Koexistenz nicht-kompatibler TCP-Varianten in softwaredefinierten Netzen

Mario Hock
Institut für Telematik
Karlsruher Institut für Technologie
E-Mail: mario.hock@kit.edu

Mario Willems
Institut für Telematik
Karlsruher Institut für Technologie
E-Mail: mail@mariowillems.de

***Zusammenfassung*—Die im Internet eingesetzten Staukontrollverfahren sind überwiegend verlustbasiert. Das heißt, dass der reguläre Betrieb dieser Mechanismen vorsieht, Router-Warteschlangen bis zum Überlauf zu füllen (bzw. bis der Router gezielt beginnt Pakete zu verwerfen). Dies führt zu Paketverlusten und hohen Latenzen. Anwendungen wie hoch qualitative Videokonferenzen und moderne Cloud-Dienste sind allerdings auf geringe Latenzen, und somit auf kurze Warteschlangen angewiesen. Zu deren Unterstützung wären Staukontrollverfahren vorteilhaft, die ohne lange Warteschlangen auskommen. SDN-Netze können dabei helfen dieses Problem zu überwinden. Diese Arbeit untersucht inwieweit OpenFlow-/SDN-Switches dazu genutzt werden können, den Verkehr verschiedener TCP-Varianten voneinander zu trennen und somit die Anforderung nach Kompatibilität zwischen diesen Varianten aufzuheben. Experimentell wurde nachgewiesen, dass inkompatible TCP-Varianten dadurch separiert werden können, dass sie auf physisch getrennte Pfade umgeleitet werden. Stehen keine getrennten Pfade zur Verfügung, können Rate-limiter, sog. „OpenFlow-Meter“, zur Separierung genutzt werden. Hierbei sind allerdings einige Einschränkungen zu beachten.**

I. Einleitung

Das Internet wird zunehmend für interaktive, datenintensive Anwendungen genutzt, wie z. B. hochqualitative Videokonferenzen oder das „Screensharing“ von HD-Monitoren. Solche Anwendungen benötigen geringe Latenzen bei der Datenübertragung, produzieren aber so große Datenmengen, dass sie nicht einfach gegenüber dem restlichen Internetverkehr priorisiert werden können, wie es heute z. B. bei der Internettelefonie (VoIP) der Fall ist. Während die Übertragungskapazitäten im Internet kontinuierlich steigen, fallen die Latenzen allerdings nicht in vergleichbarem Maße. Einerseits sind die Latenzen durch die physische Signalausbreitung nach unten beschränkt. Andererseits zeigt die *„Bufferbloat“*-Problematik [1], dass auch die Warteschlangenverzögerung in Routern und Switches einen erheblichen Anteil daran hat.

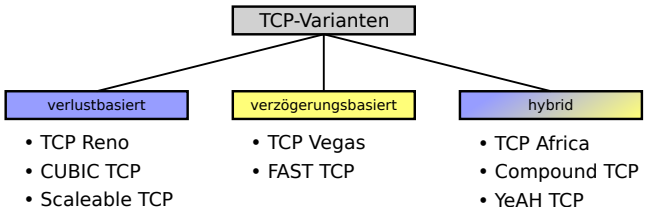
Durch die erhöhte Warteschlangenverzögerung erhöht sich auch die *Umlaufzeit (RTT)* und damit die vom Benutzer erfahrene Latenz. Durch volle Warteschlangen kommt es darüber hinaus häufig zu Paketverlusten. Laut [2] sorgt bei der Übertragung von Web-Inhalten ein Paketverlust dafür, dass sich die Übertragungszeit im Schnitt um das *Fünffache* erhöht. Die hohe Zahl kommt dadurch zustande, dass der Paketverlust häufig am Ende der Übertragung auftritt und daher erst verzögert erkannt wird.

Wünschenswert wäre daher eine Staukontrolle, die nur sehr wenige Paketverluste erzeugt und die Auslastung der Warteschlangen gering hält. Die in heute genutzten TCP-Varianten implementierten Staukontrollmechanismen erfüllen diese Anforderungen nicht. Alternative Verfahren, welche die Warteschlangenlänge gering halten, werden im Internet bisher nicht genutzt, da sie von den bestehenden TCP-Varianten unterdrückt werden. Diese Arbeit beschäftigt sich daher mit der Frage, wie das Netz Fairness zwischen inkompatiblen TCP-Varianten herstellen kann, sodass die gleichzeitige Nutzung alternativer Staukontrollverfahren ermöglicht wird.

Im Folgenden wird untersucht, welche Möglichkeiten in OpenFlow-basierten SDN-Netzen bestehen, um die Koexistenz inkompatibler TCP-Varianten zu ermöglichen.

II. Inkompatible TCP-Varianten

Da das Internet eine Stausituation nicht explizit an den Sender signalisiert[1], müssen die dort eingesetzten Staukontrollalgorithmen mit *impliziten* Stausignalen arbeiten. Hierbei werden meist *Paketverluste* als Stausignal genutzt. Sofern im Netz kein *Active Queue Managment* (wie z.B. RED [4]) genutzt wird, treten diese erst dann auf, wenn die Warteschlangen bereits überfüllt sind und das Netz Pakete verwerfen muss. Alternativ kann ein Stau auch über einen Anstieg der Verzögerung in Senderichtung, bzw. der Umlaufzeit, erkannt werden. Man unterscheidet daher zwischen TCP-Varianten mit *verlustbasierter*, *verzögerungsbasierter* und *hybrider* Staukontrolle.



Fairness zwischen konkurrierenden Verbindungen kann nur erreicht werden, wenn die genutzten Staukontrollalgo-

[1] Mittels dem in RFC 3168 [3] standardisierten *„Explicit Congestion Notification (ECN)“* kann das Netz den Sender zwar über einen Stau informieren, ECN wird allerdings im Internet nicht flächendeckend eingesetzt.

rithmen identisch oder zumindest zueinander *kompatibel* sind. RFC 2914 [5] definiert eine Verbindung als *„TCP-kompatibel“*, wenn sie bei Stau nicht mehr Daten ins Netz sendet als eine *TCP Reno*-Verbindung (RFC 2581 [6]). Eine *aggressivere* Verbindung würde die TCP-kompatiblen Verbindungen unterdrücken. Staukontrollalgorithmen, die ein solches Verhalten produzieren, sollen daher im Internet nicht genutzt werden. Ein Staukontrollalgorithmus wird als *aggressiv* bezeichnet, wenn er seine Senderate sehr schnell erhöht, oder sie während eines Stau nur leicht oder erst spät reduziert.

Verzögerungsbasierte TCP-Varianten (wie z. B. TCP Vegas [7]) erkennen einen drohenden Stau bereits sehr früh und reagieren *proaktiv* mit einer Reduktion der Senderate. Dies führt zu deutlich kürzeren Warteschlangen und zu einer sehr geringen Zahl an Paketverlusten. Konkurrieren sie jedoch mit den aggressiveren verlustbasierten Verfahren, bekommen sie keinen fairen Anteil an der Übertragungskapazität. Hybride Varianten (z. B. Compound TCP [8]) sind hingegen deutlich aggressiver als TCP Reno, solange sie keinen Anstieg der Verzögerung erkennen können. Steigt die Verzögerung über einen festgelegten Schwellenwert, reduzieren sie ihre Aggressivität auf das Niveau von TCP Reno, wodurch die Forderung nach TCP-Kompatibilität erfüllt wird. Gleichzeitig sorgen sie so dafür, dass sie einen fairen Anteil an den Übertragungskapazitäten bekommen. Erst bei einem Paketverlust reduzieren sie die Größe ihres Staukontrollfensters. Wie die verlustbasierten, füllen demnach auch die hybriden TCP-Varianten die Warteschlangen bis zum Überlauf.

Volle Warteschlangen führen zu einer merklichen Erhöhung der Umlaufzeit und häufigeren Paketverlusten. Dies beeinträchtigt *alle* Verbindungen, die durch diese Warteschlangen laufen. Eine weniger aggressive TCP-Variante ist ebenfalls von diesen Auswirkungen betroffen, auch wenn sie selbst nur gering zu der hohen Auslastung der Warteschlange beiträgt. Erhöht eine zurückhaltende Variante in einer solchen Konkurrenzsituation nicht die eigene Aggressivität, wird die zurückhaltende Variante von aggressiveren Varianten unterdrückt, wie in Abb. 1 veranschaulicht. In der Abbildung ist der Durchsatz und die RTT einer *TCP Reno*-Verbindung und einer *TCP Vegas*-Verbindung zu sehen, die miteinander konkurrieren. Da TCP-Kompatibilität im Internet über das Verhalten von TCP Reno definiert ist, dient es hier als Repräsentant einer für eine im Internet typische TCP-Verbindung. TCP Vegas hingegen ist ein rein verzögerungsbasiertes Verfahren, das versucht nur wenige eigene Pakete in den Warteschlangen zu halten und selbst kaum Paketverluste induziert. Das Experiment wurde in einem lokalen Testbett, bestehend aus zwei Sendern, einem Switch und einem Empfänger (vgl. Abb. 1a), durchgeführt. Die genaue Testumgebung wird in Abschnitt V-A nochmals detailliert beschrieben ist.

Von Beginn an fällt ein Großteil der zur Verfügung stehenden Übertragungskapazität auf die TCP Reno-Verbindung, die TCP Vegas-Verbindung hingegen belegt nur einen kleinen Teil der Gesamtkapazität (Abb. 1b). Die TCP Reno-Verbindung beginnt nun den Puffer an der gemeinsamen Engstelle zu füllen, was für beide Verbindungen eine linear steigende Umlaufzeit zur Folge hat (Abb. 1c). Nach etwa 50 s kommt es zu einem Paketverlust und TCP Reno reduziert das Staukontrollfenster, wodurch sich die Warteschlange leert und die Umlaufzeit wieder auf ca. 50 ms fällt. TCP Vegas erkennt dies und erhöht kurzzeitig die eigene Senderate, zieht sich aber sofort zurück, sobald die Warteschlange durch die TCP Reno-Verbindung erneut gefüllt wird. Es ist erkennbar, dass die Senderate der TCP Reno-Verbindung im Anschluss an einen Paketverlust schnell wieder steigt. Dies ist darin zu begründen, dass sich die Senderate aus dem Quotienten von Staukontrollfenster und RTT ergibt. Die Verringerung der RTT gleicht in diesem Experiment somit teilweise die Reduktion des Staukontrollfensters aus, was zu dem hier beobachtbaren schnellen Anstieg der Übertragungsrate führt.

Dieses Experiment zeigt, dass eine TCP-Variante, die eine geringe Auslastung der Warteschlange anstrebt, keinen fairen Anteil an der Übertragungskapazität bekommt, wenn sie an einer Engstelle mit den heute üblichen TCP-Varianten konkurriert. Durch die stark gefüllte Warteschlange an der Engstelle erhöht sich auch die Latenz der TCP Vegas-Verbindung, obwohl der Vegas-Algorithmus sicherstellt, dass nur wenige eigene Pakete in der Warteschlange enthalten sind.

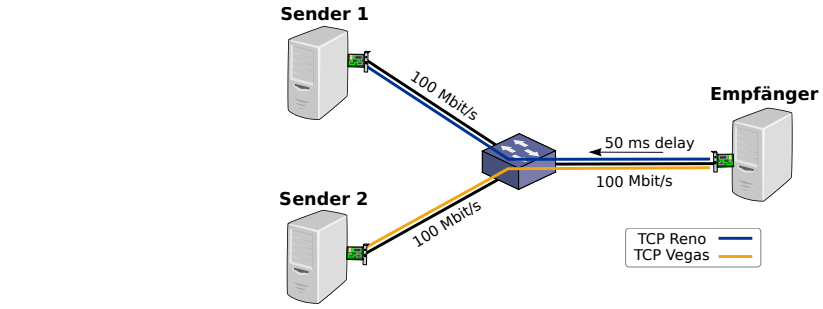
Im folgenden Abschnitt wird untersucht, welche Möglichkeiten in OpenFlow-basierten SDN-Netzen zur Verfügung stehen, um dieses Problem zu umgehen und die parallele Nutzung inkompatibler TCP-Varianten zu ermöglichen.

III. Lösungsmöglichkeiten mit OpenFlow

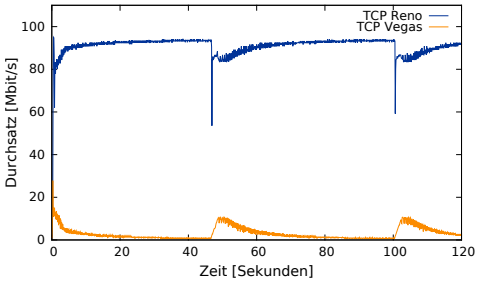
SDN-Netze unterscheiden sich insbesondere dadurch von klassischen Netzen, dass die Kontrollebene vom Datenpfad getrennt ist. Ein logisch zentraler *Controller* konfiguriert alle SDN-Switches in einem (Teil-)Netz. Dies ermöglicht ein besonders flexibles *Traffic Engineering*, das zur Separation inkompatibler TCP-Varianten genutzt werden kann. Stehen im Netz redundante Verbindungen zur Verfügung, kann der Controller inkompatible Verbindungen auf separaten Pfaden durch das Netz leiten. Stehen keine separaten Pfade zur Verfügung, können *Quality of Service (QoS)* Mechanismen genutzt werden, um auch auf einem gemeinsamen Übertragungsabschnitt die Fairness zwischen den Verbindungen zu verbessern. Diese Mechanismen sind im OpenFlow-Standard als *optional* gekennzeichnet und stehen daher nicht auf allen OpenFlow-Switches zur Verfügung.

Sowohl die Weiterleitungsregeln, als auch die QoS-Mechanismen werden vom Controller auf Basis von sogenannten *Flows* konfiguriert. Inkompatible TCP-Verbindungen müssen somit unterschiedlichen Flows zugewiesen werden. In einem Flow können die Datenpakete einer oder auch mehrerer TCP-Verbindungen zusammengefasst werden. Die Klassifikation findet dabei u. a. anhand von Feldern im IP- und TCP-Kopf statt.

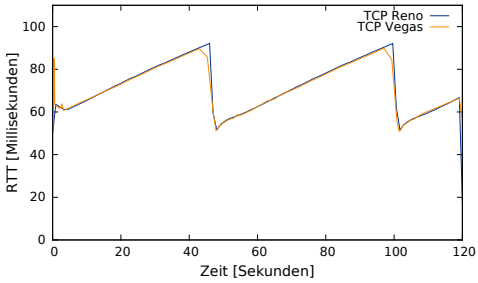
Um inkompatible TCP-Verbindungen unterschiedlichen Flows zuweisen zu können, muss dem Controller bekannt sein, welche TCP-Varianten von den entsprechenden Verbindung genutzt werden. Da bei TCP dem Kommunikationspartner nicht signalisiert wird, welche Variante genutzt wird,



(a) Testbett



(b) Durchsatz



(c) RTT

Abbildung 1: Ein TCP Vegas Sender wird von einem konkurrierenden TCP Reno Sender unterdrückt

kann der Controller diese Information nicht einfach beim Verbindungsaufbau ermitteln. Eine Klassifikation anhand des Sendeverhaltens, wie in [9], ist zwar generell möglich, aber sehr aufwändig und auch nicht sofort beim Verbindungsaufbau möglich, da die Verbindung eine Zeit lang beobachtet werden muss. Alternativ kann eine Anwendung, die eine inkompatible TCP-Verbindung aufbauen möchte und eine Sonderbehandlung durch das Netz wünscht, dies dem Controller signalisieren.

Da es in dieser Arbeit primär um die generelle Machbarkeit und die Untersuchung der Mechanismen geht, die eine parallele Nutzung inkompatibler TCP-Varianten ermöglichen, wurde hier der folgende Weg gewählt: Alle Verbindungen eines Endsystems nutzen (pro Netzwerkinterface) die gleiche TCP-Variante. Endsysteme, die eine inkompatible TCP-Variante nutzen, müssen sich zuvor beim Controller anmelden. Hierfür werden im Folgenden zwei *Verkehrsklassen* unterschieden: Loss und Delay

A. Verkehrsklassen

1) Die Klasse „Loss“ enthält die TCP-Varianten, welche die Warteschlange bis zum Überlauf füllen.
2) Die Klasse „Delay“ enthält die TCP-Varianten, die eine kurze Warteschlange anstreben.

In der Klasse Loss sind dementsprechend die heute im Internet üblicherweise genutzten verlustbasierten und hybriden TCP-Varianten enthalten. Da im Internet genutzte TCP-Varianten durch die Forderung der TCP-Kompatibilität zumindest bei Stau nicht aggressiver sein sollten als TCP Reno, werden in den Experimenten (Abschnitt V) grundsätzlich TCP Reno-Verbindungen als Vertreter dieser Klasse genutzt.

Die Klasse Delay ist insbesondere auf zukünftige TCP-Varianten ausgelegt, die einen hohen Durchsatz bei niedrigen Latenzen ermöglichen und nur wenige Paketverluste erzeugen. Bei Experimenten wird TCP Vegas als Vertreter dieser Klasse genutzt.

B. Mechanismen

Neben der Möglichkeit des *Traffic Engineering* bietet OpenFlow (ab der Version 1.3) zwei unterschiedliche *Quality of Service (QoS)* Mechanismen. Somit ergeben sich drei Mechanismen, die potentiell in der Lage sind die Koexistenz inkompatibler TCP-Varianten zu ermöglichen:

- Traffic Engineering
  1) Nutzung separater Pfade
- Quality of Service (QoS)
  2) Nutzung separater Warteschlangen mittels der *setQueue-Action*
  3) Nutzung von OpenFlow-*Meters*

1) Separate Pfade: Stehen in einem SDN-Netzwerk redundante Pfade zur Verfügung, können die Flows durch den Controller so programmiert werden, dass TCP-Verbindungen der Klasse Loss keine gemeinsamen Übertragungsabschnitte haben wie TCP-Verbindungen der Klasse Delay. Somit teilen sie sich keine gemeinsame Warteschlange und konkurrieren auch nicht um die verfügbare Übertragungskapazität am Ausgangsport. Daher ist keine Beeinträchtigung zwischen den Verbindungen zu erwarten. Dies wurde experimentell nachgewiesen und ist in Abschnitt V beschrieben.

Dieser Mechanismus kann mit allen OpenFlow-Switches genutzt werden, sofern redundante Pfade zur Verfügung

stehen. Ein Nachteil dieser Herangehensweise ist, dass die Ressourcen eines Pfads nur von der entsprechenden TCP-Variante genutzt werden können.

2) Separate Warteschlangen: Unterstützt der OpenFlow-Switch mehrere Warteschlangen pro Ausgangsport, können die Pakete eines Flows mittels der *setQueue-Action* in eine solche Warteschlange eingefügt werden. Die Warteschlangen müssen vorab auf anderem Weg (z. B. dem *OF-Config*-Protokoll [10]) konfiguriert werden.

Die Nutzung separater Warteschlangen ist konzeptionell ähnlich zur Nutzung separate Pfade. Inkompatible Verbindungen teilen sich auch hier keine gemeinsame Warteschlange und die verfügbare Übertragungskapazität kann fair auf die verschiedenen Warteschlangen aufgeteilt werden. Ein Vorteil gegenüber der Nutzung separater Pfade ist, dass Übertragungskapazität, die von einer der beiden Verkehrsklassen nicht genutzt wird, der anderen Verkehrsklasse zur Verfügung steht. Dies geschieht automatisch im Switch, ohne dass dafür eine Controller-Interaktion notwendig ist. Die *setQueue-Action* steht aber nicht auf allen OpenFlow-Switches zur Verfügung und konnte aus genau diesem Grund auch in dieser Arbeit noch nicht praktisch evaluiert werden. Eine experimentelle Untersuchung dieses Ansatzes ist aber für eine Folgearbeit geplant.

3) OpenFlow-Meters: Mit OpenFlow-*Meters* kann eine maximale Übertragungsrate für einen Flow oder eine Gruppe von Flows gesetzt werden. Dies kann genutzt werden um die Unterdrückung von weniger aggressiven TCP-Varianten durch verlustbasierte und hybride TCP-Varianten zu verhindern.

Anders als bei den beiden anderen Mechanismen werden die inkompatiblen Verbindungen hier nicht auf unterschiedliche Warteschlangen verteilt. Durch die Limitierung der Übertragungsrate werden die aggressiven Verbindungen aber effektiv daran gehindert die Warteschlange zu füllen. Freie Übertragungskapazität kann bei diesem Mechanismus nicht automatisch von den Flows einer anderen TCP-Variante genutzt werden, die Grenzen können aber vom Controller feingranular gesetzt und regelmäßig angepasst werden.

C. Weiterleitungsregeln

Es gibt zwei unterschiedliche Strategien, wie die Weiterleitungsregeln in SDN-Netzen festgelegt werden. Hierbei wird zwischen *Micro-Flows* und *Macro-Flows* unterschieden. Micro-Flows sind sehr spezifische Regeln, beispielsweise ein Flow pro TCP-Verbindung, oder pro kommunizierendem Endgerätepaar. Als Macro-Flows bezeichnet man allgemeinere Regeln, die viele potentielle Micro-Flows zusammenfassen, beispielsweise ein Flow zum Routing in ein anderes IP-Subnetz.

Alle drei Mechanismen können prinzipiell sowohl für Micro- als auch für Macro-Flows genutzt werden. Es muss jedoch sichergestellt sein, dass zwei inkompatible Verbindungen nicht innerhalb desselben Macro-Flows gruppiert werden.

IV. Umsetzung im SDN-Netz

Um neue Funktionen in ein SDN-Netz zu integrieren, muss der Controller erweitert werden. Dies wird üblicherweise in Form von *Apps* realisiert, die entweder über eine *Plug-in-Schnittstelle* Teil des Controller-Prozesses sein können oder externe Anwendungen sind, die mit dem Controller über eine sogenannte *Northbound-Schnittstelle* interagieren. Hierüber steht aber oft nur ein reduzierter Funktionsumfang zur Verfügung.

In einem realen SDN-Netz muss der Mechanismus *„Separate Pfade“* als Teil der Routing-App des Netzes implementiert werden. Zur Untersuchung der Funktionsfähigkeit des Mechanismus ist jedoch ein möglichst einfacher Testaufbau zu bevorzugen. Im Rahmen dieser Arbeit wurden daher statisch konfigurierten Flows genutzt, welche die beiden Verkehrsklassen über separate Pfade leiten.

Der Mechanismus *„OpenFlow-Meters“* muss dynamisch auf das Starten und das Beenden neuer Flows reagieren und die Meter-Begrenzungen dementsprechend anpassen. Daher wurde zur Untersuchung dieses Mechanismus eine App als Plug-in für den *OpenDaylight*-Controller entwickelt, die für die Programmierung der Flows und das Setzen von Meter-Begrenzungen zuständig ist.

A. Funktionsumfang der App

Das Einsatzgebiet der App beschränkt sich momentan auf Netze, die aus einem einzelnen OpenFlow-Switch und mehreren angeschlossenen Endsystemen bestehen. Die Flows werden durch die App nach einer *„SDN-Learning-Switch“*-Strategie programmiert. Das heißt, dass die App, ähnlich wie die Kontrollebene eines klassischen Switches, lernt welche Endsysteme an welchem Switchport angeschlossen sind. Möchten zwei Endsysteme miteinander kommunizieren, wird ein Flow mit den zugehörigen Sender- und Empfänger-MAC-Adressen programmiert und ggf. die Einstellungen der Meters angepasst (vgl. IV-B).

Es wird somit davon ausgegangen, dass ein Endsystem (pro Netzwerkschnittstelle) immer dieselbe TCP-Variante nutzt. Dies ist aber keine generelle Einschränkung der Mechanismen. Soll ein Endsystem mehrere inkompatible TCP-Varianten parallel nutzen können, müssen spezifischere Flows programmiert werden. Hierfür sind verschiedene Strategien denkbar. Beispielsweise könnten die Endsysteme unterschiedliche IPv6-Adressen für die beiden Verkehrsklassen nutzen. Anstelle von MAC-Adressen müssen die Flows in diesem Fall auf Basis von IP-Adressen programmiert werden, was über eine einfache Änderung an der App realisiert werden könnte.

Wie in Abschnitt III diskutiert, wird in dieser Arbeit davon ausgegangen, dass sich ein Endsystem, welches die Klasse Delay nutzen möchte, vorher beim Controller anmelden muss. Hierfür stellt die App eine *Rest-Schnittstelle* zur Verfügung.

B. Programmierung der Flows und Meter-Begrenzungen

Bevor der erste Flow programmiert wird, wird auf dem OpenFlow-Switch für jeden Switch-Port ein *Meter* konfiguriert. Dieser wird auf die maximale Übertragungsrate des

Ports gesetzt, sodass die Begrenzung effektiv deaktiviert ist. Dieses Vorgehen ermöglicht ein effizientes Setzen und Aufheben von dynamischen Begrenzungen für inkompatible Sender.

Empfängt der OpenFlow-Switch ein Datenpaket, das zu keinem Flow gehört, wird das Paket an den Controller weitergeleitet. Dies löst ein *„Packet-In"-Event* in der App aus, sodass diese einen passenden Flow programmieren kann.

Die Programmierung des Flows geht mit einer Anpassung der *Meter*-Einstellung einher. Wie in Abb. 2 dargestellt, wird die *Meter*-Begrenzung nur gesetzt, wenn sich Verbindungen beider Verkehrsklassen (Loss und Delay) einen Ausgangsport teilen. Somit wird keine Kapazität verschwendet, wenn gar keine Delay-Verbindungen existieren. Loss-Flows werden aber grundsätzlich an den (ggf. inaktiven) *Meter* gebunden. Kommt nun ein Delay-Flow hinzu, kann der *Meter* schnell umkonfiguriert werden, ohne dass alle Loss-Flows angepasst werden müssen.

Existieren sowohl Loss- als auch Delay-Flows, wird der *Meter* auf 50 % der Ausgangskapazität gesetzt. Eine Begrenzung in Abhängigkeit der *Anzahl* an Flows in den jeweiligen Klassen ist ebenfalls denkbar, aber in der App momentan nicht umgesetzt.

Die Flows sind mit einem *Timeout*-Wert versehen, sodass der OpenFlow-Switch einen Flow löscht, wenn dieser eine bestimmte Zeit lang nicht genutzt wurde. Der Switch benachrichtigt daraufhin den Controller. Analog zum Setzen neuer Flows überprüft die App dann, ob der *Meter* weiterhin benötigt wird, anderenfalls wird er wieder deaktiviert, indem er auf 100 % der Ausgangskapazität gesetzt wird.

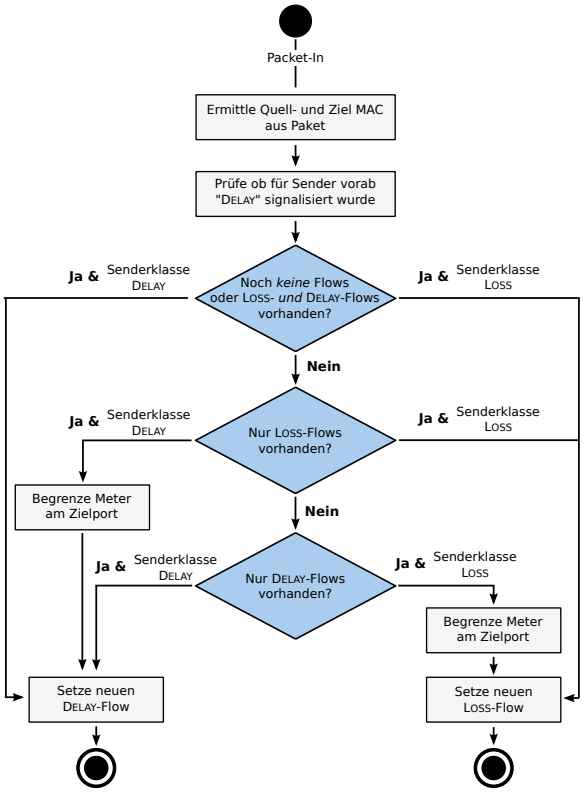


Abbildung 2: Setzen eines neuen Flows und Anpassung der *Meter*

V. Evaluation

In einem physischen Testbett wurden die beiden Ansätze

1) Nutzung separater physischer Pfade
2) Setzen dynamischer *Meter*-Begrenzungen mittels Controller-App

experimentell untersucht.

Alle Versuche wurden mehrfach wiederholt und zeigen reproduzierbare Ergebnisse. Aus Gründen der Übersichtlichkeit werden im Folgenden jeweils die Daten eines einzelnen Testlaufs dargestellt.

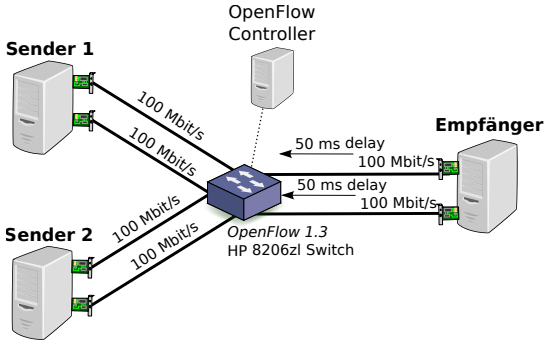
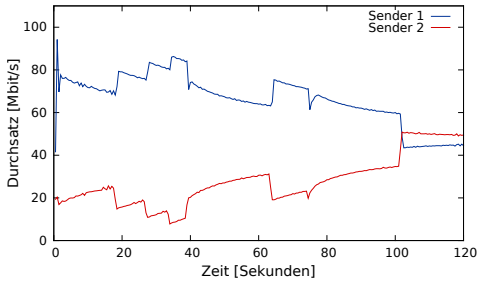


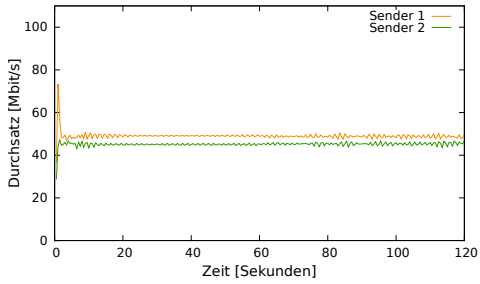
Abbildung 3: Testbett

A. Testbett

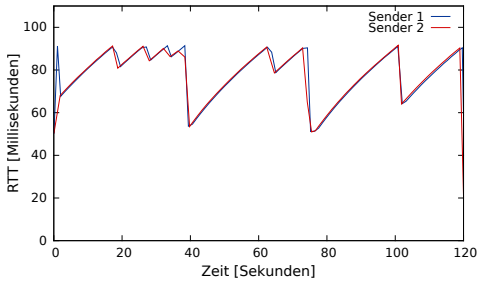
Das Testbett (Abb. 3) besteht aus einem OpenFlow-fähigen *HP 8206zl* Switch (Firmware-Version: K.15.15.0006, OpenFlow-Version: 1.3, nur v2 Module) und vier identischen Server-Rechnern, wovon einer als OpenFlow-Controller dient, einer als Empfänger und zwei als Sender. Die Sender und der Empfänger sind jeweils über zwei Netzwerkschnittstellen mit dem Switch verbunden. Die Server sind mit einer *Intel Xeon X3430 @ 2.40 GHz*-CPU und 12 GB-RAM ausgestattet, als Betriebssystem kommt *Ubuntu 14.04 LTS Server 64 bit* mit Linux-Kernelversion 3.13 zum Einsatz. Die Netzwerkschnittstellen wurden mittels `ethtool` auf 100 Mbit/s konfiguriert. Auf dem Empfänger wurde mittels `netem` eine künstliche Verzögerung von 50 ms eingestellt. Der Testverkehr wird mit dem Programm `Iperf` (Client und Server, Version: 2.0.5) erzeugt, welches immer versucht mit maximal möglicher Datenrate zu senden. Im Linux-Kernel wurde die Option `tcp_no_metrics_save=1` gesetzt, womit ein Cache von verbindungsspezifischen Variablen deaktiviert wird, welche beim Wiederholen von Testläufen u. U. die Ergebnisse verfälschen kann. Zur Untersuchung des Sendeverhaltens der TCP-Verbindungen wurde der Netzwerkverkehr auf den beiden Sendern jeweils mittels `tcpdump` aufgezeichnet.



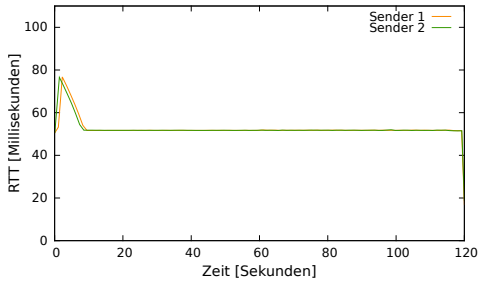
(a) TCP Reno *Durchsatz*



(b) TCP Vegas *Durchsatz*



(c) TCP Reno *RTT*



(d) TCP Vegas *RTT*

Abbildung 4: Zwei TCP Reno Sender und zwei TCP Vegas Sender getrennt über separate physische Pfade

B. Separate Pfade

Zur Überprüfung der Annahme, dass sich inkompatible TCP-Varianten auf separaten physischen Pfaden gegenseitig nicht negativ beeinflussen, wurde folgendes Experiment durchgeführt:

Sender 1 und Sender 2 nutzen jeweils eine TCP Reno- und eine TCP Vegas-Verbindung zur Übertragung von Daten an den Empfänger. Beide Sender nutzen jeweils eine der beiden Netzwerkschnittstellen für die TCP Reno-Verbindung und die andere Schnittstelle für die TCP Vegas-Verbindung. Auf dem Switch wurden statische Flows gesetzt, welche die beiden TCP Reno-Verbindungen auf die erste, und die beiden TCP Vegas-Verbindungen auf die zweite Netzwerkschnittstelle des Empfänger leiten.

Abb. 4 zeigt Durchsatz und RTT der vier Verbindungen. In Abb. 4a ist zu erkennen, wie die beiden TCP Reno-Verbindungen um die zur Verfügung stehende Übertragungskapazität auf ihrem Pfad konkurrieren. Dabei füllen sie die Warteschlange immer wieder bis zum Paketverlust, was an der erhöhten RTT in Abb. 4c zu erkennen ist. Die RTT der beiden TCP Vegas-Verbindungen (Abb. 4c) ist, nach der initialen *Slow Start*-Phase, gleichbleibend gering. Die stark gefüllte Warteschlange auf dem TCP Reno-Pfad hat somit keine negativen Auswirkungen auf die TCP Vegas-Verbindungen.

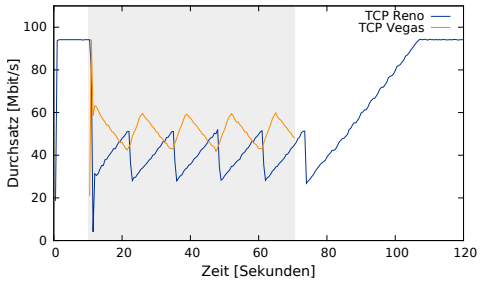
Wie Abb. 4b zeigt, wird die Übertragungskapazität des zweiten Pfades fair auf die beiden TCP Vegas-Verbindungen verteilt. Es ist zu erkennen, dass der Durchsatz der beiden TCP Vegas-Verbindungen nicht exakt gleich ist, dieses Verhalten ist für TCP Vegas jedoch nicht unüblich und kann auch in einfachen Experimenten zur Intraprotokollfairness von TCP Vegas beobachtet werden. Beim Vergleich von Abb. 4a und Abb. 4b ist gut zu erkennen, dass die Fairness zwischen den beiden TCP Vegas Sendern deutlich besser ist, als zwischen den TCP Reno Sendern und dass TCP Vegas ein wesentlich gleichmäßigeres Sendeverhalten hat als TCP Reno.

Das Experiment hat gezeigt, dass TCP Vegas-Verbindungen, die in einem OpenFlow-Switch mittels statischer Flow-Regeln einen eigenen Pfad zugewiesen bekommen, nicht von TCP Reno-Verbindungen unterdrückt werden, die auf einem anderen Pfad weitergeleitet werden. Die Funktionsweise des Mechanismus hängt hierbei nicht von der Anzahl der OpenFlow-Switches, oder von der Tatsache, dass die Flow-Regeln statisch programmiert wurden, ab. Somit können die Ergebnisse auf den Einsatz des Mechanismus in realen SDN-Netzen, bei denen ein Controller die Flow-Regeln dynamisch programmiert, verallgemeinert werden.

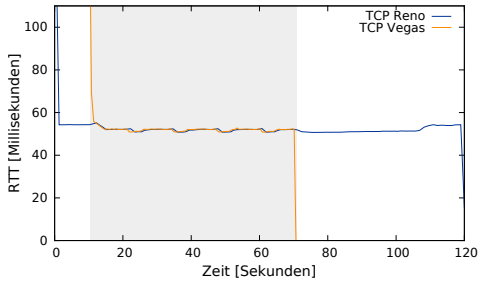
C. OpenFlow-Meters *mittels App*

Zur Evaluation des *Meter*-Ansatzes und der in Abschnitt IV beschriebenen App, wurde ein weiteres Experiment durchgeführt. Anders als bei den getrennten physischen Pfaden wird auf jedem Server nur eine Netzwerkschnittstelle genutzt. Sender 1 überträgt Daten mittels TCP Reno zum Empfänger, Sender 2 mittels TCP Vegas. Die beiden Verbindungen konkurrieren somit auf dem Übertragungsabschnitt zwischen Switch und Empfänger miteinander. Die Flows und *Meter*-Begrenzungen werden dynamisch von der App gesetzt, wie in Abschnitt IV-B beschrieben.

Um das dynamische Verhalten der App beobachten zu können, startet Sender 1 (TCP Reno) zuerst und sendet



(a) Durchsatz



(b) RTT

Abbildung 5: Durch den Einsatz eines OpenFlow-*Meters*, der über eine SDN-App dynamisch gesetzt wird, wird die TCP Vegas-Verbindung von der konkurrierenden TCP Reno-Verbindung nicht unterdrückt.

insgesamt für `120 s`. Sender 2 beginnt `10 s` später und sendet `60 s` lang.

Der Durchsatz beider Sender ist in Abb. 5a dargestellt, die Zeit in der Sender 2 aktiv ist, ist in der Abbildung grau unterlegt. Zunächst kann Sender 1 (TCP Reno) die volle Übertragungskapazität nutzen. Nach `10 s` kommt Sender 2 (TCP Vegas) hinzu und die App setzt die *Meter*-Begrenzung auf 50 Mbit/s. Daraufhin bricht der Durchsatz der TCP Reno-Verbindung stark ein und nähert sich dann wieder linear der Grenze von 50 Mbit/s an. Wird die Grenze überschritten, kommt es durch den Meter zu einem Paketverlust und TCP Reno halbiert das Staukontrollfenster. Dies wiederholt sich so lange periodisch, bis der *Meter* wieder deaktiviert wird. Danach steigt die Übertragungsrate linear an, bis die volle Übertragungskapazität des Senders erreicht ist.

Nach einem Paketverlust nutzt Sender 1 deutlich weniger Übertragungskapazität als ihm zusteht. Die TCP Vegas-Verbindung (von Sender 2) erkennt dies und nutzt die freie Kapazität ohne jedoch die koexistierende TCP Reno-Verbindung zu beeinträchtigen. Steigt die Rate von TCP Reno an, gibt TCP Vegas die entsprechende Kapazität frei. Sendet TCP Reno mit einer Übertragungsrate nahe der *Meter*-Begrenzung, fällt die Rate von TCP Vegas knapp unter die von TCP Reno.

TCP Reno wird durch den *Meter* daran gehindert die Warteschlange des Switches merklich zu füllen, was an der gleichbleibenden Umlaufzeit (Abb. 5b) zu erkennen ist. Es fällt auf, dass sich auch bei inaktivem *Meter*, also bevor die TCP Vegas-Verbindung aufgebaut ist und nach deren Ende, keine Warteschlangenverzögerung aufbaut. Dies liegt daran, dass Sender 1 der einzige Sender ist und nicht schneller senden kann, als der Switch die Daten weiterleitet.

Im Durchschnitt erreicht die TCP Vegas-Verbindung eine höhere Übertragungsrate als die TCP Reno-Verbindung. Dies hat zwei Gründe. Zum einen ist die TCP Vegas-Verbindung nicht durch einen Meter begrenzt und kann somit mehr als 50 % der Übertragungskapazität nutzen, sofern die TCP Reno-Verbindung ihren Anteil nicht voll ausschöpft. Zum anderen ist das Verhalten der Vegas-Staukontrolle auf eine geringe Pufferauslastung optimiert. Stellt sie einen drohenden Stau fest, reduziert sie ihre Senderate nur leicht aber dafür frühzeitig. Dies führt dazu dass die TCP Vegas-Verbindung die freie Kapazität sehr gut ausschöpfen kann ohne eine Überlastung des Netzes hervorzurufen. TCP Reno hingegen reagiert nur auf Paketverluste und geht dann von einer überfüllten Warteschlange aus. Daraufhin wird das Staukontrollfenster der Verbindung halbiert, was bei der kurzen Warteschlange auch ungefähr zu einer Halbierung der Senderate führt.

Das Verhalten von TCP Reno bei einer geringen Pufferauslastung unterscheidet sich somit von dem in Abschnitt II beschrieben Verhalten, bei dem die TCP Reno-Verbindung den Puffer ungehindert füllen konnte (vgl. Abb. 1 in Abschnitt II). Dort führte der Abbau der Warteschlangenverzögerung nach einem Paketverlust zu einer merklichen Reduktion der RTT. Dadurch hatte die Halbierung des Staukontrollfensters wiederum einen deutlich geringeren Effekt auf die Senderate.

Das Experiment hat somit gezeigt, dass durch den Einsatz von OpenFlow-*Meters* verhindert werden kann, dass sich inkompatible TCP-Varianten gegenseitig unterdrücken. Eine TCP Reno-Verbindung kann die ihr zustehende Übertragungskapazität jedoch nur zu circa 75 % nutzen. Die ungenutzte Kapazität kommt der konkurrierenden TCP Vegas-Verbindung zugute.

Stehen in einem SDN-Netzwerk, zumindest Abschnissweise, keine separaten Pfade zur Verfügung, stellt der Einsatz von OpenFlow-Meters einen alternativen Mechanismus dar, um die Fairness beim Parallelbetrieb inkompatibler TCP-Varianten zu verbessern.

D. Begrenzung von TCP Vegas-Verbindungen

Wie das vorangegangene Experiment zeigt, wird die Senderate der TCP Reno-Verbindung nur durch den OpenFlow-*Meter*, nicht jedoch durch die konkurrierende TCP Vegas-Verbindung begrenzt. Es ist somit nicht nötig, die TCP Vegas-Verbindung ebenfalls durch einen *Meter* zu begrenzen um die Fairness zu verbessern. Wird eine TCP Vegas-Verbindung dennoch durch einen *Meter* begrenzt, wird das übliche Protokollverhalten gestört.

Abb. 6 zeigt den Durchsatz einer TCP Vegas-Verbindung von Sender 1 zum Empfänger. Der zugehörige Flow wurde durch einen *Meter* auf 50 Mbit/s begrenzt. Da sich bei Annäherung an diese Rate aber keine

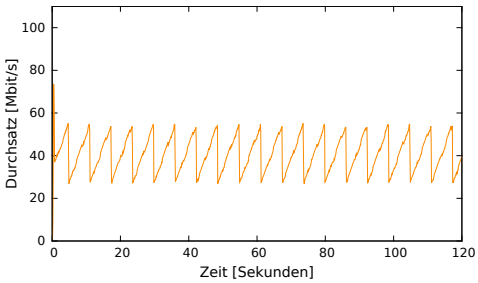


Abbildung 6: TCP Vegas begrenzt durch OpenFlow Meter

Warteschlange aufbaut, bekommt TCP Vegas kein frühes Stausignal und kann seine Senderate daher nicht proaktiv reduzieren. Stattdessen wird die Senderate solange erhöht, bis der *Meter* beginnt Pakete zu verwerfen. Die TCP Vegas-Verbindung fällt somit auf ein TCP Reno-artiges Verhalten zurück, was zu einer schlechten Ausnutzung der zur Verfügung stehenden Übertragungskapazität führt. Daher sollten OpenFlow-*Meter* nur zur Begrenzung von Verbindungen mit verlustbasierter Staukontrolle eingesetzt werden.

VI. Verwandte Arbeiten

Die hybride TCP-Variante *YeAH TCP* [11] versucht ebenfalls die Warteschlangenverzögerung gering zu halten, schützt sich aber eigenständig davor von aggressiveren Varianten unterdrückt zu werden. Eine YeAH TCP-Verbindung nutzt dazu eine Heuristik, mit der sie erkennt, ob sie gerade mit einer aggressiveren TCP-Variante konkurriert. Ist dies nicht der Fall, versucht sie die Warteschlangenverzögerung gering zu halten. Anderenfalls wechselt YeAH TCP in einen *Kompatibilitätsmodus* und verhält sich wie ein verlustbasiertes Verfahren. Das Ziel eine geringe Warteschlangenverzögerung zu verursachen wird somit gezielt aufgegeben, wenn die Gefahr besteht von einer anderen Verbindung unterdrückt zu werden.

Beim *Active Queue Management (AQM)* versucht das Netz für eine geringe Auslastung der Warteschlangen zu sorgen, indem die Router frühzeitig Stausignale an die Sender übermitteln. Hierzu kann *Explicit Congestion Notification (ECN)* (RFC 3168 [3]) zum Einsatz kommen, sofern dies von Sender und Empfänger unterstützt wird. Ist dies nicht der Fall, kann das AQM-Verfahren auch Pakete verwerfen, was vom Sender als implizites Stausignal interpretiert wird. Um die Fairness zwischen unterschiedlichen Verbindungen zu verbessern, können AQM-Verfahren mit speziellen *Scheduling-Strategien*, wie z. B. *Stochastic Fairness Queuing (SFQ)* [12] kombiniert werden (vgl. [13]). Ein solches kombiniertes Verfahrens ist *AFCD* ([14]). Am Beispiel der TCP-Varianten Reno und Cubic zeigt [14], dass damit auch die Fairness zwischen Verbindungen unterschiedlicher TCP-Varianten verbessert wird. Cubic ist, wie Reno, ebenfalls verlustbasiert. In den Experimenten mit verschiedenen AQM-Verfahren (u. a. RED [4] und CoDel [15]) hatte sich aber eine deutliche Unfairness zwischen den beiden Verbindungen gezeigt, die durch das AFCD-Verfahren wieder aufgehoben werden konnte. Im OpenFlow-Standard ist der Einsatz von AQM-Verfahren jedoch nicht vorgesehen.

Die Nutzung von separaten Warteschlangen für unterschiedliche Verkehrsklassen und der Einsatz von Ratelimitern ist nicht nur in SDN-Netzen möglich. In klassischen IP-Netzen stehen zwei Ansätze für die Umsetzung solcher QoS-Mechanismen zur Verfügung. Bei den *Differentiated Services (DiffServ)* (RFC 2474 [16]) werden IP-Pakete von den Routern am Netzrand klassifiziert und es wird im IP-Paket ein *DSCP-Codepoint* gesetzt, der von Routern im Netzinneren entsprechend interpretiert wird. Bei den *Integrated Services (IntServ)* RFC 2210 [17] können Endsysteme Netzressourcen mit Hilfe des *RSVP*-Protokolls reservieren. In einem gemischten Netz, bestehend aus OpenFlow-Switches und klassischen Routern, können sich die jeweiligen QoS-Mechanismen gegenseitig ergänzen. Die DSCP-Codepoints können auch von OpenFlow-Switches gesetzt und interpretiert werden. Eine Aushandlung über das *RSVP*-Protokoll könnte durch den SDN-Controller realisiert werden.

VII. Zusammenfassung und Ausblick

In dieser Arbeit wurde untersucht welche Möglichkeiten in *OpenFlow*-basierten SDN-Netzen bestehen, um die Koexistenz inkompatibler TCP-Varianten zu ermöglichen. SDN bietet eine hohe Flexibilität bei der Wegewahl im Netz. Dies kann genutzt werden, um inkompatible Verbindungen auf separaten Pfaden durch das Netz zu leiten. Die Verbindungen sind somit effektiv voneinander getrennt und beeinflussen sich nicht gegenseitig. Allerdings ist dieser Ansatz nur möglich, wenn redundante Pfade im Netz zur Verfügung stehen.

Stehen keine redundanten Pfade zur Verfügung, können bei einigen OpenFlow-Switches stattdessen auch QoS-Mechanismen genutzt werden. Diese Arbeit konzentriert sich hierbei auf die Untersuchung von OpenFlow-*Meters*. Für den *OpenDaylight* SDN-Controller wurde in diesem Zusammenhang eine *App* entwickelt, die dynamisch Flows programmiert und bei Bedarf *Meter*-Begrenzungen setzt. Dies wurde in einem Testbett auf physischer Hardware evaluiert. Dabei hat sich gezeigt, dass verzögerungsbasierte TCP-Varianten mithilfe von OpenFlow-*Meters* vor der Unterdrückung durch verlustbasierte Varianten geschützt werden können und geringe Verzögerungen erreicht werden.

Bei den Experimenten hat sich weiterhin herausgestellt, dass eine *Meter*-Begrenzung nur für verlustbasierte Verbindungen sinnvoll ist. Werden verzögerungsbasierte Verbindungen durch einen *Meter* begrenzt, wird ihr Protokollverhalten gestört. Sie benötigen den Aufbau einer leichten Warteschlangenverzögerung als Stausignal. Der *Meter* verwirft jedoch Pakete bevor sich eine entsprechende Verzögerung aufbaut. Da sich verzögerungsbasierte Varianten bei einem drohenden Stau frühzeitig zurückziehen, ist es dennoch nicht notwendig diese durch einen *Meter* zu begrenzen.

Für verlustbasierte TCP-Varianten sind Paketverluste hingegen das reguläre Stausignal. Sie sind allerdings darauf ausgelegt, dass ein Paketverlust erst bei einer Überfüllung der Warteschlange auftritt. Da der *Meter* bereits Pakete verwirft bevor sich die Warteschlange füllt, führt dies zu einem etwas niedrigeren Durchsatz. Die nicht genutzte Kapazität kann jedoch stattdessen von einer konkurrierenden

verzögerungsbasierten Verbindung genutzt werden und ist somit nicht verloren.

Als weiteren QoS-Mechanismus sieht der OpenFlow-Standard, neben den *Meters*, die Nutzung separater Warteschlangen mittels der *setQueue-Action* vor. Hierbei sind ähnlich gute Ergebnisse zu erwarten wie bei der Nutzung separater Pfade. Darüber hinaus bieten separate Warteschlangen den Vorteil, dass ungenutzte Übertragungskapazität automatisch auf die restlichen Warteschlangen verteilt wird. Eine experimentelle Untersuchung dieses Ansatzes ist für eine Folgearbeit geplant.

Danksagung

Diese Arbeit wurde durch das bwNET100G+ Projekt des Ministeriums für Wissenschaft, Forschung und Kunst Baden-Württemberg (MWK) unterstützt. Die Autoren allein sind für den Inhalt der Arbeit verantwortlich.

Literatur

[1] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *Queue*, vol. 9, no. 11, p. 40, 2011.

[2] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: The virtue of gentle aggression," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 159–170, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534169.2486014

[3] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168 (Proposed Standard), Internet Engineering Task Force, Sep. 2001, updated by RFCs 4301, 6040. [Online]. Available: http://www.ietf.org/rfc/rfc3168.txt

[4] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *Networking, IEEE/ACM Transactions on*, vol. 1, no. 4, pp. 397–413, Aug 1993.

[5] S. Floyd, "Congestion Control Principles," RFC 2914 (Best Current Practice), Internet Engineering Task Force, Sep. 2000, updated by RFC 7141. [Online]. Available: http://www.ietf.org/rfc/rfc2914.txt

[6] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC 2581 (Proposed Standard), Internet Engineering Task Force, Apr. 1999, obsoleted by RFC 5681, updated by RFC 3390. [Online]. Available: http://www.ietf.org/rfc/rfc2581.txt

[7] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "Tcp vegas: New techniques for congestion detection and avoidance," in *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, ser. SIGCOMM '94. New York, NY, USA: ACM, 1994, pp. 24–35. [Online]. Available: http://doi.acm.org/10.1145/190314.190317

[8] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound tcp approach for high-speed and long distance networks," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, April 2006, pp. 1–12.

[9] P. Yang, W. Luo, L. Xu, J. Deogun, and Y. Lu, "Tcp congestion avoidance algorithm identification," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, June 2011, pp. 310–321.

[10] O. N. Foundation, "Openflow management and configuration protocol," Oktober 2013, https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf.

[11] A. Baiocchi, A. P. Castellani, and F. Vacirca, "Yeah-tcp: yet another highspeed tcp," in *Proc. PFLDnet*, vol. 7, 2007, pp. 37–42.

[12] P. McKenney, "Stochastic fairness queueing," in *INFOCOM '90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration. Proceedings, IEEE*, Jun 1990, pp. 733–740 vol.2.

[13] F. Baker and R. Pan, "On Queuing, Marking, and Dropping," Internet-Draft, Internet Engineering Task Force, Work in Progress, May 2015, https://tools.ietf.org/html/draft-ietf-aqm-fq-implementation-02.

[14] L. Xue, S. Kumar, C. Cui, P. Kondikoppa, C.-H. Chiu, and S.-J. Park, "Afcd: An approximated-fair and controlled-delay queuing for high speed networks," in *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, July 2013, pp. 1–7.

[15] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, "Controlled Delay Active Queue Management," Internet-Draft, Internet Engineering Task Force, Work in Progress, April 2015, https://tools.ietf.org/html/draft-ietf-aqm-codel-01.

[16] K. Nichols, S. Blake, F. Baker, and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," RFC 2474 (Proposed Standard), Internet Engineering Task Force, Dec. 1998, updated by RFCs 3168, 3260. [Online]. Available: http://www.ietf.org/rfc/rfc2474.txt

[17] J. Wroclawski, "The Use of RSVP with IETF Integrated Services," RFC 2210 (Proposed Standard), Internet Engineering Task Force, Sep. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2210.txt