 Mbit/s in 10-Mbit-Schritten. Je weiter eine Messung vorangeschritten ist, desto größer ist der Unterschied der Kapazitäten zwischen den beiden Pfaden.

Reno gegen LIA



(a) Reno gegen LIA ohne zusätzliche Latenz

(b) Reno gegen LIA mit 5ms Latenz

Abbildung 5.31.: Reno gegen LIA auf dem *Half-Bottleneck*

Die erste Messung mit den asynchronen Kapazitäten zeigt die bisher stärksten Schwankungen hinsichtlich des Bandbreitenverlaufs auf dem *Half-Bottleneck*. Im ersten Diagramm fällt zuerst bei 20 Mbit/s ein Sturz der Bandbreite bei LIA auf. Dieser ist den Schwankungen der Messung mit der Auswahl des Medians an dieser Stelle geschuldet. Der weitere Verlauf verbleibt relativ stabil. Im Vergleich der Messungen in 5.31a und 5.31b zueinander ist nur aufseiten von LIA ein leichter Rückgang der Bandbreite zu beobachten. Die Effizienz bleibt bei beiden Messungen nahezu identisch.

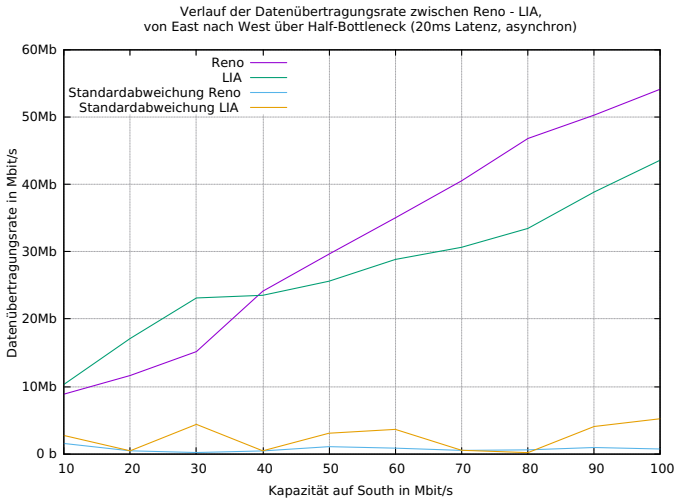
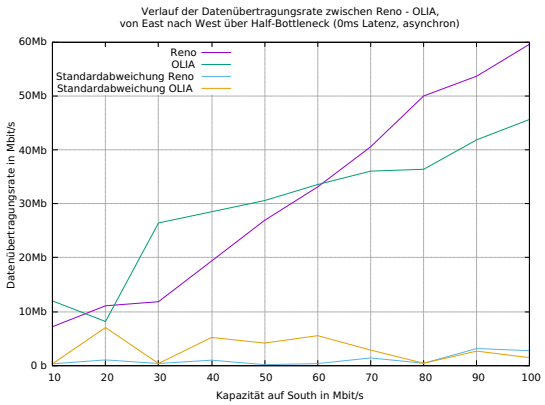


Abbildung 5.32.: Reno gegen LIA mit 20ms Latenz

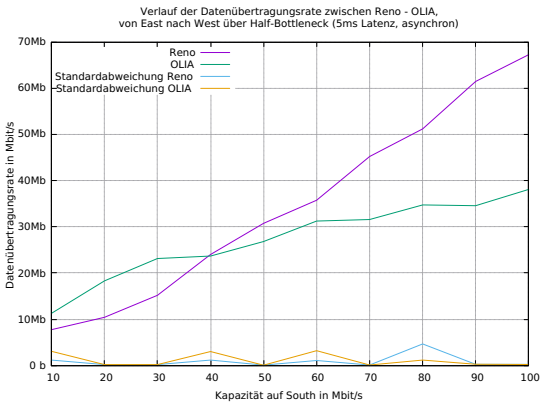
Da in diesem Szenario nur auf dem südlichen Pfad der Testumgebung die Bandbreite schrittweise erhöht wird und die Bandbreite auf dem nördlichen Pfad auf 10 Mbit/s verbleibt, resultiert verglichen zum vorherigen Szenario ein deutlich geringeres Bandbreiten-Delay-Produkt. Dieses erlaubt die Messungen mit 20ms weiter fortzuführen und zu beschreiben.

Bei einer Latenz von 20ms sind die Schwankungen nicht mehr so stark wie zuvor. Ab 30 Mbit/s nimmt die Steigung der Bandbreite von LIA stark ab, weshalb sich bei 40 Mbit/s ein Schnittpunkt zwischen den beiden Verbindungen ergibt. Danach kann sich die Steigung von LIA etwas erholen, unterliegt jedoch über die restliche Dauer der Messung der TCP-Verbindung. Gegenüber den geringeren Latenzen weist LIA eine verringerte Bandbreite auf. Reno kann dabei nicht wesentlich an Bandbreite hinzugewinnen, weshalb die Effizienz etwas zurückbleibt.

Reno gegen OLIA



(a) Reno gegen OLIA ohne zusätzliche Latenz



(b) Reno gegen OLIA mit 5ms Latenz

Abbildung 5.33.: Reno gegen OLIA auf dem *Half-Bottleneck*

Das selbe Szenario mit OLIA zeigt bei einer Latenz von <1ms bei 20 Mbit/s, wie LIA zuvor auch (5.31a), einen Rückgang der Bandbreite, welche danach sprunghaft nach oben steigt. Danach steigen beide Verbindungen konstant an, wobei OLIA bei 60 Mbit/s der TCP-Verbindung im weiteren Verlauf unterliegt. Ab 80 Mbit/s gleichen sich jedoch die Steigungsraten der Bandbreiten an, wodurch sich der Abstand voneinander nicht erhöht. In 5.33b unterliegt OLIA kurz vor 40 Mbit/s Reno und steigt wesentlich langsamer als Reno an. Davon unberührt bleibt die Effizienz, welche verglichen mit der in 5.31a gleich bleibt.

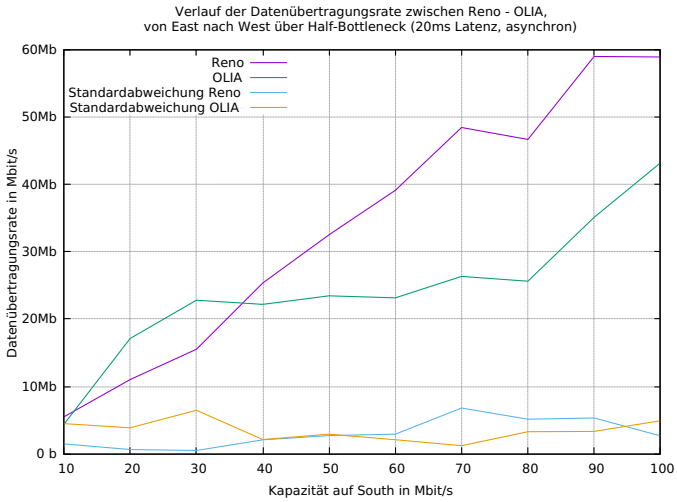
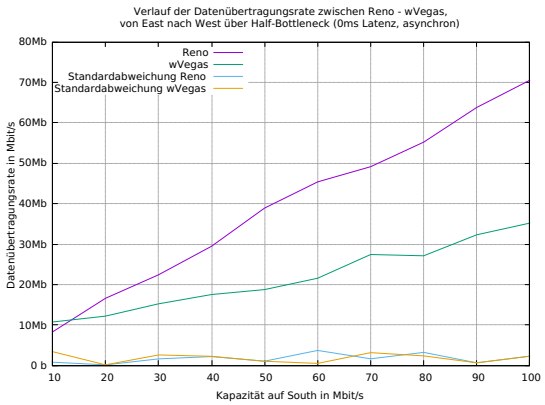


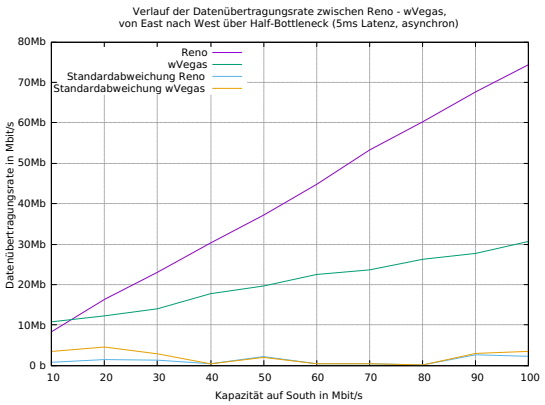
Abbildung 5.34.: Reno gegen OLIA mit 20ms Latenz

Aus der Abbildung 5.34 geht hervor, dass OLIA ab 30 Mbit/s bis ca. 80 Mbit/s fast eine Stagnation der Bandbreite ausweist. Die TCP-Verbindung kann währenddessen annähernd linear die eigene Bandbreite ausbauen. Erst danach bessert sich OLIA und legt zwischen den Kapazitäten 80 und 100 Mbit/s ca. 20 Mbit/s hinzu. Die Effizienz ist während der Phase der Beinahe-Stagnation im Unterschied zur LIA-Messung zuvor leicht vermindert. Danach kann sich diese jedoch durch eine Steigerung von OLIA wieder erholen und sogar einen leichten Vorteil zur LIA-Messung bei einer Kapazität von 100 Mbit/s erzielen. Auffällig ist besonders die Stagnation der TCP-Verbindung zwischen 90 und 100 Mbit/s. Es ist naheliegend, dass hier das selbe Problem in Erscheinung tritt wie in 5.23 diskutiert.

Reno gegen wVegas



(a) Reno gegen wVegas ohne zusätzliche Latenz



(b) Reno gegen wVegas mit 5ms Latenz

Abbildung 5.35.: Reno gegen wVegas auf dem *Half-Bottleneck*

Das wVegas zeigt im asynchronen Szenario das typische Verhalten einer delay-based *Congestion Control*. Hierbei unterliegt das wVegas konsequent Reno. Die Steigungen beider Verbindungen sind dafür konstant und zeigen keine nennenswerten Einbrüche auf. Daraus resultiert bei beiden Latenzen eine ähnliche Effizienz wie bei den vorherigen Messungen.

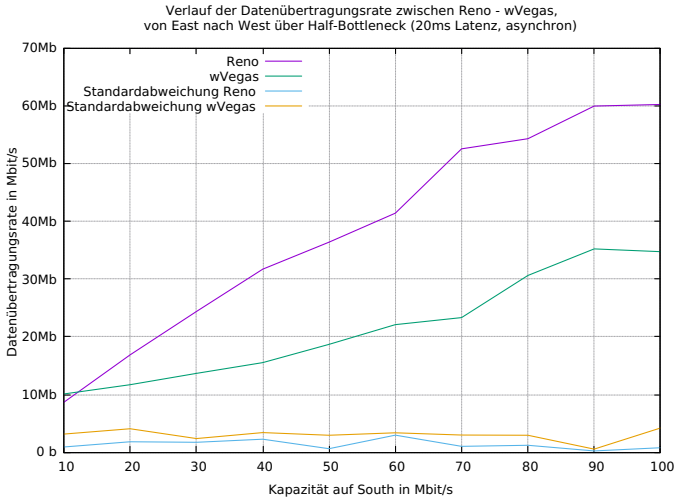
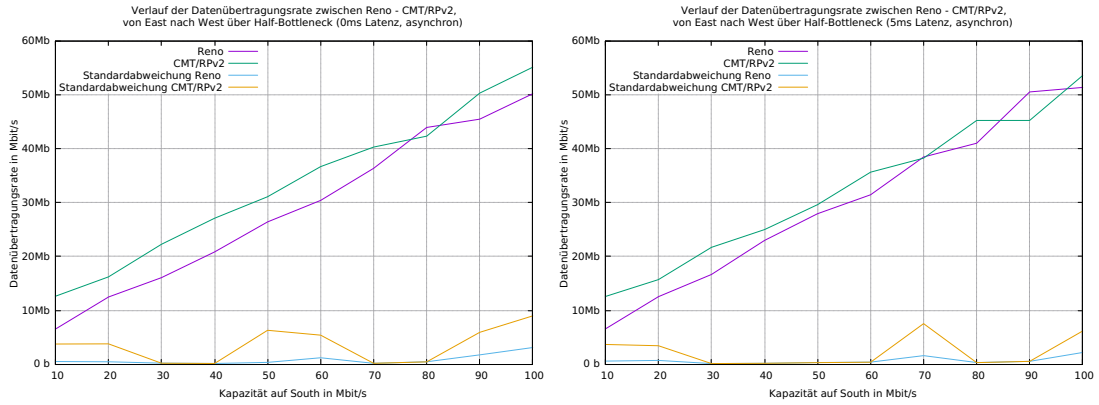


Abbildung 5.36.: Reno gegen wVegas mit 20ms Latenz

Bis 70 Mbit/s zeigen sowohl das wVegas als auch Reno ein ähnliches Bild wie im Fall mit der 5ms Latenz. Danach ist deutlich eine Reduktion der Übertragungsrate auszumachen, welches zuletzt in eine Stagnation mündet. Genauso wie in Abbildung 5.23 stagniert die Bandbreite von Reno bei 60 Mbit/s und kann nicht darüber hinaus wachsen. Aufgrund der daraus resultierenden verfälschten Werte wird das Ende des Messergebnisses nicht in die Bewertung einfließen.

Reno gegen CMT/RPv2



(a) Reno gegen CMT/RPv2 ohne zusätzliche Latenz (b) Reno gegen CMT/RPv2 mit 5ms Latenz

Abbildung 5.37.: Reno gegen CMT/RPv2 auf dem *Half-Bottleneck*

Das CMT/RPv2 kann verglichen mit den vorherigen Messungen des Szenarios einen deutlich ruhigeren Bandbreitenverlauf vorweisen. Ein Absturz der Bandbreite bei 20 Mbit/s ist nicht zu sehen. Bis auf den Messpunkt bei 80 Mbit/s in 5.37a steigen beide Verbindungen nahezu linear, wobei das Bandbreitenverhältnis zwischen dem CMT/RPv2 und Reno immer gleich bleibt. Über fast die gesamte Messung hinweg besitzt das CMT/RPv2 die größte Bandbreite der MPTCP-Verbindungen für eine Latenz <1ms in diesem Szenario. Abbildung 5.37b zeigt ein ähnliches Bild wie das zuvor. Gegen Ende der Messung fällt auf, dass die Verbindungen öfter schwingen, weshalb ein Flecht-Muster zustande kommt. Trotz der Schwingungen und des höheren Bandbreiten-Delay-Produkts mit 5ms Latenz ist die Effizienz immer noch die selbe wie ohne zusätzliche Latenz.

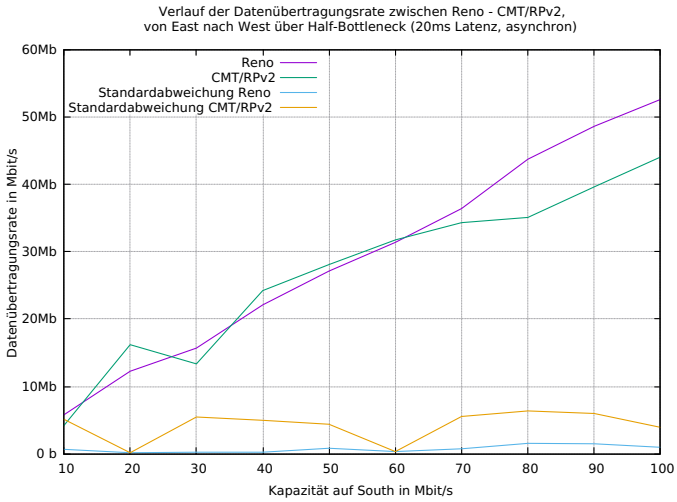


Abbildung 5.38.: Reno gegen CMT/RPv2 mit 20ms Latenz

Bis 60 Mbit/s sind das CMT/RPv2 und Reno sehr nahe beieinander. Danach ist eine Trennung zwischen den Verbindungen zu beobachten, in der Reno konsequent über dem CMT/RPv2 liegt. Die Effizienz hat sich zum 5ms-Latenz-Fall leicht verringert. Dabei velor das CMT/RPv2 bis zu 10 Mbit/s, während die TCP-Verbindung die Bandbreite beibehalten und an einigen Stellen erhöhen konnte.

Das asynchrone Szenario erlaubte einen Blick darauf zu werfen, wie sich die Bandbreitenverteilung bei einer Erhöhung des gemeinsamen Pfades verändert. Dabei ähnelt dieses Szenario einem *Bottleneck*, da mit zunehmender Kapazität des gemeinsamen Pfades der alternative Pfad immer weniger Einfluss auf das Geschehen hat. Dennoch haben aufgrund des alternativen Pfades die ersten beiden Regeln des *Resource Pooling Principles* zur Beseitigung des Bottleneck-Problems keine Wirkung, weshalb keine Fairness auf dem gemeinsamen Pfad gewährleistet wird.

5.2. Bewertung

Jedes einzelne der sieben Testszenarien stellte die Congestion-Control-Algorithmen vor eine neue Herausforderung. Anhand der im Kapitel 4.4 beschriebenen Kriterien der Fairness und der Effizienz wird versucht, die Leistungen der Algorithmen einzuordnen und gegeneinander abzuwägen. Am Ende einer Bewertung eines Testszenarios erfolgt die Hervorhebung von mindestens einem Algorithmus oder mehreren Algorithmen, die ein Szenario besser absolvieren konnten, als andere.

Bottleneck

Eine TCP-Verbindung gegen eine MPTCP-Verbindung

In Konkurrenz mit einer TCP-Verbindung um die selben Ressourcen zeigten alle loss-based Congestion-Control-Algorithmen die deutliche Tendenz zu einer fairen Bandbreitenverteilung von 50:50. Aufgrund des im ersten Szenario beschriebenen niedrigen statistischen Multiplexing lag die LIA-Congestion-Control über dem Anteil von 50%. OLIA als auch das CMT/RPv2 wiesen das selbe Verhalten auf, obwohl solch ein Mechanismus bei einem niedrigen statistischen Multiplexing bei diesen Algorithmen nicht vorgesehen war. Deshalb bewegten sich die Bandbreiten aller loss-based Algorithmen auf dem selben Niveau. Das wVegas sprach durch seine delay-based Eigenschaft Reno am meisten Kapazität zu, die jedoch mit einer eigenen Durchsatzschwäche einherging. Besonders sticht bei näherer Betrachtung der einzelnen Bandbreiten hervor, dass sich die loss-based Algorithmen schwer tun, bei einem hohem Bandbreiten-Delay-Produkt eine gute Effizienz zu gewährleisten. Hierbei erweist sich die Messung mit dem CMT/RPv2 mit einer Gesamtkapazitätsausnutzung von 76 Mbit/s bei einer eingestellten Kapazität von 100 Mbit/s und einer Latenz von 20ms am schwächsten und nutzt den Link nur zu 75% aus. Trotz dieser Durchssatzschwäche handelt das CMT/RPv2 noch im Rahmen des *Resource Pooling Principles*, was für eine korrekte Implementierung im Linux Kernel spricht. Danach folgt die LIA-Congestion-Control mit einer geringfügig besseren Bandbreite der TCP-Verbindung. Im völligen Gegensatz dazu kann die wVegas-Messung bei einer eingestellten 20ms-Latenz und 100 Mbit/s Kapazität noch einen hervorragenden Wert von 92 Mbit/s ausnutzen. Jedoch ist diese Effizienz auf die delay-based Eigenschaft des wVegas in Verbindung mit der aggressiven Bandbreitenausnutzung von Reno zurückzuführen, was letztendlich in einem einseitigen Verhältnis für Reno mündet.

Es bleibt hier festzuhalten, dass keiner der Algorithmen eine optimale Fairness erzielen konnte. Mit zunehmendem Bandbreiten-Delay-Produkt haben vor allem die loss-based Algorithmen

Schwierigkeiten. Mit einem kleinen Vorsprung vor LIA hinsichtlich der Effizienz und der besseren Bandbreitenverteilung kann das OLIA hier am meisten überzeugen.

Eine MPTCP-Verbindung gegen eine andere MPTCP-Verbindung

Im Szenario der MPTCP-Algorithmen gegen sich selbst hat sich bei allen Messungen ein verbessertes Bandbreitenverhältnis herausgestellt, als es gegen die TCP-Verbindung der Fall war. In fast jeder Messung konnte ein Verhältnis von 50:50 erzielt werden. Nur unterschieden sich die Messungen im Hinblick ihrer Effizienz. Das wVegas konnte gegen sich selbst mit maximal 94 Mbit/s über alle Latenzen hinweg das beste Ergebnis erreichen. Danach folgen LIA und OLIA, welche bei einer Latenz von 20ms einen maximalen Durchsatz von 80 und 82 Mbit/s vorweisen konnten. Das CMT/RPv2 kann hingegen, ähnlich wie in der Messung mit Reno, wieder nur knapp 75% der Kapazität für sich beanspruchen.

Zwei MPTCP-Verbindungen gegen eine TCP-Verbindung

Im Szenario mit dem höheren statistischen Multiplexing unterlagen die MPTCP-Verbindungen OLIA und CMT/RPv2 über alle Latenzen hinweg um ungefähr die Hälfte der TCP-Verbindung. Mit steigendem Bandbreiten-Delay-Produkt verringerte sich die Effizienz leicht. Das CMT/RPv2 hingegen steigerte konstant seine Bandbreite. Es ist in diesem Fall verwunderlich, weshalb die vier *MPTCP-Subflows* nur die Hälfte der zur Verfügung stehenden Bandbreite untereinander teilen, während eine einzelne TCP-Verbindung die andere Hälfte für sich alleine beanspruchen kann. Insgesamt bleibt nur das CMT/RPv2 hervorzuheben, welches trotz hohem Bandbreiten-Delay-Produkt immer die selbe Leistung erbringen konnte.

Eine TCP-Verbindung gegen eine MPTCP-Verbindung mit der RED-Queue

Die Umstellung des Queue-Managements auf RED hat in allen Testszenarien große Auswirkungen gezeigt. Bei keiner zusätzlichen Latenz wurde bei LIA, OLIA und dem CMT/RPv2 sichtbar, dass die Fairness auf dem *Bottleneck* deutlich besser befolgt wurde als im Tail-Drop-Szenario. Im 5ms-Latenz-Fall verbesserte sich bei LIA und OLIA das Bandbreitenverhältnis zur TCP-Verbindung fast zum Ideal. Nur in der Messung mit dem CMT/RPv2 wurde die TCP-Verbindung stärker verdrängt, weshalb sich das Verhältnis verschlechterte. Das spiegelt sich auch in der Effizienz wieder, die mit dem größten Abfall von 7 Mbit/s bei einer Kapazität von 100 Mbit/s zur Messung ohne zusätzliche Latenz zurückliegt.

Zuletzt zeigte die Messung im 20ms-Latenz-Fall bei allen drei loss-based Algorithmen zu-

sammen eine konsequent niedrigere Ausnutzung der Bandbreite als Reno. Bei der Effizienz haben alle Algorithmen einen deutlichen Rückgang zu verzeichnen. Dieser Rückgang beträgt 16 und 17 Mbit/s bei den loss-based Algorithmen, während es beim wVegas nur 8 Mbit/s sind. Insgesamt zeigte das wVegas im Gegensatz zu den anderen Algorithmen ein anderes Bild. Die dem loss-based Reno vorher stark unterlegene *Congestion Control* (5.4) nimmt ohne zusätzliche Latenz mehr als 50% der Bandbreite in Anspruch und besitzt durchweg eine höhere Bandbreite als Reno. Mit höherem Bandbreiten-Delay-Produkt schwindet dieser Vorteil, jedoch zeigt die Messung unter fast allen Latenzen in diesem Szenario das beste Ergebnis.

Das wVegas mag am Ende die effizienteste und durchsatzstärkste Messung vorweisen, jedoch ging dieser Vorteil auf Kosten der Fairness zwischen den beiden Verbindungen. Deutlich fairer, aber dafür uneffizienter erwiesen sich LIA und OLIA. Stellenweise konnte das CMT/RPv2 die beste Fairness aufzeigen, allerdings liegt die Kapazitätsausnutzung deutlich hinter den anderen Algorithmen zurück.

Half-Bottleneck

Aufgrund des zusätzlichen Pfades können für die Messungen auf dem Half-Bottleneck-Aufbau nur das Verhältnis und die Effizienz bewertet werden.

Eine TCP-Verbindung gegen eine MPTCP-Verbindung im Half-Bottleneck-Aufbau

Unter den Congestion-Control-Algorithmen für MPTCP lagen die Bandbreiten ohne Latenz recht nahe beieinander und wiesen keine stärkeren Einbrüche auf, obwohl die Messungen starken Schwankungen unterworfen waren. Auch die Effizienz war mit 189 bis 191 Mbit/s bei einer Gesamtkapazität von 200 Mbit/s sehr gut. Mit einer Latenz von 5ms haben sich stärkere Unterschiede bemerkbar gemacht. Während die loss-based MPTCP-Algorithmen leicht an Bandbreite verloren haben, die jedoch mit einem leichten Gewinn der TCP-Verbindung einherging, stürzte das wVegas regelrecht ab und verlor 31 Mbit/s, die Reno nicht an sich nehmen konnte, um die vorher starke Effizienz beizubehalten. Das CMT/RPv2 wies in beiden Messungen die größte Übertragungsrate auf, welche jedoch wieder auf Kosten der Bandbreite der TCP-Verbindung ging. Unter Berücksichtigung aller Pfadkapazitäten wäre es wünschenswert und fair, wenn die TCP-Verbindung eine Bandbreite in der Nähe von 100Mbit/s erhalten würde und die MPTCP-Verbindungen dafür den alternativen Pfad voll in Anspruch nehmen würden. Mit einer Bandbreite von 90Mbit/s bei einer Kapazität von 100Mbit/s entspricht die TCP-Verbindung bei der OLIA-Messung am ehesten dem Ideal.

Insgesamt bleibt hier festzuhalten, dass LIA und OLIA die beste Effizienz vorzeigen können. Während LIA mehr den eigenen Durchsatz hoch hält, überlässt OLIA der TCP-Verbindung mehr Bandbreite.

Eine TCP-Verbindung gegen eine MPTCP-Verbindung mit der RED-Queue im Half-Bottleneck-Aufbau

Die Messungen mit der RED-Queue sollten darlegen, inwiefern sich das Queue-Management auf die Schwankungen und die Effizienz der Algorithmen auswirken kann. Ohne zusätzliche Latenz erfuhren die MPTCP-Algorithmen, mit Ausnahme des CMT/RPv2, eine Steigerung der eigenen Bandbreite von mindestens 10 Mbit/s bei einer Kapazität von 100 Mbit/s verglichen zur Tail-Drop-Messung. Im Gegenzug dafür büßten die TCP-Verbindungen der vorher angesprochenen Messungen mindestens die selbe Bandbreitengröße ein. Mit einem höheren Bandbreiten-Delay-Produkt nahm bei einer Latenz von 5ms in allen Messungen die Sendeleistung ab. Jedoch waren die Übertragungsraten der MPTCP-Algorithmen weiterhin auf einem ähnlichem Level wie in der Tail-Drop-Messung. Das zuvor im Bottleneck-Szenario mit der RED-Queue erstarkte wVegas kann keine Vorteile aus dem Queue-Management ziehen und besitzt mit 31 Mbit/s den größten Einbruch der Bandbreite durch die Erhöhung der Latenz. Bis auf die Messung mit dem CMT/RPv2 konnten die TCP-Verbindungen der anderen Messungen sogar etwas an Bandbreite hinzugewinnen. Jedoch fällt dieser Gewinn zu klein aus, um die Effizienz auf das Niveau der Tail-Drop-Messung zu bringen.

Zusammenfassend lässt sich für die RED-Messung auf dem *Half-Bottleneck* sagen, dass die Datenübertragungen eine gesamte Stabilisierung der Verbindungen erfahren haben, obgleich die Kapazitätsausnutzung dafür abgenommen hat. Dafür ist insbesondere die Abnahme der Bandbreite der TCP-Verbindung die Ursache. Das CMT/RPv2 konnte mit dem wVegas in diesem Szenario am wenigsten überzeugen.

Eine TCP-Verbindung gegen eine MPTCP-Verbindung mit asynchronen Kapazitäten im Half-Bottleneck-Aufbau

Hervorzuheben ist im asynchronem Szenario, dass alle Messungen bei einer Latenz von <1ms und 5ms mit einer Bandbreite von insgesamt 104 und 105 Mbit/s nahe am absoluten Optimum bei einer Gesamtkapazität von 110 Mbit/s auf dem südlichen Pfad agierten. Erst bei 20ms stellten sich aufgrund der Bandbreitenbeschränkung größere Einbrüche in der Effizienz ein. Den letzten Fall ausgenommen, sind alle Messungen dieses Szenarios als einzige im Half-Bottleneck-Aufbau sehr nahe beieinander. Allerdings besteht in diesem Szenario durch die fehlende Dynamik

der Bandbreiten eine zu starke Ähnlichkeit mit dem einfachen Bottleneck-Szenario, weshalb große Effekte nicht zu beobachten waren. Nur im Verhältnis der Bandbreiten der Verbindungen sind Unterschiede auszumachen. Das CMT/RPv2 agiert hier am stärksten, wenn auch die TCP-Verbindung dafür die geringste Bandbreite in Anspruch nimmt. Das wVegas spricht wie üblich der TCP-Verbindung am meisten Bandbreite zu. Den Mittelweg zeigten LIA und OLIA auf, welche nicht ganz so viel Bandbreite wie das CMT/RPv2 in Anspruch nahmen, aber dafür der TCP-Verbindung etwas mehr zusprachen.

Insgesamt kann nach den Messungen und der anschließenden Bewertung festgehalten werden, dass LIA und OLIA über die meisten Messungen hinweg die beste Effizienz oder Fairness aufweisen konnten. Dabei nahm die LIA-Congestion-Control immer etwas mehr oder im Szenario gegen sich selbst genauso viel Bandbreite ein wie OLIA. Letztere *Congestion Control* war der Fairness mehr zugeneigt und ermöglichte der TCP-Verbindung mehr Bandbreite. Das im Rahmen dieser Arbeit implementierte CMT/RPv2 hatte besonders im Bottleneck-Aufbau der Testumgebung Schwierigkeiten bei einem hohen Bandbreiten-Delay-Produkt anderen Verbindungen als auch sich selbst eine ausreichend große Bandbreite zu gewähren. Trotz der frei verfügbaren Kapazität wurde diese nicht voll ausgenutzt, was eine schwache Bandbreitenausnutzung nach sich zog. Auf dem Half-Bottleneck-Aufbau fiel das CMT/RPv2 dagegen besonders im asynchronen und synchronen Szenario mit der Tail-Drop-Queue durch die größte Durchsatzrate aller MPTCP-Verbindungen auf. Hier kann der loss-based Algorithmus seine Stärken ausspielen, auch wenn diese leicht auf Kosten anderer Verbindungen auf dem selben Link gehen. Die einzige delay-based *Congestion-Control* wVegas konnte in Szenarien in Konkurrenz mit dem loss-based Reno und der Tail-Drop-Queue wenig überzeugen. Nur auf dem *Bottleneck* gegen sich selbst und im ersten Szenario mit der RED-Queue zeigte sich das Potential dieses Algorithmus. Es bestätigt sich letztendlich hier das in 3.2.2 beschriebene Einsatzgebiet für die Übertragung von Hintergrunddaten, ohne die nebenher laufenden Verbindungen zu beeinträchtigen.

6. Fazit

Das Ziel dieser Arbeit bestand darin, die bisher nur für FreeBSD verfügbare *Congestion-Control* für MPTCP CMT/RPv2 zu implementieren und anhand von Vergleichen mit anderen Congestion-Control-Algorithmen zu evaluieren. Die Messungen auf der *multihomed* Testumgebung legten offen, wie das CMT/RPv2 verglichen zu den etablierten Congestion-Control-Algorithmen abgeschnitten hat. Der Vergleich mit dem wVegas konnte hierbei zur Evaluierung wenig beitragen, da es aufgrund seiner delay-based Eigenschaft ganz andere Anforderungen erfüllt als die loss-based Algorithmen. Trotz des Erkenntnisgewinns durch die Messungen über die Einsatzmöglichkeiten des wVegas hätte sich anstelle dessen der Vergleich mit einer anderen loss-based *Congestion Control* mehr angeboten.

Aufgrund der Bandbreitenbeschränkung mit höheren Bandbreiten-Delay-Produkten war der Raum für die Messung mit selbst definierten Latenzen auf maximal 20ms beschränkt. Diese Beschränkung verhinderte, Aussagen über die Leistung der Algorithmen bei deutlich höheren Latenzen zu treffen und somit die Möglichkeit, eine umfangreichere Evaluation durchzuführen. In vielen Szenarien wären ohne das Auftreten des Problems größere Unterschiede zwischen den Algorithmen mit einem großen Bandbreiten-Delay-Produkt festgestellt worden. Für zukünftige Messungen in diesem Bereich sollte daher die Ursache des Problems gefunden und beseitigt werden.

Aus den durchgeführten Messungen geht hervor, dass das CMT/RPv2 zwar hinsichtlich der Fairness auf dem *Bottleneck* auf Augenhöhe mit den bereits etablierten loss-based Algorithmen LIA und OLIA ist, jedoch beim Datendurchsatz und der Effizienz diesen wenig entgegenzusetzen hat. Erst auf dem Half-Bottleneck-Aufbau bietet es durch seine Durchsatzstärke einen deutlichen Mehrwert zu LIA und OLIA. Ob dieser Vorteil einen möglichen Einsatz des Algorithmus in der Praxis rechtfertigt, ist am Ende von der gegebenen Infrastruktur und den eigenen Anforderungen abhängig.

A. Messergebnisse

A.1. Bottleneck

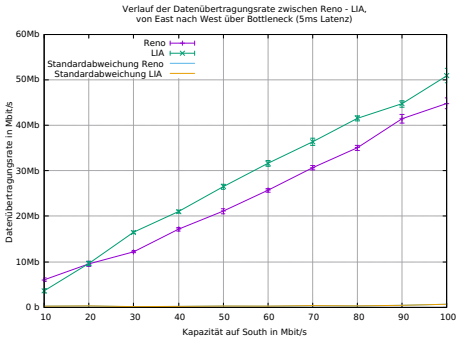


Abbildung A.1.: Reno gegen LIA mit 5ms Latenz

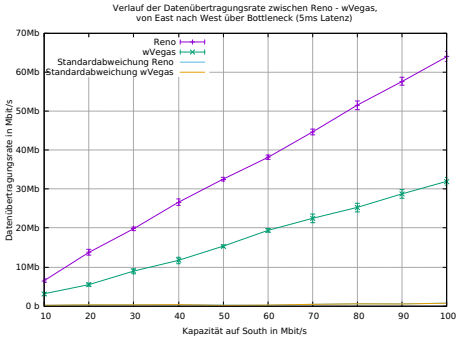


Abbildung A.2.: Reno gegen wVegas mit 5ms Latenz

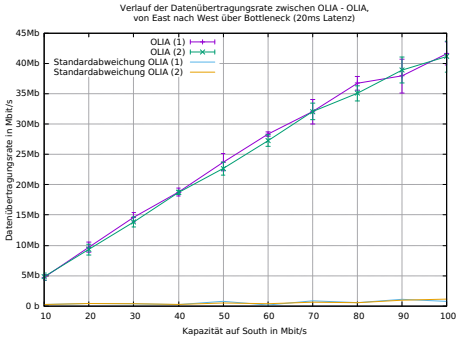


Abbildung A.3.: OLIA gegen OLIA mit 20ms Latenz

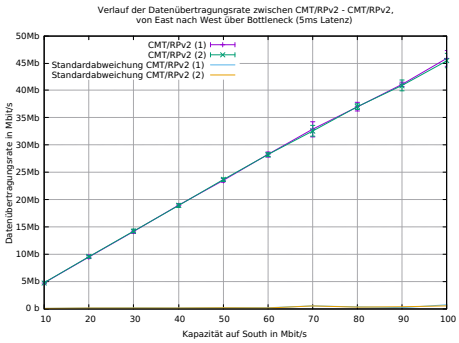


Abbildung A.4.: CMT/RPv2 gegen CMT/RPv2 mit 20ms Latenz

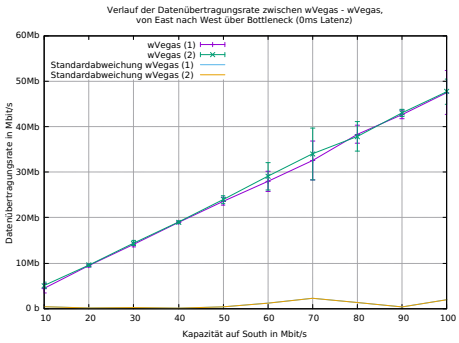


Abbildung A.5.: wVegas gegen wVegas ohne zusätzliche Latenz

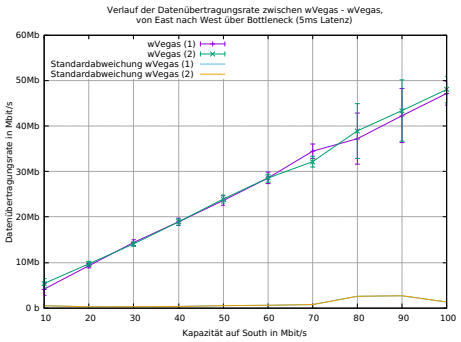


Abbildung A.6.: wVegas gegen wVegas mit 5ms Latenz

A.2. Half-Bottleneck

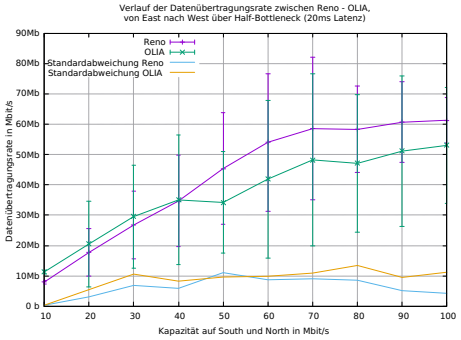


Abbildung A.7.: Reno gegen OLIA mit 20ms Latenz

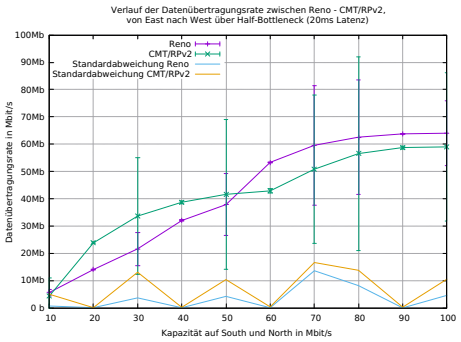


Abbildung A.8.: Reno gegen CMT/RPv2 mit 20ms Latenz

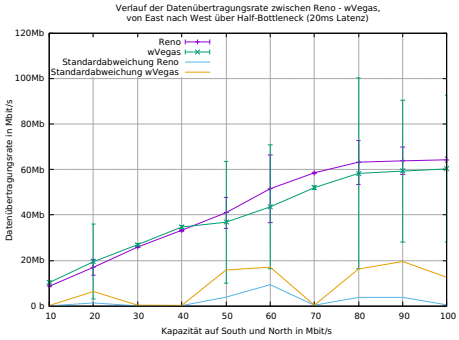


Abbildung A.9.: Reno gegen wVegas mit 20ms Latenz

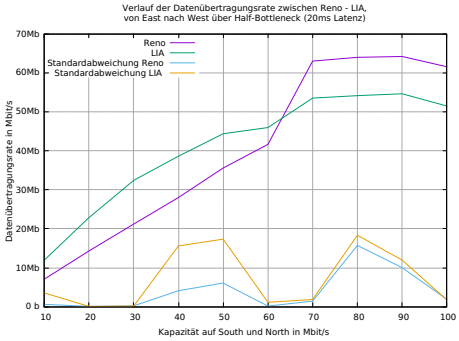


Abbildung A.10.: Reno gegen LIA mit 20ms Latenz

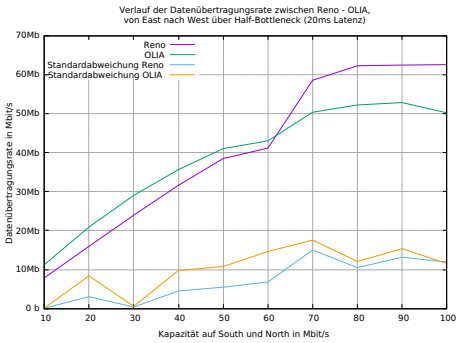


Abbildung A.11.: Reno gegen OLIA mit 20ms Latenz

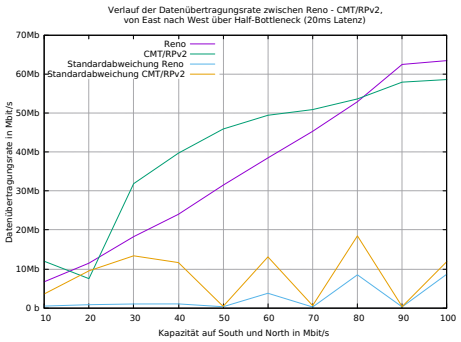


Abbildung A.12.: Reno gegen CMT/RPv2 mit 20ms Latenz

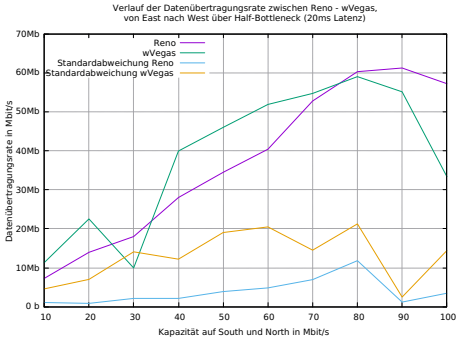


Abbildung A.13.: Reno gegen wVegas mit 20ms Latenz

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	9	10	19	9	9	18	9	9	18	6	13	19
40	21	16	37	21	16	37	20	17	37	12	26	38
60	32	25	57	31	26	57	30	26	56	19	37	56
80	41	35	76	40	36	76	38	37	75	26	50	76
100	50	45	95	47	48	95	47	47	94	36	59	95

Tabelle A.1.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Bottleneck* ohne zusätzliche Latenz

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	9	9	18	9	9	18	10	9	19	5	13	18
40	21	12	33	20	17	37	20	17	37	11	26	37
60	31	25	56	30	27	57	31	25	56	19	38	57
80	41	35	76	39	37	76	40	34	74	25	51	75
100	50	44	94	48	47	95	50	42	92	31	63	94

Tabelle A.2.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Bottleneck* für 5ms Latenz

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	9	9	18	9	9	18	9	9	18	6	13	19
40	19	17	36	19	18	37	18	17	35	12	25	37
60	27	26	53	26	28	54	25	25	50	20	34	54
80	32	35	67	29	37	66	32	32	64	25	54	79
100	36	44	80	36	47	83	37	39	76	36	56	92

Tabelle A.3.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Bottleneck* für 20ms Latenz

Kapazität	LIA	LIA	∑	OLIA	OLIA	∑	CMT/RPv2	CMT/RPv2	∑	wVegas	wVegas	∑
20	9	9	18	9	9	18	9	9	18	9	9	18
40	18	19	37	19	18	37	18	19	37	19	19	38
60	28	28	56	28	28	56	28	28	56	29	27	56
80	37	38	75	37	38	75	37	38	75	37	38	75
100	47	47	94	47	47	94	47	46	93	47	47	94

Tabelle A.4.: Übertragungsraten zwischen zwei MPTCP-Verbindungen in Mbit/s auf dem *Bottleneck* ohne zusätzliche Latenz

Kapazität	LIA	LIA	∑	OLIA	OLIA	∑	CMT/RPv2	CMT/RPv2	∑	wVegas	wVegas	∑
20	9	9	18	9	9	18	9	9	18	9	9	18
40	19	19	38	19	19	38	19	19	38	19	19	38
60	28	28	56	28	29	57	28	28	56	28	28	56
80	38	38	76	38	37	75	37	37	74	39	37	76
100	47	48	95	46	48	94	45	45	90	48	47	94

Tabelle A.5.: Übertragungsraten zwischen zwei MPTCP-Verbindungen in Mbit/s auf dem *Bottleneck* für 5ms Latenz

Kapazität	LIA	LIA	∑	OLIA	OLIA	∑	CMT/RPv2	CMT/RPv2	∑	wVegas	wVegas	∑
20	9	9	18	9	9	18	9	9	18	9	9	18
40	18	19	37	18	18	36	17	18	35	19	18	37
60	27	28	55	27	28	55	25	25	50	28	27	55
80	36	36	72	35	36	71	31	31	62	37	38	75
100	40	40	80	41	41	82	37	37	74	47	47	94

Tabelle A.6.: Übertragungsraten zwischen zwei MPTCP-Verbindungen in Mbit/s auf dem *Bottleneck* für 20ms Latenz

0ms Latenz					5ms latenz				20ms latenz			
Kapazität	OLIA	CMT/RPv2	Reno	∑	OLIA	CMT/RPv2	Reno	∑	OLIA	CMT/RPv2	Reno	∑
20	4	5	9	18	4	5	9	18	4	5	9	18
40	10	10	18	38	9	10	18	37	8	9	19	36
60	15	14	27	56	13	14	28	55	12	13	28	53
80	19	18	37	74	18	19	38	75	16	18	36	70
100	23	23	48	94	21	24	48	93	18	23	42	83

Tabelle A.7.: Übertragungsraten zwischen einer TCP-Verbindung und zwei MPTCP-Verbindungen in Mbit/s auf dem *Bottleneck*

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	9	9	18	9	9	18	10	8	18	11	7	18
40	20	18	38	20	18	38	20	18	38	21	16	37
60	30	26	56	30	27	57	29	28	57	32	25	57
80	39	36	75	40	36	76	37	36	73	42	33	75
100	48	45	93	48	46	94	44	47	91	49	45	94

Tabelle A.8.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Bottleneck* mit der RED-Queue ohne zusätzliche Latenz

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	9	9	18	9	9	18	10	8	18	10	8	18
40	19	18	37	19	19	38	20	17	37	20	18	38
60	28	27	55	28	28	56	29	25	54	27	29	56
80	37	36	73	36	37	73	37	32	69	35	40	75
100	46	45	91	43	47	90	44	40	84	43	50	93

Tabelle A.9.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Bottleneck* mit der RED-Queue für 5ms Latenz

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	8	10	18	8	10	18	9	9	18	9	9	18
40	17	18	35	16	18	34	16	16	32	18	18	36
60	23	26	49	21	26	47	22	22	44	27	26	53
80	29	33	62	27	35	62	27	29	56	32	36	68
100	34	41	75	32	42	74	31	36	67	40	45	85

Tabelle A.10.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Bottleneck* mit der RED-Queue für 20ms Latenz

Kapazität	LIA	Reno	$\sum$	OLIA	Reno	$\sum$	CMT/RPv2	Reno	$\sum$	wVegas	Reno	$\sum$
20	24	13	37	22	15	37	24	13	37	22	16	38
40	49	26	75	46	29	75	49	26	75	44	31	75
60	70	44	114	63	51	114	71	42	113	66	46	112
80	88	64	152	83	68	151	90	61	151	88	64	152
100	113	77	190	106	85	191	117	72	189	111	80	191

Tabelle A.11.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Half-Bottleneck* ohne zusätzliche Latenz

Kapazität	LIA	Reno	$\sum$	OLIA	Reno	$\sum$	CMT/RPv2	Reno	$\sum$	wVegas	Reno	$\sum$
20	23	14	37	21	16	37	24	13	37	21	17	38
40	45	31	76	41	34	75	47	28	75	42	33	75
60	67	46	113	61	53	114	70	42	112	60	51	111
80	89	62	151	81	71	152	91	57	148	78	68	146
100	106	83	189	99	90	189	111	73	184	80	85	165

Tabelle A.12.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Half-Bottleneck* für 5ms Latenz

Kapazität	LIA	Reno	$\sum$	OLIA	Reno	$\sum$	CMT/RPv2	Reno	$\sum$	wVegas	Reno	$\sum$
20	24	13	37	23	18	41	25	12	37	25	13	38
40	50	26	76	48	28	76	50	26	76	49	27	76
60	75	39	114	72	42	114	73	39	112	73	41	114
80	99	52	151	96	56	152	95	52	147	97	55	152
100	123	63	186	118	67	185	116	65	181	121	70	191

Tabelle A.13.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Half-Bottleneck* mit der RED-Queue ohne zusätzliche Latenz

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	24	13	37	23	15	38	24	13	37	23	14	37
40	48	28	76	45	30	75	46	27	73	46	29	75
60	71	41	112	65	45	110	68	38	108	65	45	110
80	92	54	146	83	61	144	87	49	136	83	58	141
100	110	66	176	98	76	174	104	59	163	81	74	155

Tabelle A.14.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Half-Bottleneck* mit der RED-Queue für 5ms Latenz

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	7	11	18	8	11	19	16	12	28	12	16	28
40	26	21	47	28	19	46	27	20	47	17	29	46
60	35	31	66	33	33	66	36	30	66	21	45	66
80	38	47	85	36	49	85	42	43	85	27	55	82
100	52	52	104	45	59	104	55	50	105	35	70	105

Tabelle A.15.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Half-Bottleneck* mit asynchronen Kapazitäten ohne zusätzliche Latenz

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	17	10	27	18	10	28	15	12	27	12	16	28
40	23	24	47	23	23	46	24	22	46	17	30	47
60	34	32	66	31	35	66	35	31	66	22	44	66
80	41	44	85	34	51	85	45	40	85	26	60	86
100	49	55	104	38	67	105	53	51	104	30	74	104

Tabelle A.16.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Half-Bottleneck* mit asynchronen Kapazitäten für 5ms Latenz

Kapazität	LIA	Reno	∑	OLIA	Reno	∑	CMT/RPv2	Reno	∑	wVegas	Reno	∑
20	17	11	28	17	11	28	16	12	28	11	16	27
40	23	24	47	22	25	47	24	22	46	15	31	47
60	28	35	64	23	39	62	31	31	62	22	41	63
80	33	46	79	25	46	71	35	43	78	30	54	84
100	43	54	97	43	58	101	44	52	96	34	60	94

Tabelle A.17.: Übertragungsraten zwischen Reno und einer MPTCP-Verbindung in Mbit/s auf dem *Half-Bottleneck* mit asynchronen Kapazitäten für 20ms Latenz

Literaturverzeichnis

[1] J. Postel, "Transmission control protocol," Internet Requests for Comments, RFC Editor, STD 7, September 1981, http://www.rfc-editor.org/rfc/rfc793.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[2] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "Tcp extensions for multipath operation with multiple addresses," Internet Requests for Comments, RFC Editor, RFC 6824, January 2013, http://www.rfc-editor.org/rfc/rfc6824.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6824.txt

[3] C. Raiciu, M. Handley, and D. Wischik, "Coupled congestion control for multipath transport protocols," Internet Requests for Comments, RFC Editor, RFC 6356, October 2011, http://www.rfc-editor.org/rfc/rfc6356.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6356.txt

[4] D. Wischik, M. Handley, and M. B. Braun, "The resource pooling principle," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 47–52, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1452335.1452342

[5] C. Raiciu, D. Wischik, and M. Handley, "Practical congestion control for multipath transport protocols," *University College London, London/United Kingdom, Tech. Rep*, 2009.

[6] T. Dreibholz, M. Becke, H. Adhari, and E. P. Rathgeb, "On the impact of congestion control for concurrent multipath transfer on the transport layer," in *Telecommunications (ConTEL), Proceedings of the 2011 11th International Conference on*, June 2011, pp. 397–404.

[7] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, "Architectural guidelines for multipath tcp development," Internet Requests for Comments, RFC Editor, RFC 6182, March 2011, chap. 7 (Interactions with Middleboxes), chap. 5.5 (Path Management). [Online]. Available: http://www.rfc-editor.org/rfc/rfc6182.txt

[8] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "Tcp extensions for multipath operation with multiple addresses," Internet Requests for Comments, RFC Editor,

RFC 6824, January 2013, http://www.rfc-editor.org/rfc/rfc6824.txt, chap. 2.2. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6824.txt

[9] M. Allman and V. Paxson and E. Blanton, "Tcp congestion control," Internet Requests for Comments, RFC Editor, RFC 5681, September 2009, http://www.rfc-editor.org/rfc/rfc5681.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5681.txt

[10] M. Allman, V. Paxson, and E. Blanton, "Tcp congestion control," Internet Requests for Comments, RFC Editor, RFC 5681, September 2009, http://www.rfc-editor.org/rfc/rfc5681.txt, chap. 3.1. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5681.txt

[11] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Tcp friendly rate control (tfrc): Protocol specification," Internet Requests for Comments, RFC Editor, RFC 5348, September 2008, http://www.rfc-editor.org/rfc/rfc5348.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5348.txt

[12] D. Wischik, M. Handley, and M. B. Braun, "The resource pooling principle," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, Sep., chap.

[13] M. Allman, "Tcp congestion control with appropriate byte counting (abc)," Internet Requests for Comments, RFC Editor, RFC 3465, February 2003, http://www.rfc-editor.org/rfc/rfc3465.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3465.txt

[14] F. Kelly and T. Voice, "Stability of end-to-end algorithms for joint routing and rate control," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 5–12, 2005. [Online]. Available: http://doi.acm.org/10.1145/1064413.1064415

[15] C. Raiciu, M. Handley, and D. Wischik, "Coupled congestion control for multipath transport protocols," Internet Requests for Comments, RFC Editor, RFC 6356, October 2011, http://www.rfc-editor.org/rfc/rfc6356.txt, chap. 5. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6356.txt

[16] R. Khalili, N. Gast, M. Popovic, and J.-Y. L. Boudec, "Opportunistic linked-increases congestion control algorithm for mptcp," Working Draft, IETF Secretariat, Internet-Draft draft-khalili-mptcp-congestion-control-05, July 2014, http://www.ietf.org/internet-drafts/draft-khalili-mptcp-congestion-control-05.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-khalili-mptcp-congestion-control-05.txt

[17] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec, "Mptcp is not pareto-optimal: Performance issues and a possible solution," *IEEE/ACM Trans. Netw.*, vol. 21, no. 5, pp. 1651–1665, Oct. 2013. [Online]. Available: http://dx.doi.org/10.1109/TNET.2013.2274462

[18] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, "Low extra delay background transport (ledbat)," Internet Requests for Comments, RFC Editor, RFC 6817, December 2012, http://www.rfc-editor.org/rfc/rfc6817.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6817.txt

[19] Y. Cao, M. Xu, and X. Fu, "Delay-based congestion control for multipath tcp," in *Proceedings of the 2012 20th IEEE International Conference on Network Protocols (ICNP)*, ser. ICNP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICNP.2012.6459978

[20] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "Tcp vegas: New techniques for congestion detection and avoidance," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 4, pp. 24–35, Oct. 1994. [Online]. Available: http://doi.acm.org/10.1145/190809.190317

[21] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained tcp retransmissions for datacenter communication," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 303–314, Aug. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594977.1592604

[22] Y.-C. Chan, C.-L. Lin, C.-T. Chan, and C.-Y. Ho, "Improving performance of tcp vegas for high bandwidth-delay product networks," in *2006 8th International Conference Advanced Communication Technology*, vol. 1, Feb 2006, pp. 6 pp.–464.

[23] "Linux kernel gnu general public license," https://www.kernel.org/pub/linux/kernel/COPYING, abgerufen am 11.11.2016.

[24] S. Arianfa, "Tcps congestion control implementation in linux kernel," *Aalto University*.

[25] R. Love, *Linux Kernel Development*. Pearson Education, 2010, chapter 5 - System Calls.

[26] ——, *Linux Kernel Development*. Pearson Education, 2010, chapter 2 - No (Easy) Use of Floating Point.

[27] ——, *Linux Kernel Development*. Pearson Education, 2010, chapter 18 - Debugging.

[28] C. Lea and J. Chu, "Non-blocking destination-based routing networks," Mar. 1 2011, uS Patent 7,898,957. [Online]. Available: https://www.google.com/patents/US7898957

[29] "Freebsd dummynet man page," https://www.freebsd.org/cgi/man.cgi?dummynet, 2002, abgerufen am 09.11.2016.

[30] A. N. A. C. L. R. Ugen J. S. Antsilevich, Poul-Henning Kamp, "Freebsd ipfw man page," https://www.freebsd.org/cgi/man.cgi?ipfw, 2012, abgerufen am 09.11.2016.

[31] B. Braden, D. D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang, "Recommendations on queue management and congestion avoidance in the internet," Internet Requests for Comments, RFC Editor, RFC 2309, April 1998, http://www.rfc-editor.org/rfc/rfc2309.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2309.txt

[32] C. Raiciu, M. Handley, and D. Wischik, "Coupled congestion control for multipath transport protocols," Internet Requests for Comments, RFC Editor, RFC 6356, October 2011, http://www.rfc-editor.org/rfc/rfc6356.txt, 1. Introduction. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6356.txt

[33] "Why is the multipath tcp scheduler so important ?" http://blog.multipath-tcp.org/blog/html/2014/03/30/why_is_the_multipath_tcp_scheduler_so_important.html, abgerufen am 15.11.2016.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 05. Januar 2017 Denis Lugowski