Bachelorarbeit

Optimierung rekonfigurierbarer Netzwerke für Bulk-Transfers mit gängigen Scheduling-Algorithmen in Mininet

Julian Huhn

22. Juni 2023

Begutachtung:
Prof. Dr. Dr. Klaus-Tycho Förster
Prof. Dr. Peter Buchholz

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl IV - Praktische Informatik
https://ls4-www.cs.tu-dortmund.de/

Inhaltsverzeichnis

1. Abstract

Die vorliegende Bachelorarbeit beschäftigt sich mit der Optimierung rekonfigurierbarer Netzwerke für Bulk-Transfers unter Verwendung gängiger Scheduling-Algorithmen in Mininet. Rechenzentren sind heute eine wichtige Säule der kritischen Infrastruktur und müssen große Mengen an Traffic bewältigen können. Optische Rechenzentren mit dynamischen Netzwerktopologien haben sich als vielversprechende Lösung entwickelt. Diese Arbeit stellt rekonfigurierbare Netzwerke vor, die es ermöglichen, die Netzwerktopologie dynamisch anzupassen, insbesondere unter Verwendung der Optical Circuit Switching-Technologie (*OCS*). Es werden verschiedene Algorithmen zur Steuerung der Netzwerkrekonfiguration sowie Scheduling-Algorithmen zur Optimierung der Reihenfolge von Bulk-Transfers untersucht. Die Algorithmen First Come First Serve (*FCFS*), Shortest Job First (*SJF*) und Longest Job First (*LJF*) werden im Detail betrachtet. Zusätzlich wird das Routing-Protokoll Open Shortest Path First (*OSPF*) vorgestellt.

Der Schwerpunkt dieser Bachelorarbeit besteht aus der Durchführung und anschließenden Diskussion der Ergebnisse von experimentellen Untersuchungen in Mininet. Im Rahmen dieser Experimente wird untersucht, inwiefern sich mit rekonfigurierbaren Kanten und der Verwendung von verschiedenen Scheduling-Algorithmen für Bulk-Transfers anhand von definierten Performance-Metriken Performancegewinne erzielen lassen. Hierzu wird für 60 Experimentvarianten, die sich anhand einer Kombination aus drei Scheduling-Algorithmen, vier Typologien und fünf zufällig generierten Mengen an Bulk-Transfers ergeben und jeweils zehn mal wiederholt werden, eine Versuchsreihe von insgesamt 600 Versuchen durchgeführt. Für die experimentellen Untersuchungen werden Python-Bibliotheken und Tools wie NetworkX, iperf3 und Mininet kurz vorgestellt und verwendet.

Die Diskussion der Ergebnisse zeigt, dass die Scheduling-Algorithmen hinsichtlich der zugrunde liegenden Topologien verschiedene Charakteristiken aufweisen und hinsichtlich gewisser Metriken ein Verbesserungspotential besitzen. So ist unter Umständen *LJF* in rekonfigurierbaren Netzwerken zu bevorzugen, die ihre dynamischen Kanten anhand großer Transfer konfiguriert haben, während *SJF* in Netzwerken, mit dem Ziel der Minimierung allgemeiner Pfadlängen, effizienter ist.

Außerdem zeigt sich, dass hybride Topologien mit rekonfigurierbaren Kanten und Scheduling einen signifikanten Leistungsgewinn über statische Netzwerke in den betrachteten Metriken erzielen, was zu einer

gesamtheitlich besseren Netzwerk-Performance führt. Insbesondere der Rekonfigurationsalgorithmus *DemandFirst* erweist sich als effizient, auch wenn das volle Potential rekonfigurierbarer Netzwerke hierdurch noch nicht völlig ausgeschöpft wird. Die Priorisierung großer Transfers ermöglicht es, diese direkt zum Zielknoten zu leiten und entlastet gleichzeitig längere Pfade, was auch kleinere Transfers begünstigt.

2. Motivation

2.1. Einleitung

Rechenzentren sind Teil kritischer Infrastruktur unserer Gesellschaft geworden [6]. Der Bedarf große Mengen an Netzwerk-Traffic zu bedienen ist rasant gestiegen, weshalb neue Arten entwickelt werden, wie die Netzwerke innerhalb der Rechenzentren diese Datenmengen bewältigen können. Unter anderem sind neue *optische* Rechenzentren entstanden, die *statische* und *dynamische* Netzwerktopologien kombinieren [13].

In dieser wissenschaftlichen Arbeit werden rekonfigurierbare Netzwerke betrachtet, die es ermöglichen, die Netzwerktopologie dynamisch anzupassen. Rekonfigurierbare Netzwerke werden vorgestellt und diskutiert, in deren Kontext besonders auf die Optical Circuit Switching-Technologie (*OCS*) eingegangen wird. Darüber hinaus werden verschiedene Algorithmen betrachtet, die zur Steuerung der Netzwerkrekonfiguration eingesetzt werden können.

Die von modernen Anwendungen erzeugten großen Datenmengen performant zu übertragen ist ein wichtiger Aspekt bei der Analyse von Bulk-Transfers. Des Weiteren werden verschiedene Scheduling-Algorithmen betrachtet. Diese dienen der Optimierung der Reihenfolge der Bulk-Transfers. Im Rahmen der Bachelorarbeit werden die Algorithmen First Come First Serve (*FCFS*), Shortest Job First (*SJF*) und Longest Job First (*LJF*) untersucht.

Darüber hinaus werden Routing-Protokolle betrachtet, insbesondere das Open Shortest Path First Protokoll (*OSPF*), das für die effiziente Datenübertragung entlang der kürzesten Pfade in Computernetzwerken eingesetzt wird. Im Rahmen der experimentellen Untersuchungen werden Bibliotheken und Tools wie NetworkX, iperf3 und Mininet vorgestellt und verwendet.

Das übergeordnete Ziel der vorliegenden Bachelorarbeit ist es, experimentelle Untersuchungen in Mininet durchzuführen, um die Auswirkungen von Scheduling-Algorithmen auf die Optimierung von rekonfigurierbaren Netzwerken für Bulk-Transfers zu untersuchen. Durch die Evaluierung der Technologien und Algorithmen wird ein Beitrag zur effizienten Gestaltung von rekonfigurierbaren Netzwerken geleistet und somit auch zur Weiterentwicklung von leistungsfähigen Rechenzentren beigetragen.

2.2. Struktur der Arbeit

Nach der Motivation werden in Kapitel 3 die Grundlagen der zugrunde liegenden Technologien und Algorithmen vorgestellt und diskutiert.

Darauf folgt in Kapitel 4 die experimentelle Untersuchung in Mininet. Dabei wird zunächst die Methodik der experimentellen Untersuchung untergliedert in Topologien, Workloads und Performance-Metriken vorgestellt, bevor darauffolgend die Reproduzierbarkeit des Experiments kritisch analysiert wird. Zum Abschluss des Kapitels werden die Ergebnisse der Experimente vorgestellt und in den Kontext der Optimierung rekonfigurierbarer Netzwerke für Bulk-Transfers mit gängigen Scheduling-Algorithmen in Mininet eingeordnet.

In Kapitel 5 werden die Ergebnisse der Experimente im Kontext der vorliegenden Bachelorarbeit diskutiert und es werden erste Handlungsempfehlungen für Peformancesteigerungen ausgesprochen.

Im abschließenden Kapitel 6 werden die Erkenntnisse der Forschungsarbeit zusammengefasst und es werden weitere Forschungsschwerpunkte benannt, die auf den Ergebnissen aufbauen und die Ergebnisse weiterentwickeln können.

2.3. Contributions

Diese Arbeit trägt einen Teil dazu bei das bisher nur wenig untersuchte Themengebiet der effizienten Einstellung hybrider Netzwerke aus statischer und rekonfigurierbarer Netzwerk-Hardware [11] weiter zu erforschen. Die kombinierte Betrachtung rekonfigurierbarer Netzwerke mit der Bearbeitungsreihenfolge von Bulk-Transfers mittels Scheduling ist weitgehend unerforscht. Hier liefern die durchgeführten Experimente erste Erkenntnisse, wie sich Performancegewinne erzielen lassen.

Im Rahmen dieser Bachelorarbeit wurde festgestellt, dass sich durch Scheduling der Bulk-Transfers signifikante Performancesteigerungen in hybriden Netzwerken ergeben. Durch die richtige Kombination aus Scheduling-Algorithmus und hybrider Topologie kann eine bis zu 23% Reduzierung der *Average Transfer Completion Time* und eine 5,5% Verbesserung der *Longest Completion Time* erzielt werden. Der *Average Throughput* konnte durch keinen Scheduling-Algorithmus verbessert werden.

Darüber hinaus konnte gezeigt werden, dass statische Netzwerke dem hybriden Pendant deutlich unterlegen sind und sich durch die kombinierte Einführung aus hybriden Topologien und Scheduling-Algorithmen eine bis zu 28% niedrigere Auslastung des Netzwerks hinsichtlich der Bandbreite, eine bis zu 40% kürzere durchschnittliche Transfer-Bearbeitungszeit und eine bis zu 25% kürzere Maximallaufzeit der Bulk-Transfers im Netzwerk feststellen lässt.

3. Grundlagen

Um die in Kapitel 4 durchgeführten Experimente nachvollziehen zu können, werden im Laufe dieses Kapitels einige Grundlagen erläutert. Außerdem werden Notationen und Bewertungskriterien für die experimentellen Untersuchungen definiert.

3.1. Bulk-Transfers

Ein Bulk-Transfer ist die Übertragung einer vordefinierten Datenmenge von einem Startknoten im Netzwerk zu einem Zielknoten. Die Übertragung ist beendet, sobald die gesamte Datenmenge am Zielknoten angekommen ist [17]. Bulk-Transfers werden häufig auch als *Elephant Flows* bezeichnet [13, 25], da sie im Unterschied zu anderen Netzwerktransfers besonders groß sind und die Datenmenge je Transfer mehrere Petabytes annehmen kann. Damit spielen Bulk-Transfers beispielsweise bei Content Delivery Networks, Backups von Finanzinstituten oder der Synchronisation von Suchindizes bei Suchmaschinen über verschiedene Datenzentren hinweg eine große Rolle [17, 21].

Im Rahmen dieser Arbeit werden nicht einzelne Bulk-Transfers betrachtet, sondern das Zusammenspiel verschiedener Transfers innerhalb eines Netzwerks (Definition 1). Im Folgenden wird zunächst die verwendete Notation zum Umgang mit Bulk-Transfers eingeführt. Die Menge $B_{v,u}$ umfasst jene Transfers, die von Knoten $v \in V$ nach Knoten $u \in V$ gesendet werden sollen. Die Menge der Bulk-Transfers, die von einem Knoten v ausgehen, ist demnach $B_v = \bigcup_{u \in V} B_{v,u}$ und die Gesamtmenge der betrachteten Bulk-Transfers innerhalb des Netzwerks ist $B = \bigcup_{v \in V} B_v$. Wird ein einzelner Bulk-Transfer betrachtet, der an Knoten v startet und zu Knoten u gesendet werden soll, so schreiben wir auch $b_{v,u}$. Sind der Start- und Zielknoten irrelevant, so schreiben wir einfach b. Die Größe eines Transfers ist definiert durch $|b|$. Um die Güte der Gesamtheit der Transfers innerhalb eines Netzes zu bewerten, werden im nächsten Abschnitt einige Metriken eingeführt.

Metrik	Beschreibung
Fairness	Die Ressourcen sollten den Transfers fair zugeteilt werden [24].
Latenz	Die Zeit zwischen Start und Ankunft eines Pakets im Netzwerk.
Bandbreite	Die maximale Rate an Daten, die übertragen werden können.
Transfer Completion Time	Die Zeit, die ein Transfer benötigt um übermittelt zu werden.
Longest Completion Time	Die Zeit, bis der langsamste Transfer aus einer Menge an Bulk-Transfers übermittelt wurde.
Link Utilization	Die Auslastung der Kanten in einem Netzwerk. [14]
Erfüllte Deadlines	Das Verhätnis der erfüllten Deadlines zur Gesamtzahl an Deadlines [23].
Reliability	Z.B. die Rate an verlorenen Paketen im Netzwerk [2].

Abbildung 3.1.: Beschreibung klassischer Bewertungsmetriken für das Scheduling von Bulk-Transfers.

Definition 1 (Statisches Netzwerk [11]) *Ein gewichtetes statisches Netzwerk N wird formal definiert als Tripel $N = (V, E, w)$, wobei $V = \{v_1, \ldots, v_n\}$ die Knotenmenge und $E \subseteq V \times V$ die statische ungerichtete Kantenmenge ist. Die Funktion $w : E \longrightarrow \mathbb{N}$ weist jeder Kante eine nicht-negative Gewichtung zu. Für eine Kante $(i, j) \in E$ schreiben wir auch $e_{i,j}$.*

3.1.1. Bewertungsmetriken

Bulk-Transfers können hinsichtlich verschiedener Kriterien begutachtet werden. Einige Kriterien lassen sich aus dem Vorhandensein von Fristen *(Deadlines)* ableiten. Solche Fristen definieren für spezifische Transfers einen Zeitpunkt, zu dem die Daten am Zielknoten vorhanden sein müssen. Dabei kann zwischen weichen und harten Fristen unterschieden werden. Harte Fristen gilt es stets einzuhalten, während weiche Fristen zugunsten anderer Transfers mit harten Fristen bis zu einem gewissen Grad überschritten werden können. Andere Kriterien hingegen zielen mehr auf die Bearbeitungszeit der Transfers, d.h. die Zeit von der Ankunft am Startknoten bis zur Ankunft am Zielknoten, als entscheidenden Faktor ab [24].

Die Tabelle (Abbilung 3.1) gibt eine Übersicht über klassische Bewertungsmetriken. Im Rahmen dieser Arbeit bleiben jedoch jegliche Arten von Fristen unberücksichtigt und es wird sich im Wesentlichen auf die Bandbreite und die Bearbeitungs- bzw. Übertragungszeit der Transfers konzentriert. Konkret wird die durchschnittlich genutzte Bandbreite, die durchschnittliche Transferbearbeitungszeit sowie die maximale Transferbearbeitungszeit betrachtet. Auch in vergleichbaren wissenschaftlichen Arbeiten werden diese Metriken zur Bewertung der Ergebnisse verwendet. So werden in [13, 25] die Bandbreite und in [21, 25] die Transferbearbeitungszeit als zentrale Bewertungskriterien genutzt. Diese Metriken werden nun formal

eingeführt.

Um die durchschnittliche sowie die maximale Transferbearbeitungszeit bestimmen zu können, wird zunächst die Transferbearbeitungszeit für einen einzelnen Transfer definiert.

Definition 2 (Transfer Completion Time [3, 24]) *Die Bearbeitungszeit (Transfer Completion Time)* $t^b_{completion}$ *eines Bulk-Transfers* b *ist definiert als die Zeitspanne zwischen dem Zeitpunkt, an dem die Transferdetails am Startknoten bekannt sind bis zu dem Zeitpunkt, an dem die Daten vollständig am Zielknoten angekommen sind.*

Diese Definition lässt sich einfach auf eine Menge an Bulk-Transfers erweitern, über die ein Durchschnitt gebildet werden kann.

Definition 3 (Average Transfer Completion Time) *Bei Betrachtung einer Menge* B *an Bulk-Transfers innerhalb eines Netzwerks ist die durchschnittliche Transferbearbeitungszeit* $t^B_{avg\_completion}$ *die Summe der Transferbearbeitungszeiten über alle betrachteten Transfers hinweg geteilt durch die Menge an Transfers:* $t^B_{avg\_completion} = \frac{\sum_{b \in B} t^b_{completion}}{|B|}$.

Die maximale Transferbearbeitungszeit hingegen ist schlicht das Maximum der Transferbearbeitungszeiten aller betrachteten Transfers.

Definition 4 (Longest Completion Time) *Bei Betrachtung einer Menge* B *an Bulk-Transfers innerhalb eines Netzwerks ist ist die maximale Transferbearbeitungszeit wie folgt definiert:*
$t^B_{max\_completion} = \max\{t^b_{completion} \mid b \in B\}$

Nachdem nun die verschiedenen Transferbearbeitungszeiten definiert wurden, kann die Bandbreite über die insgesamt versendete Datenmenge sowie die Summe der Transferbearbeitungszeiten definiert werden.

Definition 5 (Average Throughput [31]) *Die durchschnittlich verwendete Bandbreite (Average Throughput) ist die Summe der erfolgreich versendeten Daten geteilt durch die Summe der einzelnen Bearbeitungszeiten aller Transfers, gemessen in* $Mbit/s$.

Das im Rahmen dieser Arbeit verfolgte Ziel ist es, den *Average Throughput* zu maximieren und die *Average Completion Time* sowie die *Longest Completion Time* zu minimieren. Im weiteren Verlauf dieses Kapitels werden dazu verschiedene Techniken (Rekonfigurierbare Netzwerke und Sechduling-Algorithmen) betrachtet, deren Sinnhaftigkeit in den folgenden Abschnitt zunächst theoretisch diskutiert und anschließend in Kapitel 4 experimentell ermittelt wird.

3.2. Routing

Bevor erläutert wird, wie die Bulk-Transfers hinsichtlich der Bewertungsmetriken aus Kapitel 3.1.1 mittels verschiedener Techniken optimiert werden können, werden im Folgenden zunächst die Routing-Grundlagen erläutert, die in Kapitel 4 verwendet werden. Damit soll ein Grundverständnis der Paketvermittlung in Netzwerken geschaffen werden, auf dessen Basis auch die Idee und Vorteile von rekonfigurierbaren Netzwerken (Abschnitt 3.3) deutlicher werden.

Daten werden in Form von kleinen Paketen durch das Netzwerk hin zu ihrem Ziel geleitet. Das Routing-Problem beschäftigt sich mit der Frage, auf welchem Weg diese Datenpakete von ihrem Ausgangsknoten zu ihrem Zielknoten gelangen. Die Komplexität dieses Problems ist durch die meist große Anzahl an möglichen Routen zwischen Start- und Endpunkt gegeben. Ziel ist es, einen möglichst optimalen Weg für die Datenpakete zu finden [12]. Betrachten wir das ISO/OSI-Schichtenmodell (siehe Abbildung 3.2), so liegen jene Protokolle, die sich mit dem Routing beschäftigen auf Schicht 3, der Netzwerkschicht [31]. Klassische Protokolle, die dieses Problem behandeln sind das *Open Shortest Path First* Protokoll (OSPF) [27] oder das *Border Gateway Protocol* (BGP) [36]. BGP ist ein *Exterior Gateway Protocol*, d.h. es befasst sich mit dem Routing zwischen abgeschlossenen, autonomen Systemen, während OSPF als *Interior Gateway Protocol* eine Lösung für Routing innerhalb solcher Systeme anbietet. Im Folgenden wird OSPF genauer eingeführt, da dieses Protokoll in den weiteren Betrachtungen verwendet wird.

3.2.1. OSPF

OSPF arbeitet auf einem Netzwerk N mit gewichteten, nicht-negativen Kanten, wie es in Definition 1 eingeführt wurde. Dabei wird der optimale Pfad hin zu einem Zielknoten als jener betrachtet, bei dem die Summe der Kantengewichte minimal ist. Jeder Zwischenknoten auf dem Pfad betrachtet dabei nicht den Startknoten, sondern lediglich den Zielknoten und schickt die Pakete auf dem kürzesten Pfad von sich zum Zielknoten weiter. Gibt es mehrere gleichwertige Pfade, so werden die Paketete gleichmäßig auf diese Pfade aufgeteilt. Dieses Vorgehen nennt man *Equal-Cost Multipath Routing* (ECMP) [27, 33].

Im Allgemeinen basiert OSPF auf dem Dijkstra-Algorithmus, der dazu dient, den optimalen Pfad zu berechnen und im Folgenden kurz skizziert werden soll. Der Dijkstra-Algorithmus wird auf Basis der Erläuterungen und des Pseudo-Codes in [8] und [31] eingeführt. Es handelt sich um einen gierigen *(greedy)* Algorithmus, was bedeutet, dass er in jedem Ausführungsschritt die nächste bestmögliche Lösung wählt, um zum Ziel zu kommen. Die Idee ist, dass immer der kürzest mögliche Pfad vom Startknoten aus betrachtet wird. Dabei wird die Länge eines Pfades nicht über die Anzahl der Kanten definiert, sondern über die Summe der Gewichtungen der Kanten. Übersteigt eine Pfadlänge die eines möglichen kürzeren Pfades, wird zuerst dem kürzeren Pfad nachgegangen. Erreicht man in der Art irgendwann den Zielknoten, so ist

der betrachtete Pfad der kürzeste hin zu diesem Knoten. Formal beschrieben wird diese Idee in Algorithmus 1. Mittels der Funktion p, die dieser Algorithmus als Output liefert, kann der Pfad vom Startknoten s hin zum Zielknoten z rekonstruiert werden. Es gilt $p^l(z) = v$, wobei l die Anzahl der Kanten auf dem Pfad von s nach z ist.

Algorithm 1 Dijkstra

Input: Ein Netzwerk $N = (V, E, w)$, Startknoten $s \in V$ und Zielknoten $z \in V$.

1. Initialisiere die Abstandsfunktion $d : V \to \mathbb{N}$ mit
   (a) $d(s) = 0$ und
   (b) $d(v) = \infty$ für alle $v \in V \setminus \{s\}$.
2. Initialisiere die Vorgängerfunkntion $p : V \to V \cup \{\text{null}\}$ mit $p(v) = \text{null}$ für alle $v \in V$.
2. Initialisiere die Menge besuchter Knoten G mit $G = V$.
3. Solange $G \neq \emptyset$:
   (a) Wähle $v \in G$ mit minimalem Wert $d(v)$.
   (b) Entferne v aus G.
   (c) Wenn $v = z$, dann beende.
   (d) Berechne $\text{tmp}_u = d(v) + w((v, u))$ für jeden Knoten $u \in G$ mit $(v, u) \in E$.
   (e) Wenn $\text{tmp}_u < d(u)$, dann setze $d(u) = \text{tmp}_u$ und $p(u) = v$.

Output: Funktion p, über die der kürzeste Weg von s zu z hergeleitet werden kann.

3.3. Rekonfigurierbare Netzwerke

Traditionelle Rechenzentrumsnetzwerke basieren auf paketbasierten Switchen in ihren Netzwerken [16]. Moderne Anwendungen, wie Cloud, Big Data oder Streaming, haben dafür gesorgt, dass die Anforderungen hinsichtlich der verfügbaren Bandbreite in Rechenzentren und der Skalierbarkeit von Netzwerken stark gewachsen sind, wodurch die Kosten für den Aufbau und die Administration solcher traditionellen Rechenzentren sehr groß geworden sind [16, 42]. Eine Alternative, die die Kosten für den Aufbau neuer und die Erweiterung bestehender Netzwerke senkt, sind rekonfigurierbare Netzwerke.

Rekonfigurierbare Netzwerke *(Software Defined Networks)* bestehen aus Netzwerkequipment mit der Fähigkeit, die Kontroll- *(control plane)* und die Datenschicht *(data plane)* voneinander separat zu behandeln, sowie die Möglichkeit, die Kontrollschicht dynamisch zu programmieren [42]. Im Gegensatz zu traditionellen, paketbasierten Switchen, die auf Schicht 2 oder 3 des ISO/OSI-Modells (Abbildung 3.2) operieren, bietet die Programmierbarkeit der Kontrollschicht von SDN-Switchen die Möglichkeit, beispielsweise die Weiterleitung der Pakete auf der physikalischen Schicht (Layer 1) des ISO/OSI-Modells zu realisieren,

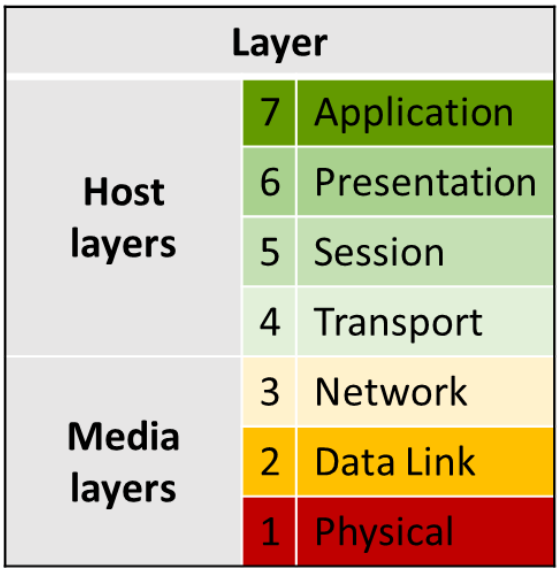


Abbildung 3.2.: In Anlehnung an [41]: ISO/OSI-Schichten-Modell

wodurch ein Performancegewinn erzielt wird. Ein weiterer großer Vorteil rekonfigurierbarer Netzwerke gegenüber traditionellen Netzwerken ist die verbesserte Kontrolle über das gesamte Netzwerk durch die Trennung von Kontroll- und Datenschicht [42] und dadurch über die Verteilung des Traffics innerhalb des Netzwerks, um zum Beispiel die Pfade zwischen zwei Servern zu minimieren [16].

Grundsätzlich kann zwischen traditionellen *statischen* Netzwerken und neueren *rekonfigurierbaren* Netzwerken unterschieden werden. Häufig existieren aber hybride Netzwerke (Abbildung 3.3), die sowohl statische Verbindungen zwischen ihren Servern (bzw. *top-of-the-rack Switchen*) besitzen, als auch dynamische Verbindungen [11]. Im experimentellen Teil dieser Bachelorarbeit werden ausschließlich hybride Netzwerke betrachtet, da Algorithmen zum Design solcher Netzwerke und für das Routing des Traffics auf diesen nicht hinreichend untersucht wurden [11] und außerdem kaum rein rekonfigurierbare Netzwerke existieren [7]. Zunächst wird ein hybrides Netzwerk als Erweiterung eines statischen Netzwerks aus Definition 1 formal eingeführt.

Definition 6 (Hybride Netzwerke [11]) *Ein gewichtetes hybrides Netzwerk* $N = (V, E, M, w)$ *ist ein statisches Netzwerk gemäß Definition 1, das um eine Menge* $M \subseteq V \times V$ *an bidirektionalen rekonfigurierbaren Kanten erweitert wird. Für eine rekonfigurierbare Kante* $(i, j) \in M$ *schreiben wir auch* $m_{i,j}$.

Zur Vereinfachung betrachten wir in einem solchen Netzwerk keine Kanten zu Switchen oder Routern, sondern ausschließlich resultierende Pfade von Knoten zu Knoten und nehmen an, dass die Gewichtung einer jeden Kante zunächst gleich ist.

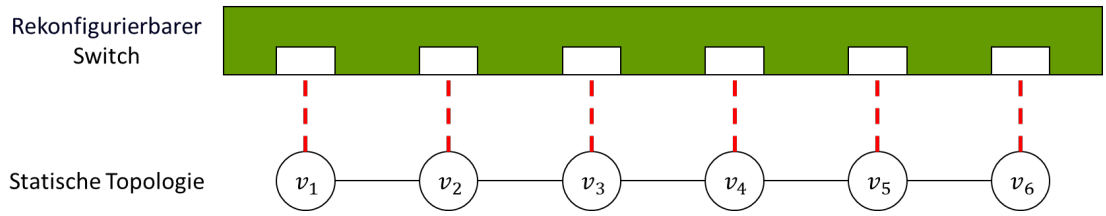


Abbildung 3.3.: In Anlehnung an [11]: In dieser Abbildung wird ein kleines hybrides Netzwerk vorgestellt, dass sowohl statische Kanten (z.B.: zwischen v_1 und v_2) als auch rekonfigurierbare Kanten über den rekonfigurierbaren Switch besitzt. Der Switch kann nun konfiguriert werden um zwei beliebige Knoten mit einer Kante zu verbinden.

Realisiert werden können rekonfigurierbare Netzwerke durch verschiedene Technologien. Unter anderem sind dies gezielte drahtlose Verbindungen *(beamformed wireless connections)*, optische Freiraum-Verbindungen *(free-space optical interconnects)* oder *Opctical Circuit Switches* [11].

3.3.1. Optical Circuit Switching

Eine spezielle Form der SDN stellen Netzwerke auf Basis von Optical Circuit Switches *(OCS)* dar. Da traditionelle Rechenzentren mit paketbasierten, *elektronischen* Switchen bei wachsenden Datenraten kostspieliger und energiehungriger wurden, sind rekonfigurierbare Lösungen auf optischer Basis entstanden [7].

Bei OCS werden Datenströme über dedizierte Lichtpfade durch das Netzwerk geleitet. Ein OCS besitzt wie ein herkömmlicher Switch mehrere Ports für Verbindungen zu verschiedenen Knoten, wobei bei OCS ausschließlich optische Medien Verwendung finden. Innerhalb des optischen Switches befinden sich Spiegel, die typischerweise elektro-magnetisch (*MEMS*) in Millisekunden verstellt werden können und so zwei mit dem Switch verbundene Knoten verbinden können (Abbildung 3.4). Dazu muss nicht wie bei herkömmlichen Switchen das Paket die elektronischen Schaltkreise des Geräts und somit eine optisch-elektrisch-optische Umwandlung durchlaufen, sondern bleibt in optischer Form, wodurch Zeit und Energie gespart wird [5, 25, 34, 35].

Da in [7] festgestellt wurde, dass eine einmalige Einstellung eines OCS-Netzwerks eine hinreichende Optimierung des Netzwerks ermöglicht, betrachten wir im Rahmen der experimentellen Untersuchungen ausschließlich ein einmalig eingestelltes Netzwerk. Eine dynamische Rekonfigurierung wäre außerdem außerhalb des Rahmens einer Bachelorarbeit.

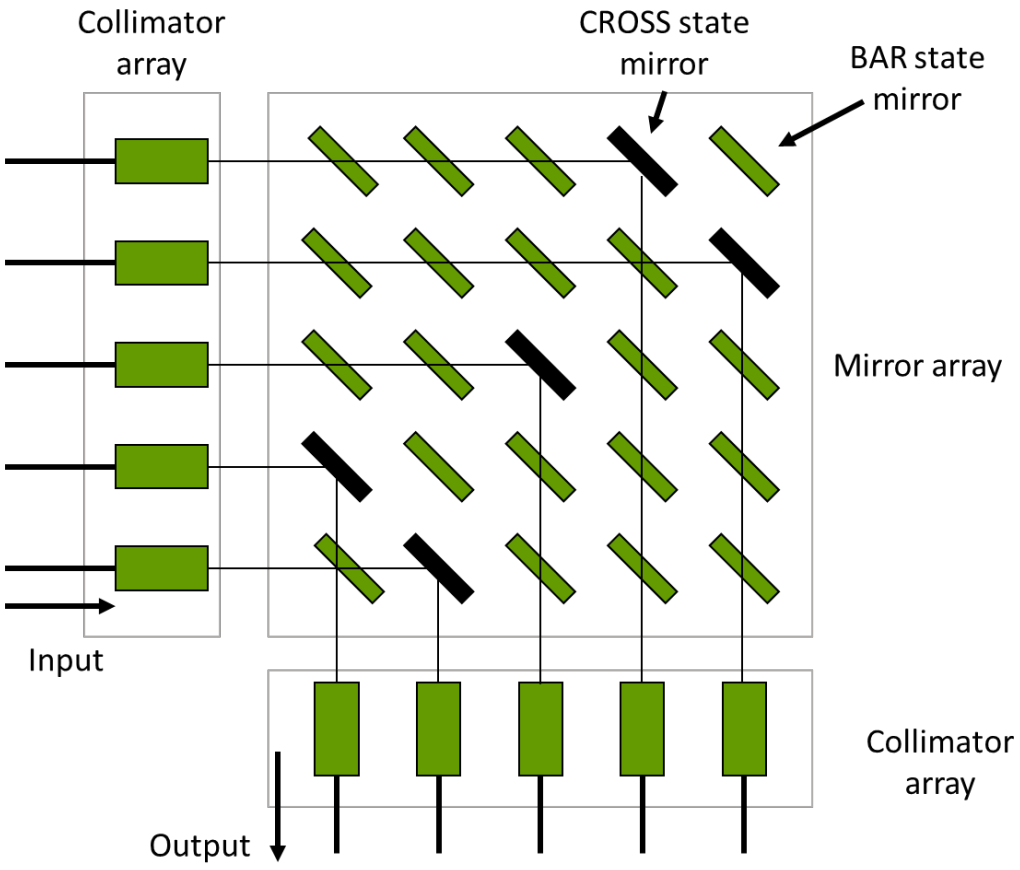


Abbildung 3.4.: In Anlehnung an [10]: Struktur eines 2D MEMS (*Micro-Electro-Mechanical Systems*) optischen Switchs. Jeder Spiegel kann sich in einem *CROSS* oder einem *BAR* Zustand befinden. Während im Zustand *BAR* das Licht ungehindert vorbei geleitet wird, kann das Licht im Zustand *CROSS* zu den jeweiligen Ausgängen gespiegelt werden.

3.3.2. Algorithmen

Für hybride Netzwerke bestehend aus statischen Kanten E und dynamischen Kanten M betrachten wir verschiedene existierende Algorithmen, die die Einstellung der rekonfigurierbaren Kanten ermitteln. Dabei werden so lange dynamische Kanten $m_{i,j}$ zwischen Knoten i und Knoten j dem hybriden Netzwerk hinzugefügt, bis eine Abbruchbedingung erfüllt ist. Eine solche Abbruchbedingung besteht beispielsweise darin, dass dem Netzwerk N keine weiteren Kanten hinzugefügt werden können, da bereits alle möglichen dynamischen Kanten m hinzugefügt wurden. Ziel ist es, mit Hilfe dieser Algorithmen eine ideale Auslastung hinsichtlich der bereits in Kapitel 3.1.1 vorgestellten Metriken zu erzielen. Dazu versuchen alle hier vorgestellten Algorithmen neue kürzeste Wege durch die rekonfigurierbaren Kanten zu schaffen, damit die Transfers möglichst optimal zu ihrem Ziel geroutet werden können.

Es wird ein hybrides Netzwerk N nach Definition 6 betrachtet und eine Menge B an Bulk-Transfers, die innerhalb dieses Netzwerks verschickt werden sollen. Es sei an der Stelle nochmal darauf hingeweisen, dass eine rekonfigurierbare Kante $m_{i,j}$ bidirektional von Knoten i zu Knoten j verläuft. Es gilt also, dass wenn $m_{i,j}$ existiert, auch $m_{j,i}$ existiert [11].

In den in dieser Arbeit durchgeführten Experimenten besitzt jeder Knoten drei statische Kanten e und maximal eine rekonfigurierbare Kante m. Es können nicht zwei Kanten dieselben Knoten verbinden, egal ob statisch oder rekonfigurierbar. Außerdem betrachten wir alle gegebenen Kanten mit einer einheitlichen Gewichtung (Kapitel 3.3), wodurch keine Kante über eine andere bevorzugt wird. Lediglich die Länge der Pfade (entspricht im Fall gleicher Gewichtung der Anzahl an Kanten eines Pfades) und die Größe der Transfers werden für die Konfiguration der dynamischen Kanten berücksichtigt.

Die einzelnen Algorithmen werden anhand eines Beispiels veranschaulicht. Die Topologie aus Abbildung 3.5 wird als Basis verwendet. Innerhalb dieses Netzes sollen vier Transfers versendet werden mit $|b_{2,8}| = 10GB$, $|b_{1,7}| = 8GB$, $|b_{1,6}| = 5GB$ und $|b_{3,5}| = 2GB$. Die durch Anwendung der Algorithmen resultierenden Topologien sind in den Abbildungen 3.6, 3.7 und 3.8 dargestellt. Auch in diesem Beispiel wird davon ausgegangen, dass jeder Knoten maximal eine rekonfigurierbare Kante besitzen kann.

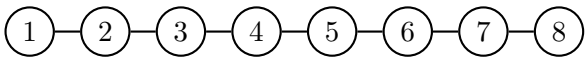


Abbildung 3.5.: Statische Topologie.

DemandFirst

Zuerst betrachten wir den Algorithmus *DemandFirst* [11], welcher in Algorithmus 2 detailliert beschrieben wird. Dieser *greedy* Algorithmus sucht die größten Transfers und fügt für diese die passenden Kanten

hinzu, damit diese ideal durch das Netzwerk N geleitet werden. Besitzen Start- oder Zielknoten eines Transfers bereits eine Kante m oder eine gemeinsame Kante e, so wird für diesen Transfer keine neue Kante hinzugefügt. Beendet wird der Algorithmus, wenn alle Transfers bearbeitet wurden, oder wenn keine Kanten dem Netzwerk mehr hinzugefügt werden können. Um diesen Algorithmus anwenden zu können, müssen alle Transfers und deren Größe im Vorfeld bekannt sein.

Algorithm 2 DemandFirst

Input: Ein hybrides Netzwerk N mit Transfermenge B.

1. Sortiere die Transfers B absteigend nach ihrer Größe speichere sie in Liste D.
2. Bis die Liste leer ist oder keine weiteren Kanten zu N hinzugefügt werden können:
   (a) Wähle den größten Transfer $b_{i,j}$ in D.
   (b) Füge, falls möglich, $m_{i,j}$ dem Netzwerk N hinzu.
   (c) Entferne den ersten Transfer aus D.

Output: Ein neues Netzwerk N mit konfigurierten dynamischen Kanten M.

Nach Anwendung des Algorithmus 2 auf die Topologie aus Abbildung 3.5 ergibt sich die Topologie aus Abbildung 3.6. Die Reihenfolge, in der die rekonfigurierbaren Kanten hinzugefügt wurden, lässt sich der Kantenbeschriftung entnehmen. Für den Transfer $b_{1,6}$ konnte keine Kante hinzugefügt werden, da bereits eine für einen größeren Transfer hinzugefügt wurde, der an Knoten 1 startet.

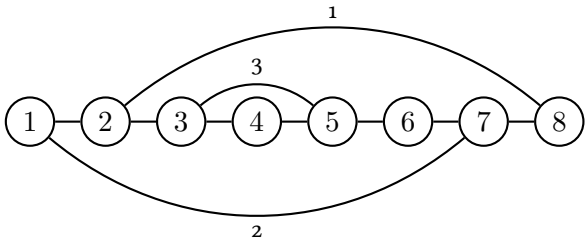


Abbildung 3.6.: Topologie nach Anwendung von *DemandFirst*.

LongestPathFirst

Der Algorithmus *LongestPathFirst* (Algorithmus 3) ist sehr ähnlich zu *GreedyLinks* aus [11]. *LongestPathFirst* arbeitet jedoch unabhängig von den Transfers und betrachtet stattdessen alle kürzesten Pfade zwischen Knoten und minimiert die maximale kürzeste Pfadlänge $|P|$ zwischen zwei Knoten im gesamten Netzwerk N. Dazu sucht er den längsten kürzesten Pfad P und verbindet die Knoten mit einer rekonfigurierbaren Kante m. Wenn eine neue Kante dem Netzwerk hinzugefügt wurde, dann wird die Pfadlänge zwischen den Knoten erneut berechnet, da sich nun neue kürzeste Pfade gebildet haben. Können keine

weiteren rekonfigurierbaren Kanten dem Netzwerk hinzugefügt werden, dann endet der Algorithmus.

Algorithm 3 LongestPathFirst

Input: Ein hybrides Netzwerk N.

1. Initialisiere eine leere Liste *paths*.
2. Bis keine weiteren Kanten zu N hinzugefügt werden können:
   (a) Für jede mögliche rekonfigurierbare Kante $m_{i,j}$ in N:
      i. Berechne die Länge des Pfades $P_{i,j}$ von i zu j.
      ii. Füge $P_{i,j}$ zu *paths* hinzu.
   (b) Finde den längsten Pfad $P_{i,j}$ in *paths*.
   (c) Füge für den längsten Pfad $P_{i,j}$ die rekonfigurierbare Kante $m_{i,j}$ dem Netzwerk N hinzu.
   (d) Leere die Liste *paths*.

Output: Ein neues Netzwerk N mit konfigurierten dynamischen Kanten M.

Nach Anwendung des Algorithmus 3 auf die Topologie aus Abbildung 3.5 ergibt sich die Topologie aus Abbildung 3.7. Die Reihenfolge, in der die rekonfigurierbaren Kanten hinzugefügt wurden, lässt sich der Kantenbeschriftung entnehmen.

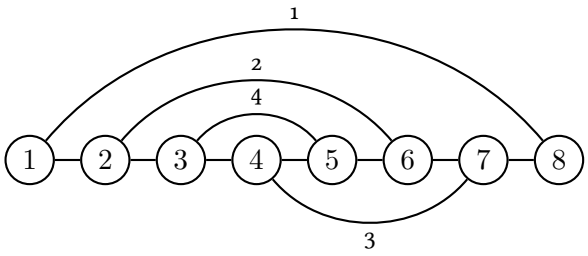


Abbildung 3.7.: Topologie nach Anwendung von *LongestPathFirst*.

DemandAwareLongestPathFirst

DemandAwareLongestPathFirst (Algorithmus 4) ist ein um die Kenntnis der Transfers erweiterter *LongestPathFirst* Algorithmus und ähnlich zu *GainUpdate* aus [11]. *DemandAwareLongestPathFirst* funktioniert analog zu *LongestPathFirst*, allerdings werden nur die längsten kürzesten Pfade $P_{i,j}$ betrachtet, für die auch ein Transfer $b_{i,j}$ existiert.

Nach Anwendung des Algorithmus 4 auf die Topologie aus Abbildung 3.5 ergibt sich die Topologie aus Abbildung 3.8. Die Reihenfolge, in der die rekonfigurierbaren Kanten hinzugefügt wurden, lässt sich der Kantenbeschriftung entnehmen. Auffällig ist hier, dass keine rekonfigurierbare Kante $(1, 7)$ hinzugefügt

Algorithm 4 DemandAwareLongestPathFirst

Input: Ein hybrides Netzwerk N mit Transfermenge B.

1. Initialisiere eine leere Liste *paths*.
2. Bis keine weiteren Kanten zu N hinzugefügt werden können:
   (a) Für jede mögliche rekonfigurierbare Kante $m_{i,j}$ in N, für die ein Transfer $b_{i,j} \in B$ existiert:
      i. Berechne die Länge des Pfades $P_{i,j}$ von i zu j.
      ii. Füge $P_{i,j}$ zu *paths* hinzu.
   (b) Finde den längsten Pfad $P_{i,j}$ in *paths*.
   (c) Füge für den längsten Pfad $P_{i,j}$ die rekonfigurierbare Kante $m_{i,j}$ dem Netzwerk N hinzu.
   (d) Leere die Liste *paths*.

Output: Ein neues Netzwerk N mit konfigurierten dynamischen Kanten M.

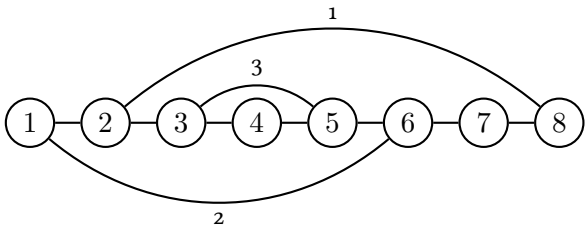


Abbildung 3.8.: Topologie nach Anwendung von *DemandAwareLongestPathFirst*.

wurde. Dies lässt sich damit begründen, dass zunächst die Kante $(2, 8)$ hinzugefügt wurde, wodurch der Pfad von Knoten 1 zu Knoten 7 kürzer ist, als der von Knoten 1 zu Knoten 6.

3.4. Scheduling-Algorithmen

Im Allgemeinen liefern Scheduling-Algorithmen ein Verfahren, um zu entscheiden, wann konsumierbare Ressourcen wie Speicher, Bandbreite oder Prozessorzeit für einzelne Komponenten freigegeben werden. So bestimmen sie im Kontext von Betriebssystem beispielsweise, welcher Prozess zu welchem Zeitpunkt von der CPU bearbeitet werden soll [3] und im Kontext von Netzwerken entscheiden sie, wann welches Paket transferiert oder wann welcher Transfer gestartet wird [24]. Je nach Anwendungsgebiet werden dabei verschiedene Ziele verfolgt und die Algorithmen können mit verschiedenen Metriken verglichen werden. So ist im Kontext von Betriebssystemen beispielsweise von Bedeutung, dass die einzelnen Prozesse schnell abgearbeitet werden (Minimierung der Bearbeitungszeit) und dass der Nutzer nicht zu lange auf (Zwischen-)Ergebnisse einzelner interaktiver Aufgaben warten muss (Minimierung der Antwortzeit) [3]. Im Kontext von Netzwerken spielen Scheduling-Algorithmen eine Rolle, um Transfers hinsichtlich der

in Kapitel 3.1.1 beschriebenen Bewertungsmetriken zu optimieren.

Im Rahmen dieser Arbeit wird ein Netzwerk N nach Definition 1 und ein Knoten $v \in V$, von dem aus eine endliche Menge $B_v = \{b_1, b_2, ..., b_n\}$ an Bulk-Transfers zu anderen Knoten innerhalb des Netzes durchgeführt werden soll, betrachtet. Für jeden Transfer ist die exakte zu sendende Datenmenge bekannt und bestimmt durch $|b_i|$. Dabei wird vereinfachend davon ausgegangen, dass alle Daten zum Startzeitpunkt am Knoten v vorliegen. Der Zielknoten der einzelnen Bulk-Transfers sowie Routing-Details werden im Folgenden außen vor gelassen (Routing wird in Kapitel 3.2 genauer betrachtet), stattdessen dienen die Scheduling-Algorithmen in diesem Szenario der Entscheidung, wann welcher Transfer von v aus gestartet werden soll. Folgend werden die drei gängigen Scheduling-Algorithmen *First Come First Serve*, *Shortest Job First* und *Longest Job First* als Varianten für Netzwerk-Scheduling auf Basis der Beschreibungen in [3] eingeführt.

$$B_v = \{\ 1\text{: } \boxed{1\text{ GB}}\ ,\ 2\text{: } \boxed{10\text{ GB}}\ ,\ 3\text{: } \boxed{8\text{ GB}}\ ,\ 4\text{: } \boxed{12\text{ GB}}\ \}$$

Abbildung 3.9.: Beispiel für eine Menge an Bulk-Transfers, die am Startknoten v vorliegen. Die Werte in den Boxen beschreiben die Größe der je Transfer zu versendenden Datenmenge.

3.4.1. First Come First Serve

First Come First Serve ist der naheliegendste Scheduling-Algorithmus. Es wird jener Transfer zuerst gestartet, der als erstes beim Startknoten vorliegt [3]. Damit ist *First Come First Serve* der Standardalgorithmus aller Netzwerke, in denen der Traffic nicht aktiv durch einen komplexeren Scheduling-Algorithmus verwaltet werden soll. Da in unserem Fall vereinfachend davon ausgegangen wurde, dass alle Transfers zum gleichen Zeitpunkt vorliegen, werden die Transfers nach der Reihenfolge ihrer Indizes gestartet, d. h. b_1 wird zuerst transferiert, dann b_2, bis schlussendlich b_n gesendet wird. Dieses Vorgehen wird in Algorithmus 5 beschrieben und kann in Abbildung 3.9 nachvollzogen werden.

Algorithm 5 First Come First Serve

Input: Menge $B_v = \{b_1, \ldots, b_n\}$ der Bulk-Transfers am Knoten v.

1. Erstelle eine endliche Folge i^{fcfs} mit $i^{\text{fcfs}} = (1, \ldots, n)$.

Output: i^{fcfs} bestimmt die Reihenfolge der Indizes, nach denen die Transfers aus B_v gestartet werden.

3.4.2. Shortest Job First

Bei dem *Shortest Job First* Algorithmus werden die Transfers nicht lediglich nach der Reihenfolge ihres Auftretens versendet, sondern nach der Größe der zu versendenden Datenmenge. Der Transfer mit der

kleinsten Datenmenge wird zuerst gestartet, der mit der größten als letztes. Sollten mehrere Transfers die gleiche Größe haben, so kommt bei der Sortierung dieser der *First Come First Serve* Algorithmus zum Einsatz [3]. Es wird dazu eine endliche Folge $i^{\text{sjf}} = (i_1, \ldots, i_n)$ nach Algorithmus 6 definiert, die die Reihenfolge festlegt, in der die Transfers aus B_v nach dem *Shortest Job First* Algorithmus gestartet werden.

Algorithm 6 Shortest Job First

Input: Menge $B_v = \{b_1, \ldots, b_n\}$ der Bulk-Transfers am Knoten v.

1. Erstelle eine endliche Folge $i^{\text{sjf}} = (i_1, \ldots, i_n)$, für die folgende Eigenschaften gelten:
   (a) Jeder Wert aus $\{1, \ldots, n\}$ kommt exakt einmal in der Folge vor.
   (b) Für alle $j, k \in \{1, \ldots, n\}$ mit $j < k$ gilt $|b_{i_j}| \leq |b_{i_k}|$.
   (c) Für alle $j, k \in \{1, \ldots, n\}$ mit $|b_{i_j}| = |b_{i_k}|$ und $i_j < i_k$ gilt $j < k$.

Output: i^{sjf} bestimmt die Reihenfolge der Indizes, nach denen die Transfers aus B_v gestartet werden.

$$i_v^{\text{sjf}} = (1, 3, 2, 4)$$

Abbildung 3.10.: Beispiel für die sich aus Abbildung 3.9 ergebende Reihenfolge, in der die Transfers nach dem *Shortest Job First* Algorithmus an Knoten v gestartet werden.

3.4.3. Longest Job First

Longest Job First ist sehr ähnlich zu *Shortest Job First*, jedoch werden hierbei nicht die Transfers mit der kleinsten Datenmenge zuerst gestartet, sondern die mit der größten. Die Indexfolge i^{ljf}, die sich aus der Anwendung des Algorithmus 7 ergibt, definiert damit die Reihenfolge, in der die Transfers aus B_v nach dem *Longest Job First* Algorithmus gestartet werden.

Algorithm 7 Longest Job First

Input: Menge $B_v = \{b_1, \ldots, b_n\}$ der Bulk-Transfers am Knoten v.

1. Erstelle eine endliche Folge $i^{\text{ljf}} = (i_1, \ldots, i_n)$, für die folgende Eigenschaften gelten:
   (a) Jeder Wert aus $\{1, \ldots, n\}$ kommt exakt einmal in der Folge vor.
   (b) Für alle $j, k \in \{1, \ldots, n\}$ mit $j < k$ gilt $|b_{i_j}| \geq |b_{i_k}|$.
   (c) Für alle $j, k \in \{1, \ldots, n\}$ mit $|b_{i_j}| = |b_{i_k}|$ und $i_j < i_k$ gilt $j < k$.

Output: i^{ljf} bestimmt die Reihenfolge der Indizes, nach denen die Transfers aus B_v gestartet werden.

3.4.4. Weitere Scheduling-Algorithmen

Alle bisher betrachteten Algorithmen verhalten sich kooperativ, d.h. sie betrachten jeden Transfer als Einheit, der nicht unterbrochen werden darf. Darüber hinaus gibt es unterbrechende Algorithmen, die es

$$i_v^{\text{ljf}} = (4, 2, 3, 1)$$

Abbildung 3.11.: Beispiel für die sich aus Abbildung 3.9 ergebende Reihenfolge, in der die Transfers nach dem *Longest Job First* Algorithmus an Knoten v gestartet werden.

erlauben, einzelne Transfers nicht vollständig laufen zu lassen, sondern zu pausieren, um einen anderen zu starten oder fortzusetzen [24]. Solche Algorithmen sind insbesondere dann besonders relevant, wenn nicht alle durchzuführenden Transfers schon zu Beginn bekannt sind, da sich so die die durchschnittliche Transferbearbeitungszeit (siehe Definition 3) verringern lässt. Auch von Bedeutung sind unterbrechende Algorithmen, wenn die gleichmäßige Abarbeitung der Transfers zur Steigerung der Fairness ein relevantes Kriterium ist oder die Transfers bestimmte Fristen einhalten müssen [3, 24].

Im Kontext dieser Arbeit spielen unterbrechende Algorithmen jedoch keine weitere Rolle, da die durch sie eingeführte Komplexität im Umgang mit einzelnen Bulk-Transfers in Hinblick auf die später durchgeführten Experimente den Rahmen sprengen würde. Zudem wird vereinfachend davon ausgegangen, dass alle zu versendenden Transfers von Beginn an am Startknoten vorliegen und Fristen keine relevante Rolle spielen. Der Vollständigkeit halber sollen einige Scheduling-Algorithmen, die unterbrechend arbeiten jedoch im Folgenden kurz erwähnt werden.

Klassische Algorithmen, die unterbrechend arbeiten, sind *Shortest Time-to-Completion First* und *Round Robin*. Ersteren kann man als Erweiterung von *Shortest Job First* einführen. In Szenarien, wo die Transfer-Details nicht schon von Beginn an bekannt sind, kann es hinsichtlich der Minimierung der durchschnittlichen Transferzeit sinnvoll sein, den Algorithmus derart zu verändern, dass ein laufender Transfer unterbrochen wird, wenn ein Transfer ankommt, dessen zu versendende Datenmenge kleiner ist als die Restmenge des laufenden Transfers. Äquivalent dazu kann *Longest Time-to-Completion First* als Erweiterung von *Longest Job First* eingeführt werden, die unterbrechend arbeitet. Bei *Round Robin* hingegen steht das gleichmäßige Abarbeiten aller Transfers im Fokus, wobei die Transfers in kleinere Datenmengen unterteilt werden, die abwechselnd versendet werden [3].

3.4.5. Bewertung der Algorithmen

In Abschnitt 3.1.1 wurden die im Kontext dieser Arbeit relevanten Optimierungsprobleme, welche u.a. durch den Einsatz der Scheduling-Algorithmen gelöst werden sollen, definiert als die Maximierung der durchschnittlich genutzten Bandbreite innerhalb des Netzwerks und die Minimierung der durchschnittlichen Transferbearbeitungszeit sowie der maximalen Tansferbearbeitungszeit.

Im Folgenden werden einige theoretische Vorüberlegungen gemacht, wobei zunächst nur die Transfers

B_v von einem einzigen Knoten v aus betrachtet werden. Der Einfachheit halber wird zudem davon ausgegangen, dass eine Datenmenge von $1GB$ innerhalb einer Sekunde am Ziel angekommen ist. Für die Einzelbewertung der Algorithmen wird zunächst nur die durchschnittliche Bearbeitungszeit (siehe Definition 3) betrachtet.

Wie bereits zuvor erwähnt, gehen wir davon aus, dass alle Transfers vom Beginn der Betrachtung an am Knoten bekannt sind. Es ist intuitiv recht schnell klar, dass der *Shortest Job First* Algorithmus hinsichtlich der durchschnittlichen Bearbeitungszeit am besten performt, wenn nur dieser eine Knoten v betrachtet wird. Wir machen uns das an den aufgeführten Beispielen klar. Die durchschnittliche Bearbeitungszeit bei den einzelenen Algorithmen sieht wie folgt aus:

1 *First Come First Serve*: $t^{B_v}_{\text{avg\_completion}} = \frac{1s+11s+19s+31s}{4} = 15,5s$

2 *Shortest Job First*: $t^{B_v}_{\text{avg\_completion}} = \frac{1s+9s+19s+31s}{4} = 15s$

3 *Longest Job First*: $t^{B_v}_{\text{avg\_completion}} = \frac{12s+22s+30s+31s}{4} = 23,75s$

In diesem Beispiel schneidet der *Shortest Job First* Algorithmus leicht besser ab als der *First Come First Serve* Algorithmus. Wesentlich schlechter ist in diesem Fall der *Longest Job First* Algorithmus. Da wir jedoch lediglich die Transfers, die von einem spezifischen Startknoten v ausgehen, betrachten, können wir auf Basis dieser Bewertung keine hinreichende Annahme darüber treffen, ob *Shortest Job First* auch innerhalb eines Netzwerks mit parallelen Transfers sinnvoll ist.

Die durchschnittlich genutzte Bandbreite macht als Metrik bezüglich der Bewertung der Scheduling-Algorithmen bei Betrachtung lediglich eines Knotens keinen Sinn, da es keine weiteren Knoten gibt, mit denen sich die Bulk-Transfers die Bandbreite teilen müssten. Eine Bewertung der parallelen Nutzung der Scheduling-Algorithmen an verschiedenen Knoten geschieht im Rahmen der Experimente in Kapitel 4.6.

3.5. Weitere Arbeiten und Eingrenzung

Wie bereits in den vorherigen Kapiteln gezeigt, werden einige Themenbereiche auf Teilaspekte eingegrenzt, die eine experimentelle Betrachtung vereinfachen. So werden in Kapitel 3.1.1 die in dieser Arbeit verwendeten Bewertungsmetriken *Average Throughput*, *Average Transfer Completion Time* und *Longest Completion Time* für Bulk-Transfers eingeführt. Weitere Metriken für Bulk-Transfers werden beispielweise in [21] oder in [24] betrachtet, wo Bulk-Transfers zwischen Rechenzentren analysiert werden.

Eine weitere Eingrenzung ist die Betrachtung ausschließlich hybrider Topologien, für die gilt, dass ein Netzwerk für anstehende Transfers nur einmalig rekonfiguriert wird. Dies folgt weiteren Arbeiten [7, 11],

wobei [7] sogar zeigt, dass durch die Betrachtung von Langzeitcharakteristiken des Traffics eine einmalige Einstellung am Tag in Rechenzentren von Facebook ausreichend ist.

Weiter werden in dieser Arbeit keine preemptiven Scheduling-Algorithmen betrachtet. Außerdem wird davon ausgegangen, dass zum Beginn der Experimente alle zu versendeten Transfers bereits bekannt sind und für die Bulk-Transfers Fristen keine Rolle spielen. All das sind vereinfachende Annahmen, die den Rahmen der Bachelorarbeit eingrenzen sollen. In [24] werden weitere Scheduling-Algorithmen im Kontext von Bulk-Transfers diskutiert.

LongestPathFirst ist ein Algorithmus, der keine Kenntnisse über den dem Netzwerk zugrundeliegenden Traffic voraussetzt. Obwohl es naheliegt, dass die Einstellung eines Netztwerks anhand der anstehenden Transfers effizient ist, gibt es weitere Arbeiten [25], die rekonfigurierbare Netzwerke nicht nach den Traffic-Bedingungen einrichten und dafür Performancegewinne feststellen konnten.

4. Experimentelle Untersuchungen in Mininet

In den folgenden Experimenten wird untersucht, inwiefern sich mit rekonfigurierbaren Kanten und der Verwendung von Scheduling-Algorithmen in einem Netzwerk hinsichtlich Bulk-Transfers anhand der definierten Performance-Metriken (Kapitel 4.4.3) ein Performancegewinn erzielen lässt. Betrachtet wird eine einfache, zufällige Topologie aus 10 Knoten (Kapitel 4.4.1) mit 20 zu Beginn der Experimente bekannten Bulk-Transfers (Kapitel 4.4.2). Die Ergebnisse lassen sich wie in Kapitel 4.5 gezeigt wird skalieren, wodurch die Ergebnisse auch für ganze Rechenzentrums-Netzwerke Gültigkeit besitzen.

4.1. Mininet

Mininet ist ein *open-source* Emulator von vollständigen Netzwerken inklusive Hosts, Switchen, Routern und deren Verbindungen. Durch die prozessbasierte Virtualisierung und separate Netzwerk-*namespaces* können die Knoten unabhängig Prozesse starten und somit reale Anwendungen ausführen. Beliebige Netzwerktopologien können erstellt werden, auf der mit allen dem Host-System zur Verfügung stehenden Technologien experimentiert werden kann. Besonders geeignet ist Mininet für die Erstellung und Untersuchung von SDN, durch die Möglichkeit SDN-Controller zu integrieren, da keine teure Hardware angeschafft werden muss und Ergebnisse schnell geteilt werden können [22].

Mininet wird für Experimente in vergleichbaren wissenschaftlichen Arbeiten [25, 28, 32, 43] verwendet, weshalb der Netzwerkemulator auch in dieser Arbeit für die praktische Untersuchung verwendet wird.

4.1.1. IPMininet

IPMininet ist eine *open-source* Erweiterung des Netzwerkemulators Mininet und ist speziell für die vereinfachte Virtualisierung von IP-Netzwerken entwickelt worden. IPMininet stellt eine API zur Verfügung, die die Konfiguration sämtlicher Netzwerkkomponenten (Interfaces, Prozesse, Routen, ...) übernimmt und dem Anwender die Konfiguration eines ganzen Netzwerks mit der Beschreibung einer Topologie ermög-

licht [19]. Dadurch eignet sich IPMininet besonders gut um Routing-Protokolle wie BGP oder OSPF zu simulieren.

Im Kontext dieser Bachelorarbeit wird IPMininet eingesetzt um das Routing im Netzwerk nicht manuell einrichten zu müssen. Bei der Implementierung einer Topologie werden automatisch mit Hilfe von OSPF (Kapitel 3.2.1) die kürzesten Pfade berechnet, sodass Transfers immer die günstigste Route zum Ziel in den entwickelten Netzwerken (Kapitel 4.4.1) nehmen.

4.2. NetworkX

Die Python-Bibliothek *NetworkX* erlaubt die Erstellung, Modifizierung und Untersuchung von Netzwerken und Netzwerkalgorithmen. *NetworkX* stellt beispielsweise Funktionen bereit um Netzwerktopologien zu erstellen und die resultierenden Pfade in einem Netzwerk zu untersuchen [15].

NetworkX wird in dieser Arbeit für die Erstellung der Topologien und zur Untersuchung der kürzesten Pfade (Kapitel 4.4.1) verwendet und folgt damit weiteren wissenschaftlichen Arbeiten [32, 39], die darauf zurückgreifen.

4.3. iperf3

iperf3 ist ein open-source Programm, das aktive Messungen durchführt, um die maximale erreichbare Bandbreite in IP-Netzwerken zu bestimmen [9].

Die für die Experimente zu generierenden Transfers werden mit *iperf3* erstellt und analysiert. Dazu stellt das Programm eine Auswertung in JSON zur Verfügung, die mit Hilfe selbst geschriebener Skripte analysiert werden können. *iperf3* findet für diesen Zweck auch in vergleichbaren wissenschaftlichen Arbeiten [20, 26, 28] Anwendung.

4.4. Methodik

Die im Rahmen dieser Bachelorarbeit durchgeführten Untersuchungen bestehen aus fünf Experimenten mit jeweils zwölf Varianten, die sich aus den drei Scheduling-Algorithmen für Bulk-Transfers (Kapitel 3.4), sowie den vier Topologien (Kapitel 4.4.1), die aus den drei Algorithmen zur Einstellung der rekonfigurierbaren Kanten resultieren, zusammensetzen. Die Experimente besitzen dieselbe statische Topologie als Startvoraussetzung, unterscheiden sich aber in den geplanten Bulk-Transfers, wodurch sich auch neue hybride Topologien (Abbildung 4.1) ergeben. Jede Variante der Experimente läuft zehn Mal, damit

Ungenauigkeiten der Messungen oder andere Störfaktoren reduziert werden. Insgesamt gibt es also 600 Versuchsdurchläufe, die durch ihre durchschnittliche Laufzeit von acht Minuten zu einer gesamten Laufzeit von 80 Stunden für alle Experimente führen. Im Folgenden werden die zehn Versuchsdurchläufe pro Experimentvariante zusammengefasst betrachtet. Dazu wird für jede Variante das arithmetische Mittel der Ergebnisse über die zehn Versuchsdurchläufe gebildet.

Alle Experimente wurden mit dem in Kapitel 4.1 vorgestellten Netzwerkemulator Mininet (Kapitel 4.1), bzw. der Erweiterung IPMininet (Kapitel 4.1.1) durchgeführt. Dazu wurde für jede Variante des Experiments ein eigenes Python-Skript erzeugt, welches die für IPMininet erforderlichen Python-Bibliotheken importiert (Programmcode 4.1: Zeile 1-3), das Netzwerk aufbaut (Programmcode 4.1: Zeile 5-17) und anschließend mit *iperf3* (Kapitel 4.3) Traffic über das Netzwerk leitet (Programmcode 4.1: Zeile 19-32).

Wie dem Programmcode 4.1 entnommen werden kann, werden die Hosts, also die Knoten, als Router in IPMininet definiert. Ursache ist, dass IPMininet die Routing-Prozesse ausschließlich auf den Routern standardmäßig ausführt und dadurch auf Hosts kein Routing möglich gewesen wäre. Auf die Ergebnisse in den Experimenten hat dies keine Auswirkungen. Die im Netzwerk erzeugten Kanten besitzen, wie in vergleichbaren Arbeiten [25, 43], eine Bandbreite von 100 Mbit/s. Die Kanten erhalten außerdem eine individuelle Gewichtung (*igp_metric*), wodurch die Pfade im Netzwerk eindeutig definiert sind und sich innerhalb meherer Versuchsdurchläufe nicht unterscheiden können. Bei einheitlicher Gewichtung wäre die Gefahr entstanden, dass es zwei gleich lange Pfade gibt und bei jedem Versuchsdurchlauf von *OSPF* ein anderer Pfad als default Pfad festgelegt worden wäre. Die Gewichtung wird aber so gewählt, dass diese nicht grundsätzlich die Pfade ändert, sondern lediglich Pfade mit Knoten über höhere Indizes präferiert.

4.4.1. Topologien

Die Vielzahl der möglichen Topologien für Netzwerke in Rechenzentren macht es schwierig allgemeingültige Aussagen für all diese Topologien zu treffen. Um den Umfang einer Bachelorarbeit nicht zu überschreiten, werden daher analog zu anderen wissenschaftlichen Arbeiten [4, 37, 40] zufällig generierte Netzwerke betrachtet.

Die Knoten *(z.B. Top-of-the-Rack Switche)* sind durch statische (z.B. elektrische) und rekonfigurierbare (z.B. optische) Kanten miteinander verbunden. Potentiell existierende Switches und Router können in den Topologien weg abstrahiert werden, sodass nur die resultierenden Pfade zwischen den Knoten in den Topologien dargestellt werden. Die Generierung der zufälligen Topologien, mit vorerst ausschließlich statischen Kanten, erfolgt anhand eines Skripts, welches anhand der Parameter

```
import ipmininet
from ipmininet.ipnet import IPNet
from ipmininet.iptopo import IPTopo

class Topo(IPTopo):

    def build(self, *args, **kwargs):

        # Add hosts
        h0 = self.addRouter('h0')
        h1 = self.addRouter('h1')
        h2 = self.addRouter('h2')

        # Add links
        self.addLink(h0, h1, igp_metric=1000, bw=100, delay="5ms")
        self.addLink(h1, h2, igp_metric=999, bw=100, delay="5ms")
        self.addLink(h2, h0, igp_metric=998, bw=100, delay="5ms")

def perfTest():

    # Setup hosts and start a traffic flow from h0 to h2
    h0 = net.routers[0]
    h1 = net.routers[1]
    h2 = net.routers[2]
    h2.cmd('iperf3 -s -p 5000 &')
    h0.cmd('iperf3 -c h2 -p 5000 -n 100M -J --logfile /tmp/h0-h2.json &')

# Start network and run performance test
net = IPNet(topo=Topo(), use_v6=False)
net.start()
perfTest()
net.stop()
```

Programmcode 4.1: Reduziertes Beispiel eines Experimentaufbaus. Der Programmcode einer echten Experimentvariante befindet sich in Anhang B.

- Grad (wie viele statische Kanten pro Knoten existieren sollen)
  und
- Anzahl der Knoten

konfiguriert werden kann:

```
# python utils/generate_random_graph.py <degree> <number of nodes>
```

Das Skript erzeugt einen Graphen mit der gewählten Anzahl an Knoten und einer zufälligen Konfiguration aus Kanten von Knoten zu Knoten, die dem gewählten Grad entsprechen. Dabei verfügt jeder Knoten über so viele statische Kanten, wie durch den Grad des Knotens definiert wurden. Erzeugt wird der Graph mit der Funktion *random_regular_graph(d, n, seed=None)* [30] der Python-Bibliothek *networkx* [15].

Die zufällig erzeugten statischen Netzwerke können nun anhand der in 3.3.2 vorgestellten Algorithmen, wie in 3.3.1 einmalig eingestellt werden, wodurch sich neue Topologien (Abbildung 4.1 (b), (c) und (d) und die Abbildungen A.1, A.2, A.3 und A.4) für die Experimente ergeben.

4.4.2. Workloads

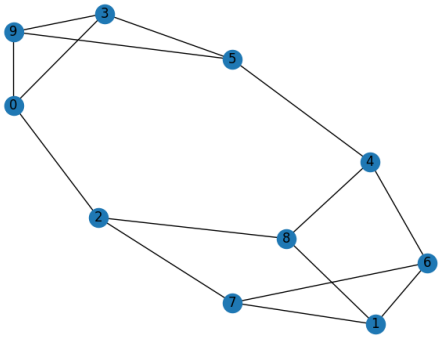
Jedes der durchgeführten Experimente besitzt 20 Transfers, deren Größen anhand einer Exponentialverteilung festgelegt werden. Über eine Gleichverteilung werden die Start- und Zielknoten der Transfers definiert. Da für diese Arbeit keine Traffic-Mitschnitte aus echten Netzwerken existieren, greifen wir auf zufällig generierten Traffic zurück. Dies folgt der Generierung zufälligen Traffics aus vergleichbaren wissenschaftlichen Arbeiten [14, 21, 23].

Die Python-Bibliothek *random* [1] bietet dazu die Funktion *expovariate* [1], welche das Inverse des angestrebten Mittelwerts der Transfers als Parameter *lambda* übergeben bekommt (Abbildung 4.2). Realisiert wird dies in Python mit der Variable *avg_flow_size*, die die durchschnittliche Größe eines Transfers angeben soll (Abbildung 4.3).

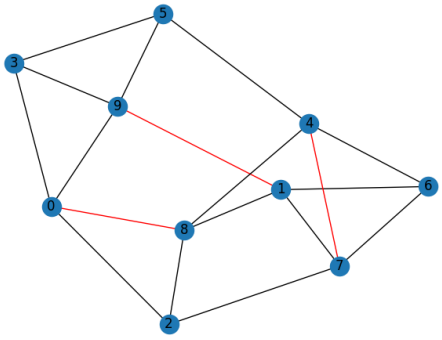
Nachdem alle nötigen Informationen der Transfers vorhanden sind, werden diese für die Experimente in Code, anhand der jeweiligen Scheduling-Algorithmen sortiert, umgesetzt (Programmcode 4.1) und mit *iperf3* erzeugt. Ein Beispiel erzeugter Transfers in JSON und Python-Code ist im Anhang (Kapitel C) angehängt.

4.4.3. Performance-Metriken

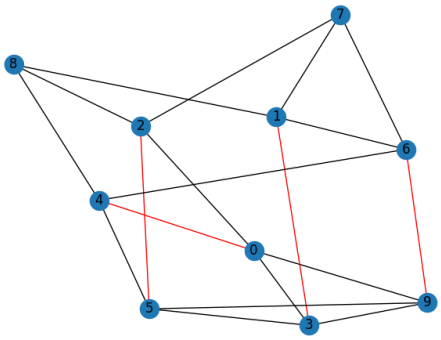
In dem Grundlagen-Kapitel 3 wurden einige Bewertungsmetriken vorgestellt, die wir im folgenden zur Bewertung der Performance nutzen wollen. Sie bieten einen Querschnitt über die im Netzwerk relevanten



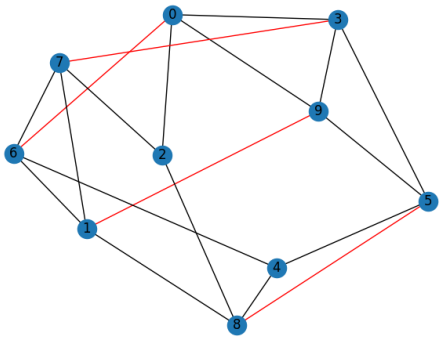
(a) Static



(b) DemandFirst



(c) LongestPathFirst



(d) DemandAwareLongestPathFirst

Abbildung 4.1.: Zufällig generierte Topologie (a) mit 10 Knoten, sowie die anhand der Algorithmen für rekonfigurierbare Kanten (3.3.2) erweiterten Topologien (b), (c) und (d) des ersten Experiments.

$$\text{lambda} = \frac{1}{\text{Mittelwert der Transfers}}$$

Abbildung 4.2.: Definition des Parameters *lambda*.

```
random.expovariate(1/avg_flow_size)
```

Abbildung 4.3.: *expovariate* Funktion der Python-Bibliothek *random*.

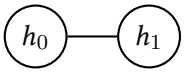


Abbildung 4.4.: Topologie des ersten Reproduzierbarkeits-Tests.

Performance-Aspekte. Auf weitere Metriken wird an dieser Stelle verzichtet, da dies den Umfang der Experimente in einer Bachelorarbeit überschreiten würde.

Die Ergebnisse werden im Folgenden anhand der Performance-Metriken *Average Throughput*, *Average Transfer Completion Time* und *Average Longest Completion Time* analysiert.

4.5. Reproduzierbarkeit

Bevor die Ergebnisse der Untersuchungen vorgestellt werden, soll gezeigt werden, dass die hier erbrachten Leistungen reproduziert und in realen Szenarien angewendet werden können. Auch wenn, wie bereits gezeigt, Mininet in vielen wissenschaftlichen Arbeiten Verwendung findet und den echten *TCP/IP-Stack* des unterliegenden Betriebssystem verwendet, soll anhand einfacher Beispiele die Korrektheit der Emulation gezeigt werden.

Reproduzierbarkeit 1. Es soll untersucht werden, wie groß die Auslastung einer minimalen Topologie, bestehend aus zwei Knoten und einer Kante, bei einem einzigen anstehenden Bulk-Transfer ist.
Versuchsaufbau: In Mininet wird ein Netzwerk aus zwei Knoten erstellt, die durch eine Kante mit einer Bandbreite von 100 Mbit/s verbunden sind (Abbildung 4.4). Es wird für 20 Sekunden ein Transfer von Knoten h_0 zu Knoten h_1 mittels *iperf3* erzeugt.
Annahme: Es wird davon ausgegangen, dass der Transfer die durch die Kante zur Verfügung gestellte Bandbreite von 100 Mbit/s voll auslastet.
Ergebnisse: Der Transfer erreicht eine erwartete Auslastung von 100.00 Mbit/s.

Reproduzierbarkeit 2. Nun soll geprüft werden, wie sich zwei Transfers innerhalb eines Netzwerks verhalten, wenn diese gleichzeitig starten und sich eine Kante teilen.
Versuchsaufbau: Ein Netzwerk aus vier Knoten wird erstellt, wobei h_0 und h_1 Kanten zu h_2 besitzen und h_2 eine Kante zu h_3 hat (Abbildung 4.5). Es wird jeweils ein Transfer für 30 Sekunden von h_0 und h_1 zu h_3 erzeugt.
Annahme: Die beiden Transfers teilen sich zu gleichen Stücken die verfügbare Bandbreite der gemeinsamen Kante von h_2 zu h_3 von 100 Mbit/s.
Ergebnisse: Der Transfer von h_0 zu h_3 hat eine Bandbreite von 50.80 Mbit/s in Anspruch genommen und der Transfer von h_1 zu h_3 von 50.50 Mbit/s. Nehmen wir eine Fehlertoleranz von etwa 1% in Kauf, so

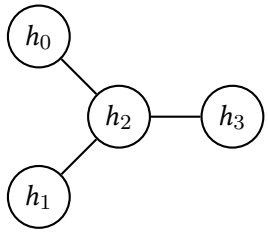


Abbildung 4.5.: Topologie des zweiten Reproduzierbarkeits-Tests.

stimmen die Ergebnisse mit der Annahme überein und die verfügbare Bandbreite von 100 Mbit/s verteilt sich ungefähr gleichmäßig auf beide Transfers.

Skalierbarkeit. Alle in dieser Arbeit erzielten Ergebnisse betrachten Netzwerke mit einer maximalen Bandbreite von 100 Mbit/s pro Kante. Da Netzwerke in realen Rechenzentren häufig Glasfaserverbindungen von 10 Gbit/s oder mehr aufweisen, ist die Skalierbarkeit der Ergebnisse für den Realitätsbezug entscheidend. Da Mininet eine maximale Bandbreite von 1000 Mbit/s unterstützt und aber auch das verwendete Computersystem bei Analysen oberhalb dieses Werts langsam an seine Grenzen kommt, betrachten wir die durchgeführten Reproduzierbarkeits-Tests erneut mit unlimitierten Kanten hinsichtlich der Bandbreite. Für den Test 1 erhalten wir eine genutzte Bandbreite von 6.06 Gbit/s. Bei Test 2 erhalten wir für beide Transfers eine genutzte Bandbreite von 3.12 Gbit/s, also genau wie zuvor eine gleichmäßige Aufteilung der genutzten Bandbreite auf beide Transfers. Wir können daher davon ausgehen, dass die hier gezeigten Ergebnisse nicht nur korrekt, sondern auch skalierbar sind.

Alle in dieser Bachelorarbeit durchgeführten Experimente wurden auf einem virtuellen Root-Server *RS 4000 G9.5* von *netcup* durchgeführt. Der virtuelle Server besaß 10 dedizierte Kerne eines *AMD EPYC™ 7702* Prozessors mit einer Taktung von bis zu 3,35Ghz je Kern [29]. Auf dem Server lief ein *Debian GNU/Linux 10 (buster)*. Im GitHub-Repository [18], sowie im erweiterten Anhang dieser Arbeit, befindet sich ein Skript mit dem Namen *deploy.sh*, welches die Versuchsumgebung automatisiert erstellt. Mit dem Skript *autorun.sh* können die Experimente ausgeführt werden. Die zur Visualisierung der Ergebnisse verwendeten Auswertungen befinden sich im Verzeichnis *experiments/results/* des Repositories.

4.6. Ergebnisse

Die Ergebnisse der fünf Experimente werden separat anhand der drei vorgestellten Performance-Metriken *Average Throughput*, *Average Transfer Completion Time* und *Longest Completion Time* (Kapitel 4.4.3) betrachtet.

Average Throughput. In Abbildung 4.6 wird die durchschnittliche Bandbreite über alle fünf Experimente hinweg veranschaulicht. So kann an der X-Achse die jeweilige Variante der Experimente, also die Kombination aus Algorithmus für die rekonfigurierbaren Kanten und Scheduling-Algorithmus abgelesen werden. An der Y-Achse befindet sich die durchschnittliche Bandbreite in Megabit pro Sekunde (*Mbit/s*). Die Werte setzen sich aus der durchschnittlich genutzten Bandbreite der einzelnen Experimentvarianten zusammen.
Deutlich zeigt sich, dass die statische Topologie den hybriden Topologien hinsichtlich der durchschnittlichen Bandbreite unterlegen ist. Die Boxen der statischen Topologien (*FCFS-STATIC, SJF-STATIC, LJF-STATIC*) liegen vollständig, bzw. nahezu vollständig unter den Boxen der hybriden Topologien. Die oberen Quartile der statischen Topologie liegen also unter den unteren Quartilen der hybriden Topologien.
Die Tabelle in Abbildung 4.7 zeigt das arithmetische Mittel der durchschnittlichen Bandbreiten der Experimentvarianten über die fünf Experimente. Auch hier wird deutlich, dass die durchschnittlich genutzte Bandbreite der Transfers in der statischen Topologie unter der von den hybriden Topologien liegt. Um mindestens etwa 18% verbessert sich die durchschnittliche Bandbreite, wenn ein statisches Netzwerk um rekonfigurierbare Kanten ergänzt wird. Zwischen den Varianten, die ausschließlich auf hybriden Netzwerken basieren, besteht unter Berücksichtigung der verschiedenen Scheduling-Algorithmen ein maximaler Performance-Unterschied von 6,3%, wobei der effizienteste Algorithmus, hinsichtlich der genutzten Bandbreite, *FCFS* in einem Netzwerk, dass nach *DemandFirst* eingestellt wurde, ist. Diese Experimentvariante erzeugt einen Performancegewinn von über 28% im Vergleich zu einem statischen Netzwerk mit *FCFS*. Auch für ein nach *LongestPathFirst* eingestelltes Netzwerk ist *FCFS* im Schnitt die beste Wahl, wobei *SJF* einen besseren Median und eine kleinere Varianz aufweist. *LJF* hat die durchschnittlich beste Bandbreite auf *DemandAwareLongestPathFirst*-Topologien und weist auch eine kleinere Varianz der Ergebnisse als *SJF* und *FCFS* auf.

Average Transfer Completion Time. Wie zuvor für den *Average Throughput*, wird auch die durchschnittliche Zeit bis ein Transfer übermittelt wurde, in einem Boxplot (Abbildung 4.8) und in einer Tabelle (Abbildung 4.9) visualisiert. Auch hier zeigt die X-Achse des Boxplots die Varianten der Experimente. Die Y-Achse zeigt in Sekunden wie lange ein Transfer durchschnittlich benötigt hat, um vollständig übermittelt zu werden. Der Boxplot beinhaltet die durchschnittliche *Transfer Completion Time* der Varianten der fünf Experimente.
Es zeigt sich, dass wie trivialerweise angenommen werden konnte, die durchschnittliche Fertigstellungszeit der Transfers deutlich geringer ist, wenn diese vorher anhand des Scheduling-Algorithmus *SJF* sortiert wurden und nicht nach *LJF*. Nahezu alle Boxen der *SJF*-Varianten liegen vollständig unter denen mit *LJF*. Die Ergebnisse der *FCFS*-Varianten liegen zwischen denen von *SJF* und *LJF*. Darüber hinaus lässt sich erkennen, dass die Mediane der Fertigstellungszeit für die hybriden Topologien unter denen der sta-

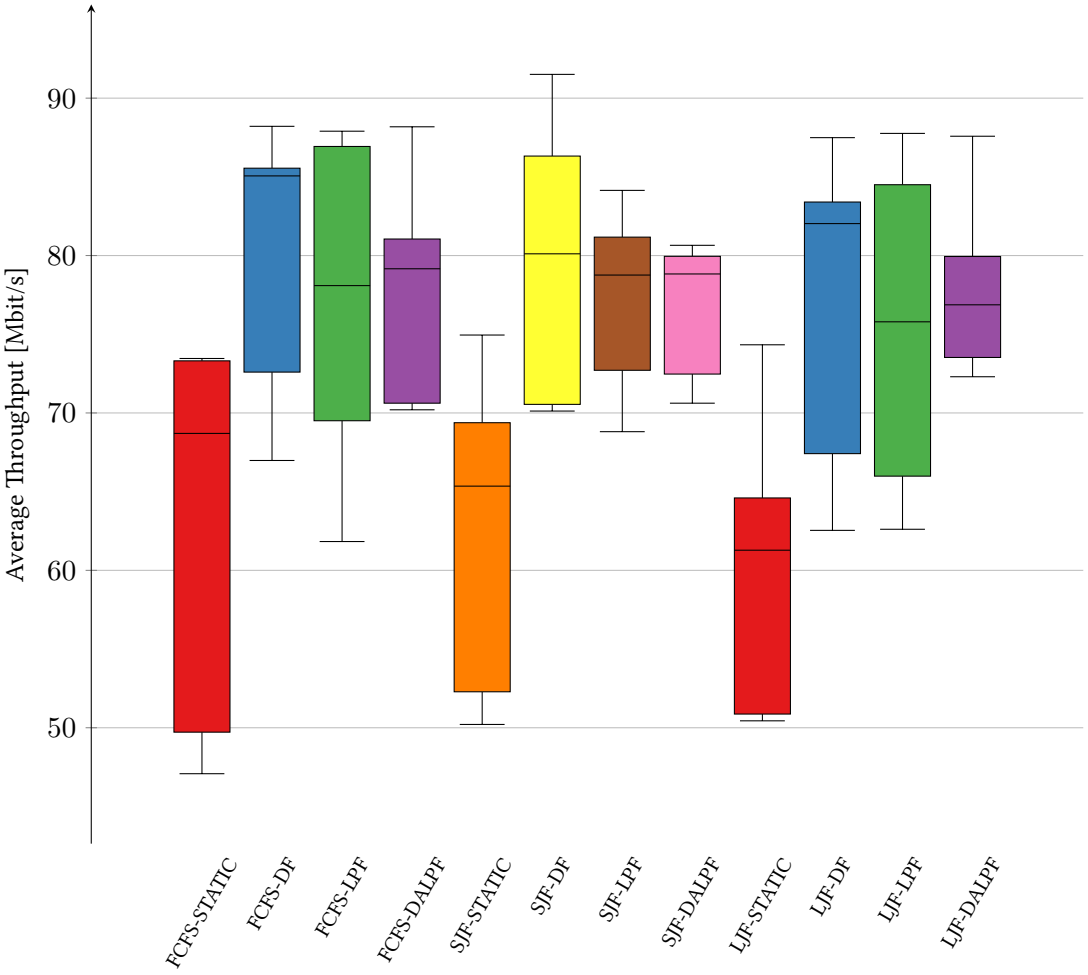


Abbildung 4.6.: Durchschnittliche Bandbreite als Boxplot über alle fünf Experimente.

Average Throughput	FCFS	SJF	LJF
Static	62.98 Mbit/s	62.85 Mbit/s	60.39 Mbit/s
DemandFirst	80.8 Mbit/s	79.80 Mbit/s	77.55 Mbit/s
LongestPathFirst	78.38 Mbit/s	77.89 Mbit/s	76.00 Mbit/s
DemandAwareLongestPathFirst	77.92 Mbit/s	76.87 Mbit/s	78.28 Mbit/s

Abbildung 4.7.: Durchschnittliche Bandbreite als Tabelle über alle fünf Experimente.

tischen liegt, wenn diese ausschließlich innerhalb eines Scheduling-Algorithmus betrachtet werden. Noch leichter ersichtlich ist dies in Abbildung 4.9, wo in den Spalten für die Scheduling-Algorithmen jeweils der höchste Wert, für die durschnittliche Zeit bis ein Transfer übermittelt wurde, in der obersten Zeile, bei der statischen Topologie, steht. Die Werte stellen hier wieder das arithmetische Mittel über die fünf Experimente dar. Auch wenn sich je nach gewählter Topologie die *Average Transfer Completion Time* für *SJF* nur um unter 3% unterscheidet, so lassen sich dennoch verschiedene Merkmale feststellen. Für *SJF* weist *LongestPathFirst* den höchsten Median, aber das geringste arithmetische Mittel auf und besitzt auch über alle Experimente hinweg die kürzeste *Average Transfer Completion Time*. *DemandFirst* mit *SJF* besitzt hier den kleinsten Median. Wenn eine hybride Topologie und *SJF* gewählt wird, so erhält man einen Performancegewinn von knapp 40% im Vergleich zu einem statischen Netzwerk mit *FCFS*. Für ein Scheduling nach *LJF* zeigt sich, dass *DemandAwareLongestPathFirst* die beste Einstellung des Netzwerks erstellt und *LongestPathFirst* für die hybriden Netzwerke die schlechteste Performance erzeugt.

Longest Completion Time. Die Zeit bis der längste Transfer fertiggestellt wurde, wird analog zum *Average Throughput* und zur *Average Transfer Completion Time* visualisiert. Der Boxplot in Abbildung 4.10 zeigt Ergebnisse der einzelnen Experimentvarianten.
Die Boxen der *DemandFirst*-Varianten schlagen für *FCFS* und *LJF* am weitesten nach unten aus und auch für *SJF* liegt das untere Quartil mit den anderen Varianten der hybriden Topologien auf einer Ebene, wobei hier das obere Quartil weit unter den anderen liegt. Die hybriden Topologien performen in jeder Variante besser als das statische Pendant. Im Vergleich zu einem herkömmlichen statischen Netzwerk mit *FCFS* beträgt der Performancegewinn in einem hybriden Netzwerk knapp 8% bis über 25%. Die beste Experimentvariante hinsichtlich der *Longest Completion Time* ist *LJF* und *DemandFirst*. Der neben *DemandFirst* andere Algorithmus, mit Kenntnis über die Transfers, zur Einstellung des Netzwerks *DemandAwareLongest* profitiert auch von *LJF*, während ein nach *LongestPathFirst* eingestelltes Netzwerk von *SJF profitiert*.

Varianz. Aufgrund der verschiedenen *Workloads* (Kapitel 4.4.2) ist die Varianz in den dargestellten Ergebnissen sehr groß. Da die Transfers zufällig generiert wurden, können hier erhebliche Abweichungen zwischen den einzelnen Experimenten existieren. Der Vergleichbarkeit der Ergebnisse schadet dies aber nicht, da jedes Experiment dieselben Varianten beinhaltet und somit die Verhältnisse zwischen den Experimenten identisch sind. Verschiedene Varianten über Experimente hinweg lassen sich aber nicht vergleichen. Aggregiert, wie in diesen Grafiken, können einfach Bewertungen über alle Experimente hinweg erfolgen.

Experiment 1. Es wird in Abbildung 4.12 und 4.13 die durchschnittliche Bandbreite eines einzelnen Experiments, bestehend aus den zwölf Varianten mit jeweils zehn Durchläufen, visualisiert.
Im Gegensatz zu den Ergebnissen, die aggregiert über alle fünf Experimente in Abbildung 4.6 gezeigt wur-

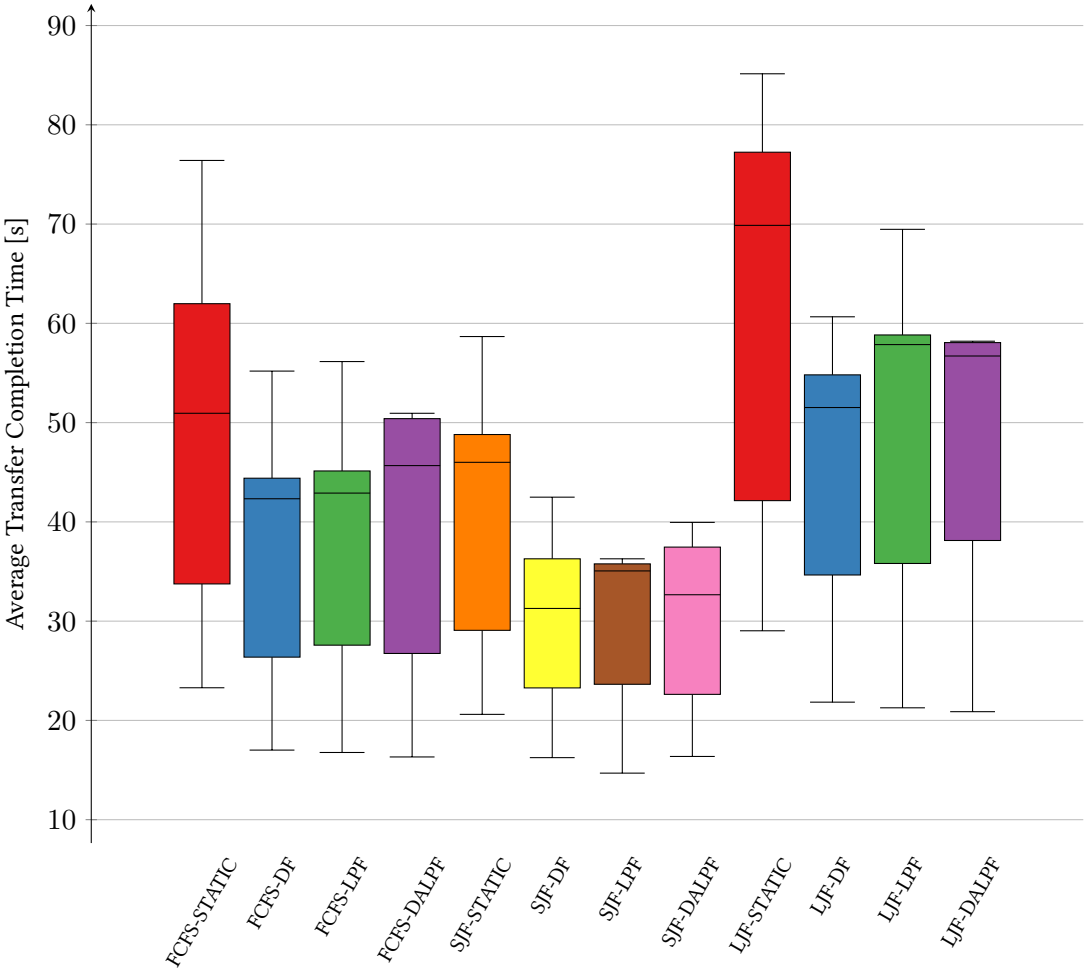


Abbildung 4.8.: Durchschnittliche Zeit bis ein Transfer vollständig übermittelt wurde über alle fünf Experimente als Boxplot.

Average Transfer Completion Time	FCFS	SJF	LJF
Static	51.36 s	42.32 s	63.30 s
DemandFirst	38.93 s	31.32 s	47.25 s
LongestPathFirst	39.86 s	30.87 s	51.55 s
DemandAwareLongestPathFirst	40.10 s	31.06 s	49.84 s

Abbildung 4.9.: Durchschnittliche Zeit bis ein Transfer vollständig übermittelt wurde über alle fünf Experimente als Tabelle.

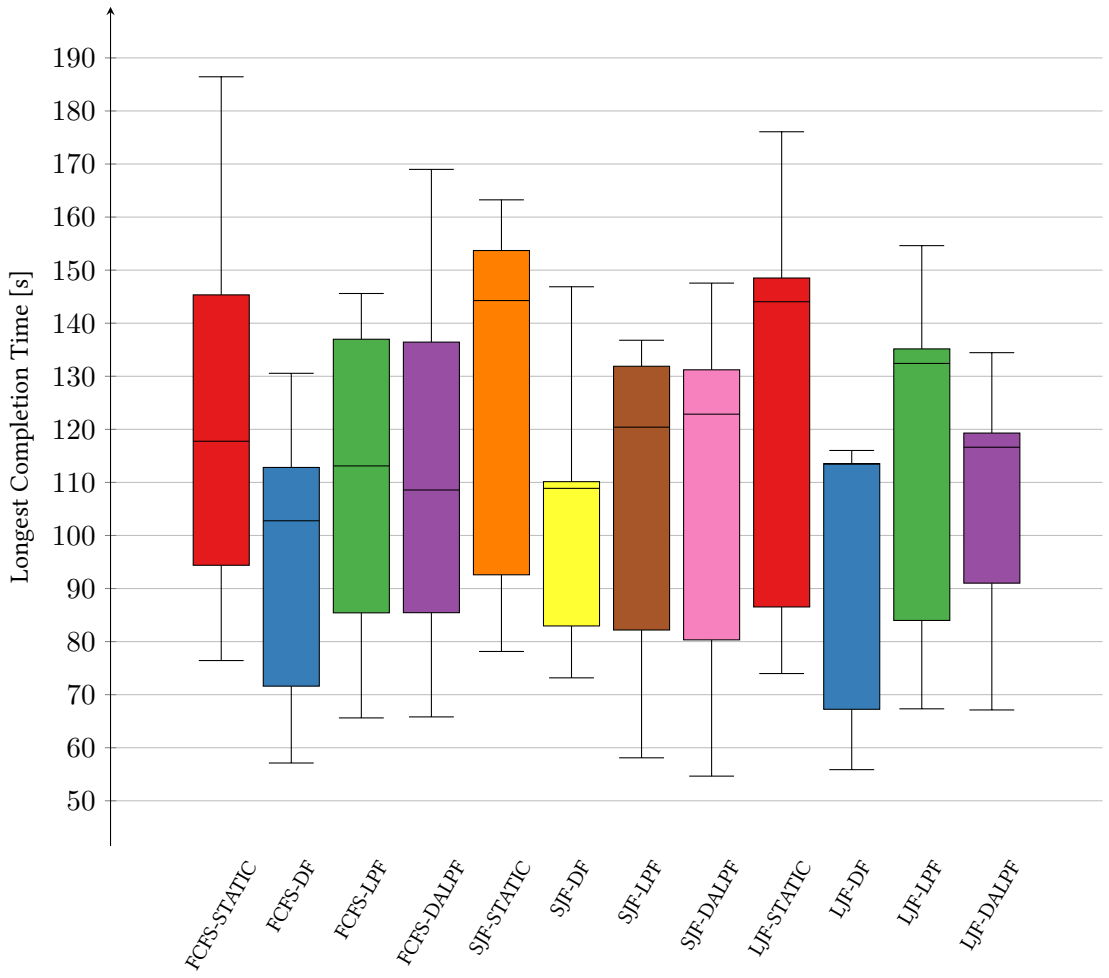


Abbildung 4.10.: Zeit bis der längste Transfer vollständig übermittelt wurde über alle fünf Experimente als Boxplot.

Longest Completion Time	FCFS	SJF	LJF
Static	127.66 s	129.28 s	128.34 s
DemandFirst	97.87 s	106.35 s	95.50 s
LongestPathFirst	113.30 s	110.69 s	118.03 s
DemandAwareLongestPathFirst	116.98 s	112.46 s	110.48 s

Abbildung 4.11.: Zeit bis der längste Transfer vollständig übermittelt wurde über alle fünf Experimente als Tabelle.

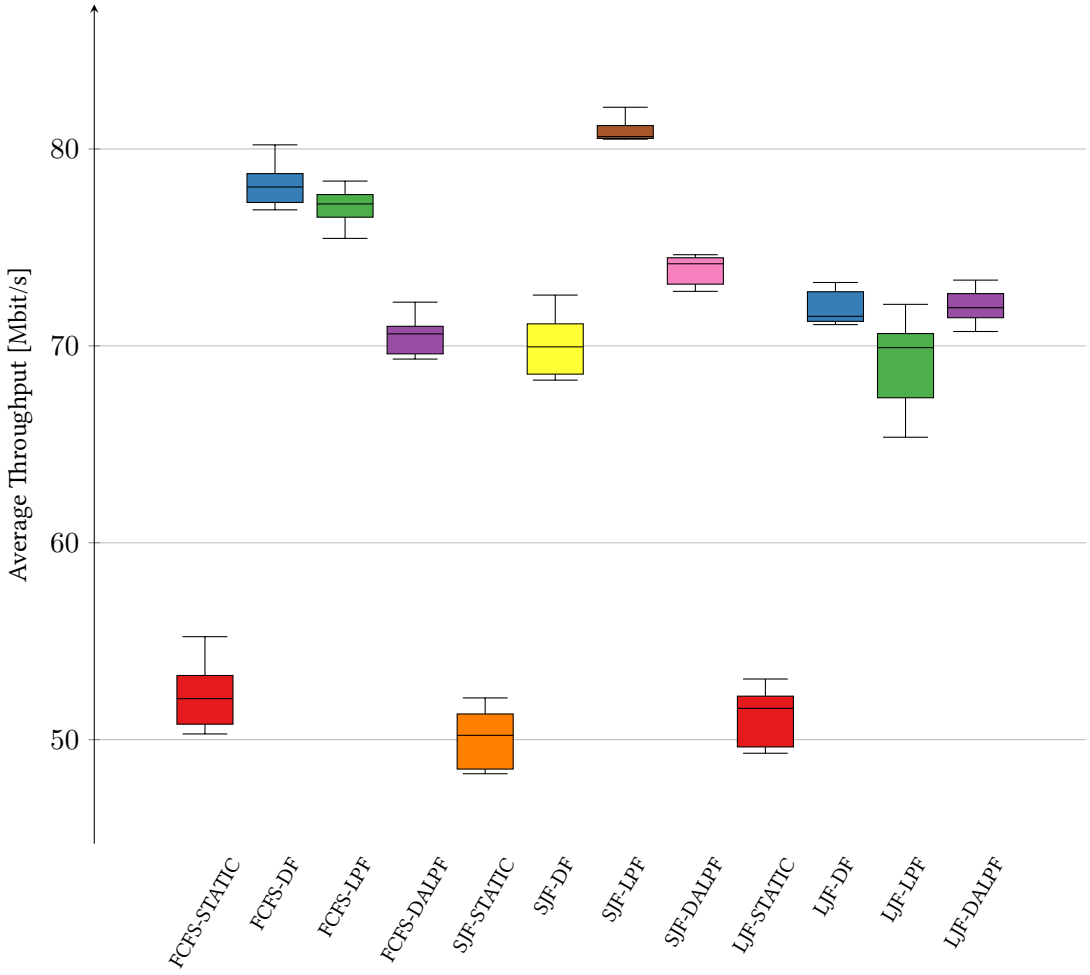


Abbildung 4.12.: Durchschnittliche Bandbreite des ersten Experiments als Boxplot.

Average Throughput	FCFS	SJF	LJF
Static	52.36 Mbit/s	50.21 Mbit/s	51.30 Mbit/s
DemandFirst	78.21 Mbit/s	70.12 Mbit/s	72.29 Mbit/s
LongestPathFirst	77.18 Mbit/s	81.17 Mbit/s	69.36 Mbit/s
DemandAwareLongestPathFirst	70.20 Mbit/s	74.32 Mbit/s	72.30 Mbit/s

Abbildung 4.13.: Durchschnittliche Bandbreite des ersten Experiments als Tabelle.

den, sieht man hier, dass oberes und unteres Quartil der Boxen eng zusammen liegen und somit keine große Varianz existiert. Für *FCFS-STATIC* liegt die Standardabweichung hinsichtlich der Bandbreite bei diesem Experiment beispielsweise bei 1,6, während die Standardabweichung der Werte über alle fünf Experimente bei 12,39 liegt. Für *SJF-LPF* liegt die Standardabweichung sogar nur bei 1,04. Hinsichtlich einer Bandbreite von 100 Mbit/s ist das eine Ungenauigkeit von <2%.

Rekonfigurierbare Kanten. Über den Vergleich anhand der Performance-Metriken hinaus lässt sich für die Algorithmen zur Einstellung der rekonfigurierbaren Kanten feststellen, dass diese unterschiedlich viele neue Kanten den Topologien hinzugefügt haben. Für *LongestPathFirst* wurden jedem Experiment, wie zu erwarten war, immer 5 neue Kanten hinzugefügt. Bei *DemandAwareLongestPathFirst* sind es bei jedem Experiment hingegen nur 4 neue Kanten und für *DemandFirst* sogar im Schnitt nur 3,4.

5. Diskussion

Im Folgenden werden die Ergebnisse aus Kapitel 4.6 analysiert und diskutiert, um anschließend Handlungsempfehlungen zur Nutzung von Scheduling-Algorithmen in hybriden Netzwerken geben zu können. Zunächst werden in Abschnitt 5.1 mögliche Vorteile der Nutzung der eingeführten Scheduling-Algorithmen (Kapitel 3.4) auf den eingeführten hybriden Topologien (Kapitel 4.4.1) betrachtet. Anschließend wird herausgestellt, inwieweit sich in den durchgeführten Experimenten durch die Nutzung hybrider Topologien in Kombination mit Scheduling-Algorithmen ein Vorteil gegenüber der einfachsten Konfiguration, nämlich statischen Netzwerken mit *FCFS*, gezeigt hat.

5.1. Scheduling in hybriden Netzwerken

Um Vorteile in der Anwendung verschiedener Scheduling-Algorithmen in hybriden Netzwerken ausmachen zu können, betrachten wir die drei hybriden Netzwerktopologien in Kombination mit *FCFS* und schauen, ob sich Verbesserungen hinsichtlich der drei betrachteten Performance-Metriken erzielen lassen, wenn andere Scheduling-Algorithmen auf den hybriden Topologien angewendet werden.

Hinsichtlich des **Average Throughputs** zeigt sich, dass für alle betrachteten Topologien *FCFS* eine leicht bessere (0,6%-1,4%) Bandbreite als *SJF* ausweisen kann. *LJF* kann den Transfers auf *DemandFirst* und *LongestPathFirst* durchschnittlich sogar nur 3% bis 4% weniger Bandbreite als *FCFS* zu Verfügung stellen. Leidglich auf *DemandAwareLongestPathFirst* hat ein Scheduling-Algorithmus besser als *FCFS* performen können. So hat *LJF* einen minimal (0.5%) besseren *Average Throughput* als *FCFS*.

Für die **Average Transfer Completion Time** zeigt sich wie in Kapitel 3.4 bereits angenommen, dass *SJF* naturgemäß gegenüber den anderen Scheduling-Algorithmen große Vorteile besitzt, die sich auch in den Ergebnissen der Experimente widerspiegeln. *SJF* erreicht eine etwa 20% bis 23% bessere durchschnittliche Fertigstellungszeit der Transfers im Vergleich mit *FCFS* auf hybriden Topologien. Die beste Kombination aus hybrider Topologie und Scheduling-Algorithmus stellt dabei *SJF* auf einem nach *LongestPathFirst* eingestellten Netzwerk dar. Deutlich schlechter ist die *Average Transfer Completion Time* für *LJF*. Auf allen drei untersuchten hybriden Topologien verschlechtert sich die Performance um etwa 21% bis 29%.

Die **Longest Completion Time** wird, wie sich in den Ergebnissen zeigte, auf allen drei hybriden Topologien durch das gezielte Scheduling verringert. So konnte auf *DemandFirst* und *DemandAwareLongestPathFirst* die Performance durch *LJF* um etwa 2,5% bzw. etwa 5,5% verbessert werden, während auf *LongestPath* durch *SJF* die Performance um etwa 2,3% gesteigert wird.

Aus den Ergebnissen ist gesamtheitlich eine leichte Tendenz erkenntlich, dass *SJF* besser auf *LongestPathFirst* performt, während *LJF* bessere Ergebnisse auf *DemandAwareLongestPathFirst* liefert. Es liegt nahe, dass durch die Priorisierung, der tatsächlich von Transfers genutzten Pfade, bei *DemandAwareLongestPathFirst* eine geeignetere Topologie für *LJF* gebildet wird. Dies könnte auf die *TCP congestion control* [2, 38] zurückzuführen sein, da die Verbindungsgeschwindigkeit der Transfers durch häufig neu eintreffende kleine Transfers (gilt für *SJF*) nicht gedrosselt werden muss. Umgekehrt ist durch das fehlende Bewusstsein über die Transfers von *LongestPathFirst* kein Pfad gezielt für die großen Bulk-Transfers optimiert worden, weshalb sich voraussichtlich auch größere Transfers über längere Pfade die vorhandene Bandbreite teilen müssen und dadurch auch die *Transfer Completion Time* und die *Longest Completion Time* verschlechtern. Noch deutlicher wird der Performanceunterschied von *SJF* und *LJF*, wenn diese auf *DemandFirst*-Topologien verglichen werden, da hier ein noch stärker Fokus darauf liegt, dass die großen Transfers möglichst kurze Wege zu ihren Zielen erhalten.

5.2. Vorteile hybrider Netzwerke

Über die bereits diskutierten Ergebnisse hinaus, lassen sich auch für die verschiedenen Rekonfigurierungsalgorithmen und den daraus folgenden hybriden Topologien im Vergleich mit einer statischen Topologie anhand der Performance-Metriken Unterschiede feststellen. Angemerkt werden muss hier jedoch, dass der Grad der Knoten auf der statischen Topologie, durch die fehlenden rekonfigurierbaren Kanten, 3 beträgt, während dieser für die hybriden Topologien zwischen 3,6 und 4 liegt und somit kein ganz fairer Vergleich erfolgen kann.

Durch die Einführung einer hybriden Topologie lässt sich der **Average Throughput** um etwa 19% bis 22% steigern, wobei *DemandFirst* hierbei die effizienteste Topologie darstellt. Kein Scheduling-Verfahren kann eine weitere Verbesserung liefern.

Auch die **Average Transfer Completion Time** lässt sich durch hybride Topologien deutlich reduzieren. Ergänzt man das statische Netzwerk um nach *DemandFirst* eingestellte Kanten, so erhalten die Transfers eine Performancesteigerung von fast 25%. Wird das Netzwerk dagegen anhand *LongestPathFirst* eingestellt und die Transfers werden anhand *SJF* sortiert, dann kann sogar eine Steigerung der *Average Transfer Completion Time* von fast 40% erreicht werden.

Für die *Longest Completion Time* kann mit Hilfe der hybriden Topologien *DemandFirst* eine kürzere Maximallaufzeit von rund 23% und mit *SJF* anstelle von *FCFS* sogar von rund 25% erzielt werden.

Hervorzuheben ist, dass *DemandFirst* ähnlich effizient (nur etwa 1,5% ineffizienter als *LongestPathFirst* für die *Average Transfer Completion Time*) oder sogar effizienter hinsichtlich aller drei Metriken als die beiden anderen Algorithmen war, obwohl dadurch im Vergleich weniger neue Kanten den Topologien hinzugefügt wurden. Erklärt werden kann dies durch die Priorisierung großer Transfers, wodurch diese ohne über andere Pfade geleitet werden zu müssen zum Zielknoten gelangen können. Doch nicht nur für diese Transfers ist das von Vorteil, sondern auch kleinere Transfers profitieren davon, wenn dadurch längere Pfade entlastet werden.

5.3. Handlungsempfehlungen

Basierend auf der in Kapitel 4.5 gezeigten Reproduzier- und Skalierbarkeit sowie den gezeigten Ergebnissen, lassen sich für Betreiber von Netzwerken Handlungsempfehlungen aussprechen, um die Auslastung zu optimieren und die Bearbeitungszeit der Bulk-Transfers zu minimieren. Da sich die Anforderungsprofile der verschiedenen real existierenden Netzwerke stark unterscheiden, lässt sich allerdings schwierig eine allgemeingültige Empfehlung aussprechen.

Es lässt sich feststellen, dass Betreiber hybrider Netzwerke kein echtes Steigerungspotential des *Throughputs* durch Scheduling erhalten. Nur an Betreiber statischer Netzwerke kann an dieser Stelle die allgemeingültige Empfehlung gerichtet werden, dass die Hinzunahme von rekonfigurierbaren Komponenten ein statisches Netzwerk anhand der in dieser Arbeit definierten Metriken definitiv verbessert.

Sind Netzwerke darauf ausgerichtet, dass die durschnittliche Bearbeitungszeit möglichst minimal ist und es kein Problem darstellt, dass große Bulk-Transfers zurückgestellt werden, dann sollten Netzwerkbetreiber ihre Infrastruktur um Möglichkeiten zum Scheduling erweitern und den Traffic anhand von *SJF* sortieren.

Ist es von großem Interesse die maximale Bearbeitungszeit der Transfers zu minimieren, z.B. wenn die Daten keinen Nutzen vor der vollständigen Übertragung haben, dann sollte ein hybrides Netzwerk nach *DemandFirst* eingestellt werden und das Scheduling sollte nach *LJF* erfolgen.

Das Scheduling der Transfers könnte sowohl auf der SDN-Hardware, als auch direkt in der Applikationsschicht implementiert werden, wobei letzteres einen deutlich größeren Aufwand bedeutet, da dafür eventuell ganze Programmlogiken umgestellt werden müssten.

Die vorgestellte Empfehlung die Netzwerkinfrastruktur um rekonfigurierbare Netzwerkkomponenten zu erweitern, ist keineswegs ein besonders invasiver Eingriff. Wie bei den für die Experimente erstellten

Topologien (Abbildung 4.1), oder auch in Abbildung 3.3, zu sehen kann ein bestehendes Netzwerk einfach durch zusätzliche Verbindungen zu SDN-Hardware (z.B. optische Switche) erweitert werden. Hierzu muss die existierende Topologie nicht umgebaut werden und es sind keine Ausfallzeiten von Systemen zu erwarten. Es kommen lediglich neue Pfade hinzu und alte brechen nicht weg.

6. Ausblick

Die Untersuchungen haben gezeigt, dass sich durch Scheduling der Bulk-Transfers signifikante Performancesteigerungen in hybriden Netzwerken hinsichtlich der durchschnittlichen und der maximalen Bearbeitungszeit ergeben. Es gelang leider keiner Kombination aus rekonfigurierbarem Netzwerk und Scheduling-Algorithmus in allen Metriken hervorzustechen und darüber hinaus hat auch kein Scheduling-Algorithmus die durchschnittlich genutzte Bandbreite steigern können.

Um weitere Tendenzen ablesen zu können und die in dieser Arbeit getätigten Beobachtungen zu bestätigen, müssen weitere Untersuchungen angestellt werden, die größere Topologien mit mehr Transfers betrachten. Dazu werden Simulationssysteme mit mehr Rechenleistung und effizienteren Methoden zur automatisierten Erstellung der Experimentumgebung benötigt. Dies würde auch eine Möglichkeit bieten, die in Kapitel 4.5 gezeigte Skalierbarkeit und damit Anwendbarkeit in realeren Szenarien zu beweisen.

Da wie bereits in der Diskussion erwähnt, der Vergleich der hier verwendeten statischen Topologie mit den hybriden Topologien nicht ganz fair war, sollten die Experimente mit einer hybriden Topologie wiederholt werden, die allerdings zufällig rekonfigurierte Kanten besitzt. So lässt sich der Nachteil der statischen Topologie durch den geringeren Grad der Knoten ausgleichen.

Darüber hinaus sollten auch komplexere Scheduling-Algorithmen (z.B. mit *preemption* oder mit Fristen) auf den hybriden Topologien untersucht und mit den Ergebnissen der gängigen Scheduling-Algorithmen verglichen werden.

Weiter wäre die Betrachtung einer Erweiterung des Rekonfigurationsalgorithmus *DemandFirst* möglich. Bisher reizt der Algorithmus nicht das volle Potenzial hybrider Topologien aus, da wie gezeigt wurde, nicht alle möglichen rekonfigurierbaren Kanten erzeugt werden. Beispielsweise wäre eine Kombination von *DemandFirst* und *LongestPathFirst* möglich, sodass die Kanten erst anhand der größten Transfers dem Netzwerk hinzugefügt werden und anschließend die verbleibenden freien Knoten mittels *LongestPathFirst* verbunden werden.

Bisher unbetrachtet blieb auch der Einfluss von *TCP congestion control*. Untersuchungen der *TCP congestion*-Algorithmen bezüglich der Performance-Metriken könnten sowohl die Performance von *SJF* und *LJF*

auf den verschiedenen hybriden Topologien genauer analysieren, als auch weiteres Performencepotential hervorbringen.

Literaturverzeichnis

[1] Python 3.7. 2018. Libary: random. https://web.archive.org/web/20230605102627/https://docs.python.org/3.7/library/random.html Accessed: 2023-06-05.

[2] Ahmad Abadleh, Aya Tareef, Alaa Btoush, Alaa Mahadeen, Maram M Al-Mjali, Saqer S Alja'Afreh, and Anas Ali Alkasasbeh. 2022. Comparative Analysis of TCP Congestion Control Methods. In *2022 13th International Conference on Information and Communication Systems (ICICS)*. IEEE, 474–478. https://doi.org/10.1109/ICICS55353.2022.9811217

[3] Remzi H. Arpaci-Dusseau. 2017. Operating Systems: Three Easy Pieces. *login Usenix Mag.* 42, 1 (2017). https://www.usenix.org/publications/login/spring2017/arpaci-dusseau

[4] Youssef Baddi, Anass Sebbar, Karim Zkik, Yassin Maleh, Faysal Bensalah, and Mohammed Boulmalf. 2023. MSDN-IoT multicast group communication in IoT based on software defined networking. *Journal of Reliable Intelligent Environments* (2023), 1–12. https://doi.org/10.1007/s40860-023-00203-x

[5] Kevin J. Barker, Alan F. Benner, Raymond R. Hoare, Adolfy Hoisie, Alex K. Jones, Darren J. Kerbyson, Dan Li, Rami G. Melhem, Ramakrishnan Rajamony, Eugen Schenfeld, Shuyi Shao, Craig B. Stunkel, and Peter Walker. 2005. On the Feasibility of Optical Circuit Switching for High Performance Computing Systems. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*. IEEE Computer Society, 16. https://doi.org/10.1109/SC.2005.48

[6] BGB. 2016. Verordnung zur Bestimmung Kritischer Infrastrukturen nach dem BSI-Gesetz. http://web.archive.org/web/20230621205135/https://www.gesetze-im-internet.de/bsi-kritisv/BJNR095800016.html BSI-Kritisverordnung vom 22. April 2016 (BGBl. I S. 958), die zuletzt durch Artikel 1 der Verordnung vom 23. Februar 2023 (BGBl. 2023 I Nr. 53) geändert worden ist. Accessed: 2023-06-21.

[7] Peirui Cao, Shizhen Zhao, Min Yee Teh, Yunzhuo Liu, and Xinbing Wang. 2021. TROD: Evolving From Electrical Data Center to Optical Data Center. In *29th IEEE International Conference on Network*

Protocols, ICNP 2021, Dallas, TX, USA, November 1-5, 2021. IEEE, 1–11. https://doi.org/10.1109/ICNP52444.2021.9651977

[8] Yong Deng, Yuxin Chen, Yajuan Zhang, and Sankaran Mahadevan. 2012. Fuzzy Dijkstra algorithm for shortest path problem under uncertain environment. *Applied Soft Computing* 12, 3 (2012), 1231–1237. https://doi.org/10.1016/j.asoc.2011.11.011

[9] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. 2023. iperf3. https://github.com/esnet/iperf Accessed: 2023-06-21.

[10] Matija Dzanko, Branko Mikac, and Vedran Miletic. 2012. Availability of all-optical switching fabrics used in optical cross-connects. In *2012 Proceedings of the 35th International Convention, MIPRO 2012, Opatija, Croatia, May 21-25, 2012*. IEEE, 568–572. https://ieeexplore.ieee.org/document/6240710/

[11] Thomas Fenz, Klaus-Tycho Foerster, Stefan Schmid, and Anaïs Villedieu. 2020. Efficient non-segregated routing for reconfigurable demand-aware networks. *Comput. Commun.* 164 (2020), 138–147. https://doi.org/10.1016/j.comcom.2020.10.003

[12] Bilal Gonen. 2011. Genetic Algorithm Finding the Shortest Path in Networks. (01 2011). https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=9959b7a98976338dfdc7f6dc6fb6fa6195450010

[13] Chen Griner, Johannes Zerwas, Andreas Blenk, Manya Ghobadi, Stefan Schmid, and Chen Avin. 2021. Cerberus: The Power of Choices in Datacenter Topology Design - A Throughput Perspective. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 3 (2021), 38:1–38:33. https://doi.org/10.1145/3491050

[14] Yingya Guo, Zhiliang Wang, Xia Yin, Xingang Shi, and Jianping Wu. 2014. Traffic Engineering in SDN/OSPF Hybrid Network. In *22nd IEEE International Conference on Network Protocols, ICNP 2014, Raleigh, NC, USA, October 21-24, 2014*. IEEE Computer Society, 563–568. https://doi.org/10.1109/ICNP.2014.90

[15] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States). 11 – 15 pages. https://www.osti.gov/biblio/960616

[16] Matthew Nance Hall, Klaus-Tycho Foerster, Stefan Schmid, and Ramakrishnan Durairajan. 2021. A Survey of Reconfigurable Optical Networks. *Opt. Switch. Netw.* 41 (2021), 100621. https://doi.org/10.1016/j.osn.2021.100621

[17] Matthias Herlich and Holger Karl. 2010. Optimizing Energy Efficiency for Bulk Transfer Networks. *Apr* 13 (2010), 1–3. https://cs.uni-paderborn.de/fileadmin-eim/informatik/fg/cn/Publications_Unrefereed_Publications/Optimizing.pdf

[18] Julian Huhn. 2023. bachelor-thesis Repository on GitHub. https://github.com/huhndev/bachelor-thesis Accessed: 2023-06-22.

[19] Mathieu Jadin, Olivier Tilmans, Maxime Mawait, and Olivier Bonaventure. 2020. Educational Virtual Routing Labs with IPMininet. In *ACM SIGCOMM Education Workshop*. 1–5. https://dial.uclouvain.be/pr/boreal/object/boreal%3A235433/datastream/PDF_01/view

[20] Jehn-Ruey Jiang, Hsin-Wen Huang, Ji-Hau Liao, and Szu-Yuan Chen. 2014. Extending Dijkstra's shortest path algorithm for software defined networking. In *The 16th Asia-Pacific Network Operations and Management Symposium, APNOMS 2014, Hsinchu, Taiwan, September 17-19, 2014*. IEEE, 1–4. https://doi.org/10.1109/APNOMS.2014.6996609

[21] Xin Jin, Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu, Guangzhi Li, Wei Xu, and Jennifer Rexford. 2016. Optimizing Bulk Transfers with Software-Defined Optical WAN. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti (Eds.). ACM, 87–100. https://doi.org/10.1145/2934872.2934904

[22] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM Workshop on Hot Topics in Networks. HotNets 2010, Monterey, CA, USA - October 20 - 21, 2010*, Geoffrey G. Xie, Robert Beverly, Robert Tappan Morris, and Bruce Davie (Eds.). ACM, 19. https://doi.org/10.1145/1868447.1868466

[23] Long Luo, Yijing Kong, Mohammad Noormohammadpour, Zilong Ye, Gang Sun, Hongfang Yu, and Bo Li. 2022. Deadline-Aware Fast One-to-Many Bulk Transfers over Inter-Datacenter Networks. *IEEE Trans. Cloud Comput.* 10, 1 (2022), 304–321. https://doi.org/10.1109/TCC.2019.2935435

[24] Long Luo, Hongfang Yu, Klaus-Tycho Foerster, Max Noormohammadpour, and Stefan Schmid. 2020. Inter-Datacenter Bulk Transfers: Trends and Challenges. *IEEE Network* 34, 5 (2020), 240–246. https://doi.org/10.1109/MNET.011.1900632

[25] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. 2017. RotorNet: A Scalable, Low-complexity, Optical Datacenter Network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017,*

Los Angeles, CA, USA, August 21-25, 2017. ACM, 267–280. https://doi.org/10.1145/3098822.3098838

[26] Sangeeta Mittal. 2017. Performance Evaluation of Openflow SDN Controllers. In *Intelligent Systems Design and Applications - 17th International Conference on Intelligent Systems Design and Applications (ISDA 2017) Held in Delhi, India, December 14-16, 2017 (Advances in Intelligent Systems and Computing, Vol. 736)*, Ajith Abraham, Pranab Kumar Muhuri, Azah Kamilah Muda, and Niketa Gandhi (Eds.). Springer, 913–923. https://doi.org/10.1007/978-3-319-76348-4_87

[27] John Moy. 1998. OSPF Version 2. RFC 2328. https://doi.org/10.17487/RFC2328

[28] May Thae Naing, Thiri Thitsar Khaing, and Aung Htein Maw. 2019. Evaluation of tcp and udp traffic over software-defined networking. In *2019 International Conference on Advanced Information Technologies (ICAIT)*. IEEE, 7–12. https://doi.org/10.1109/AITC.2019.8921086

[29] netcup GmbH. 2023. netcup Root-Server Details. https://web.archive.org/web/20230606175700/https://www.netcup.de/vserver/#root-server-details Accessed: 2023-06-06.

[30] NetworkX. 2021. Function: random_regular_graph. https://web.archive.org/web/20230605100612/https://networkx.org/documentation/networkx-2.6.2/reference/generated/networkx.generators.random_graphs.random_regular_graph.html Accessed: 2023-06-05.

[31] Ida Nurhaida, Desi Ramayanti, and Imay Nur. 2019. Performance Comparison based on Open Shortest Path First (OSPF) Routing Algorithm for IP Internet Networks. *Communications on Applied Electronics* 7 (09 2019), 12–25. https://doi.org/10.5120/cae2019652838

[32] Oluwaseyi Oginni, Peter Bull, and Yonghao Wang. 2018. Constraint-aware software-defined network for routing real-time multimedia. *SIGBED Rev.* 15, 3 (2018), 37–42. https://doi.org/10.1145/3267419.3267425

[33] Michal Pióro, Áron Szentesi, János Harmatos, Alpár Jüttner, Piotr Gajowniczek, and Stanislaw Kozdrowski. 2002. On open shortest path first related network optimisation problems. *Perform. Evaluation* 48, 1/4 (2002), 201–223. https://doi.org/10.1016/S0166-5316(02)00036-6

[34] George Porter, Richard D. Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2013. Integrating microsecond circuit switching into the data center. In *ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong,*

August 12-16, 2013, Dah Ming Chiu, Jia Wang, Paul Barford, and Srinivasan Seshan (Eds.). ACM, 447–458. https://doi.org/10.1145/2486001.2486007

[35] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Muhammad Mukarram Bin Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve D. Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. 2022. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022*, Fernando Kuipers and Ariel Orda (Eds.). ACM, 66–85. https://doi.org/10.1145/3544216.3544265

[36] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271. https://doi.org/10.17487/RFC4271

[37] Ankit Singla, Chi-Yao Hong, Lucian Popa, and Philip Brighten Godfrey. 2012. Jellyfish: Networking Data Centers Randomly. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 225–238. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/singla

[38] W. Richard Stevens. 1997. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. *RFC* 2001 (1997), 1–6. https://doi.org/10.17487/RFC2001

[39] Yinan Tang, Hongxiang Guo, Tongtong Yuan, Xiong Gao, Xiaobin Hong, Yan Li, Jifang Qiu, Yong Zuo, and Jian Wu. 2019. Flow splitter: A deep reinforcement learning-based flow scheduler for hybrid optical-electrical data center network. *IEEE Access* 7 (2019), 129955–129965. https://doi.org/10.1109/ACCESS.2019.2940445

[40] Slavica Tomovic, Nedjeljko Lekic, Igor Radusinovic, and Gordana Gardasevic. 2016. A new approach to dynamic routing in SDN networks. In *2016 18th Mediterranean Electrotechnical Conference (MELECON)*. IEEE, 1–6. https://doi.org/10.1109/MELCON.2016.7495433

[41] Wikipedia. 2023. OSI model. https://web.archive.org/web/20230614193526/https://en.wikipedia.org/wiki/ISO_OSI Accessed: 2023-06-14.

[42] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. 2014. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials* 17, 1 (2014), 27–51. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6834762

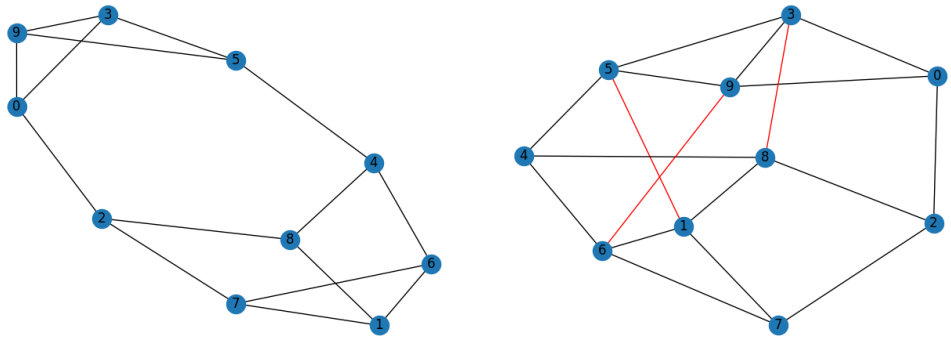
[43] Maiass Zaher, Aymen Hasan Alawadi, and Sándor Molnár. 2021. Sieve: A flow scheduling framework in SDN based data center networks. *Computer Communications* 171 (2021), 99–111. https://doi.org/10.1016/j.comcom.2021.02.013

Abbildungsverzeichnis

Algorithmenverzeichnis

Definitionsverzeichnis

A. Ergänzung Topologien



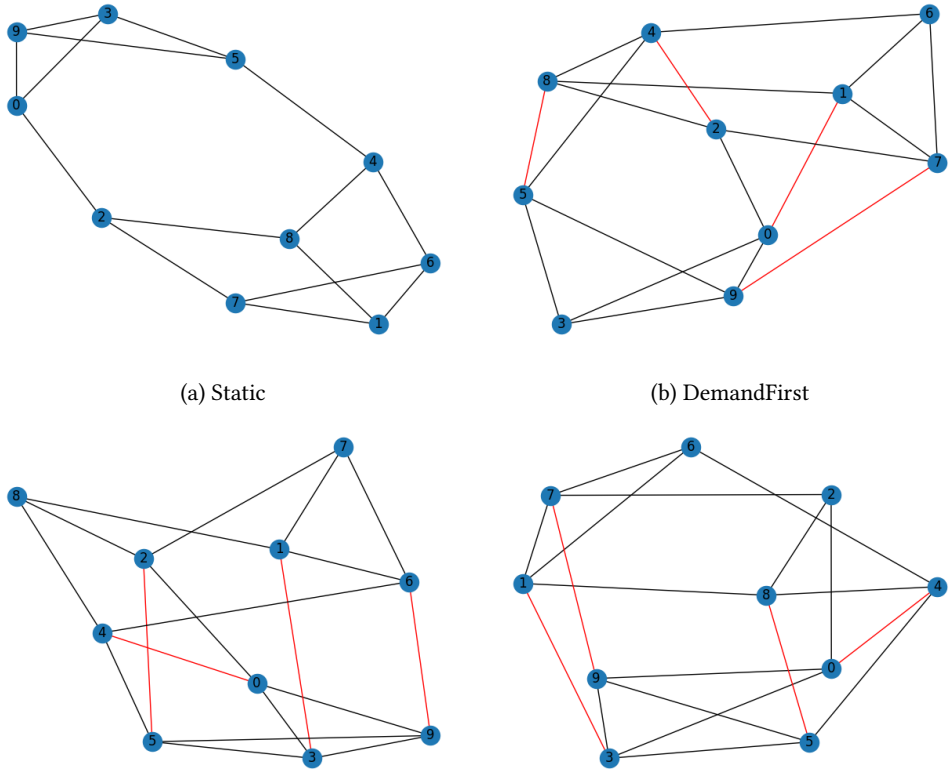
(a) Static

(b) DemandFirst

(c) LongestPathFirst

(d) DemandAwareLongestPathFirst

Abbildung A.1.: Zufällig generierte Topologie (a) mit 10 Knoten, sowie die anhand der Algorithmen für rekonfigurierbare Kanten (3.3.2) erweiterten Topologien (b), (c) und (d) des zweiten Experiments.



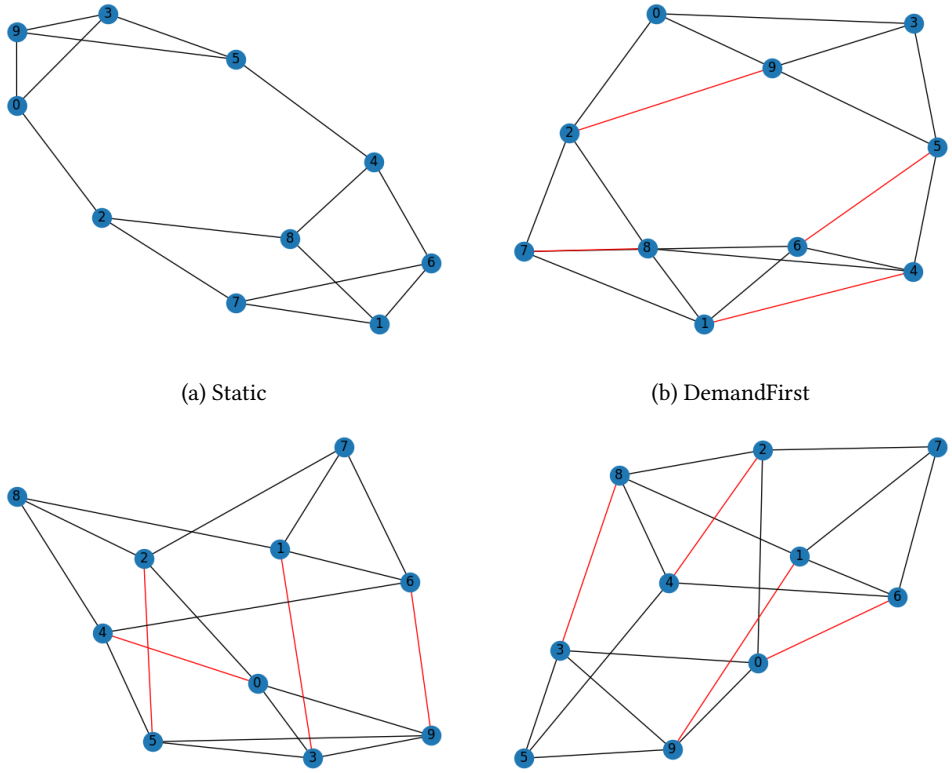
(a) Static

(b) DemandFirst

(c) LongestPathFirst

(d) DemandAwareLongestPathFirst

Abbildung A.2.: Zufällig generierte Topologie (a) mit 10 Knoten, sowie die anhand der Algorithmen für rekonfigurierbare Kanten (3.3.2) erweiterten Topologien (b), (c) und (d) des dritten Experiments.



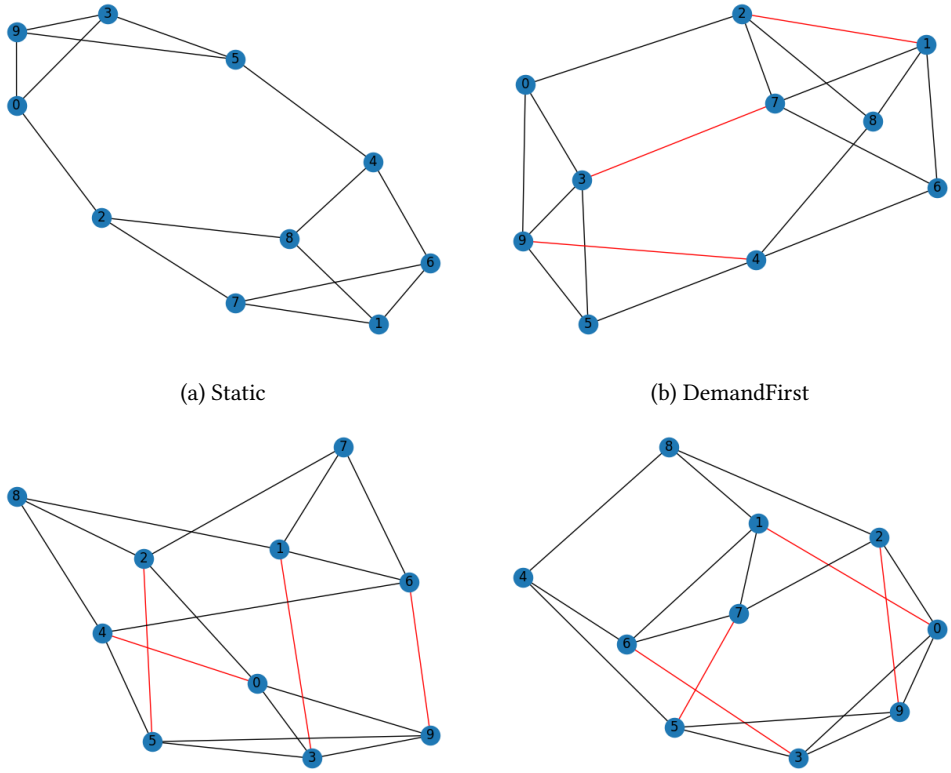
(a) Static

(b) DemandFirst

(c) LongestPathFirst

(d) DemandAwareLongestPathFirst

Abbildung A.3.: Zufällig generierte Topologie (a) mit 10 Knoten, sowie die anhand der Algorithmen für rekonfigurierbare Kanten (3.3.2) erweiterten Topologien (b), (c) und (d) des vierten Experiments.



(a) Static

(b) DemandFirst

(c) LongestPathFirst

(d) DemandAwareLongestPathFirst

Abbildung A.4.: Zufällig generierte Topologie (a) mit 10 Knoten, sowie die anhand der Algorithmen für rekonfigurierbare Kanten (3.3.2) erweiterten Topologien (b), (c) und (d) des fünften Experiments.

B. Ergänzung Programmcode

```
import time

from mininet.log import lg

import ipmininet
from ipmininet.cli import IPCLI
from ipmininet.ipnet import IPNet
from ipmininet.iptopo import IPTopo

class Topo(IPTopo):

    def build(self, *args, **kwargs):

            #Hosts
        h0 = self.addRouter('h0')
        h1 = self.addRouter('h1')
        h2 = self.addRouter('h2')
        h3 = self.addRouter('h3')
        h4 = self.addRouter('h4')
        h5 = self.addRouter('h5')
        h6 = self.addRouter('h6')
        h7 = self.addRouter('h7')
        h8 = self.addRouter('h8')
        h9 = self.addRouter('h9')

        #Links
        self.addLink(h0, h2, igp_metric=1000, bw=100, delay="5ms")
        self.addLink(h0, h3, igp_metric=999, bw=100, delay="5ms")
        self.addLink(h0, h9, igp_metric=998, bw=100, delay="5ms")

        self.addLink(h1, h6, igp_metric=997, bw=100, delay="5ms")
```

```
32          self.addLink(h1, h7, igp_metric=996, bw=100, delay="5ms")
33          self.addLink(h1, h8, igp_metric=995, bw=100, delay="5ms")
34
35          self.addLink(h2, h7, igp_metric=994, bw=100, delay="5ms")
36          self.addLink(h2, h8, igp_metric=993, bw=100, delay="5ms")
37
38          self.addLink(h3, h5, igp_metric=992, bw=100, delay="5ms")
39          self.addLink(h3, h9, igp_metric=991, bw=100, delay="5ms")
40
41          self.addLink(h4, h5, igp_metric=990, bw=100, delay="5ms")
42          self.addLink(h4, h6, igp_metric=989, bw=100, delay="5ms")
43          self.addLink(h4, h8, igp_metric=988, bw=100, delay="5ms")
44
45          self.addLink(h5, h9, igp_metric=987, bw=100, delay="5ms")
46
47          self.addLink(h6, h7, igp_metric=986, bw=100, delay="5ms")
48
49          # DemandAwareLongestPathFirst
50          #self.addLink(h1, h9, igp_metric=985, bw=100, delay="5ms")
51          #self.addLink(h3, h7, igp_metric=985, bw=100, delay="5ms")
52          #self.addLink(h0, h6, igp_metric=985, bw=100, delay="5ms")
53          #self.addLink(h5, h8, igp_metric=985, bw=100, delay="5ms")
54
55          # LongestPathFirst
56          #self.addLink(h1, h3, igp_metric=985, bw=100, delay="5ms")
57          #self.addLink(h0, h4, igp_metric=984, bw=100, delay="5ms")
58          #self.addLink(h2, h5, igp_metric=983, bw=100, delay="5ms")
59          #self.addLink(h6, h9, igp_metric=982, bw=100, delay="5ms")
60          #self.addLink(h7, h8, igp_metric=981, bw=100, delay="5ms")
61
62          # DemandFirst
63          self.addLink(h0, h8, igp_metric=980, bw=100, delay="5ms")
64          self.addLink(h1, h9, igp_metric=979, bw=100, delay="5ms")
65          self.addLink(h4, h7, igp_metric=978, bw=100, delay="5ms")
66
67          super(Topo, self).build(*args, **kwargs)
68
69  def perfTest():
70
```

```
71      print("*** Waiting for network to start")
72      for i in range(180,0,-1):
73          print(f"{i} ", end="\r", flush=True)
74          time.sleep(1)
75
76      print("*** Testing bandwidth between hosts")
77
78      h0 = net.routers[0]
79      h1 = net.routers[1]
80      h2 = net.routers[2]
81      h3 = net.routers[3]
82      h4 = net.routers[4]
83      h5 = net.routers[5]
84      h6 = net.routers[6]
85      h7 = net.routers[7]
86      h8 = net.routers[8]
87      h9 = net.routers[9]
88
89      h1.cmd('iperf3 -s -p 5009 &')
90      h0.cmd('iperf3 -s -p 5006 &')
91      h7.cmd('iperf3 -s -p 5004 &')
92      h2.cmd('iperf3 -s -p 5009 &')
93      h2.cmd('iperf3 -s -p 5008 &')
94      h9.cmd('iperf3 -s -p 5001 &')
95      h4.cmd('iperf3 -s -p 5005 &')
96      h5.cmd('iperf3 -s -p 5008 &')
97      h6.cmd('iperf3 -s -p 5001 &')
98      h0.cmd('iperf3 -s -p 5008 &')
99      h7.cmd('iperf3 -s -p 5004 &')
100     h8.cmd('iperf3 -s -p 5000 &')
101     h3.cmd('iperf3 -s -p 5007 &')
102     h1.cmd('iperf3 -s -p 5009 &')
103     h7.cmd('iperf3 -s -p 5000 &')
104     h0.cmd('iperf3 -s -p 5007 &')
105     h4.cmd('iperf3 -s -p 5003 &')
106     h5.cmd('iperf3 -s -p 5007 &')
107     h1.cmd('iperf3 -s -p 5003 &')
108     h6.cmd('iperf3 -s -p 5000 &')
109
```

```
110     h9.cmd('iperf3 -c h1 -p 5009 -n 63761K -J --logfile
            experiments/results/0-h9-h1-1.json &&'
111           'iperf3 -c h2 -p 5009 -n 209025K -J --logfile experiments/results/0-h9-h2.json
                  &&'
112           'iperf3 -c h1 -p 5009 -n 367893K -J --logfile
                  experiments/results/0-h9-h1-2.json &')
113
114     h7.cmd('iperf3 -c h3 -p 5007 -n 8342K -J --logfile experiments/results/0-h7-h3.json
            &&'
115           'iperf3 -c h5 -p 5007 -n 47340K -J --logfile experiments/results/0-h7-h5.json
                  &&'
116           'iperf3 -c h0 -p 5007 -n 205313K -J --logfile experiments/results/0-h7-h0.json
                  &')
117
118     h6.cmd('iperf3 -c h0 -p 5006 -n 522374K -J --logfile experiments/results/0-h6-h0.json
            &')
119
120     h4.cmd('iperf3 -c h7 -p 5004 -n 155699K -J --logfile
            experiments/results/0-h4-h7-2.json &&'
121           'iperf3 -c h7 -p 5004 -n 346907K -J --logfile
                  experiments/results/0-h4-h7-1.json &')
122
123     h0.cmd('iperf3 -c h7 -p 5000 -n 120804K -J --logfile experiments/results/0-h0-h7.json
            &&'
124           'iperf3 -c h6 -p 5000 -n 464581K -J --logfile experiments/results/0-h0-h6.json
                  &&'
125           'iperf3 -c h8 -p 5000 -n 724249K -J --logfile experiments/results/0-h0-h8.json
                  &')
126
127     h3.cmd('iperf3 -c h4 -p 5003 -n 64159K -J --logfile experiments/results/0-h3-h4.json
            &&'
128           'iperf3 -c h1 -p 5003 -n 206732K -J --logfile experiments/results/0-h3-h1.json
                  &')
129
130     h8.cmd('iperf3 -c h0 -p 5008 -n 40866K -J --logfile experiments/results/0-h8-h0.json
            &&'
131           'iperf3 -c h2 -p 5008 -n 80609K -J --logfile experiments/results/0-h8-h2.json
                  &&'
```

```
132           'iperf3 -c h5 -p 5008 -n 200853K -J --logfile experiments/results/0-h8-h5.json
                  &')
133
134     h1.cmd('iperf3 -c h6 -p 5001 -n 95747K -J --logfile experiments/results/0-h1-h6.json
            &&'
135           'iperf3 -c h9 -p 5001 -n 177736K -J --logfile experiments/results/0-h1-h9.json
                  &')
136
137     h5.cmd('iperf3 -c h4 -p 5005 -n 950697K -J --logfile experiments/results/0-h5-h4.json
            &')
138
139     for i in range(300,0,-1):
140         print(f"{i} ", end="\r", flush=True)
141         time.sleep(1)
142
143
144 ipmininet.DEBUG_FLAG = True
145 lg.setLogLevel("info")
146
147 # Start network
148 net = IPNet(topo=Topo(), use_v6=False)
149 net.start()
150 perfTest()
151 #IPCLI(net)
152 net.stop()
```

Programmcode B.1: Vollständiger Programmcode in Python einer Experimentvariante des ersten Experiments mit *SJF* und *DemandFirst*.

C. Ergänzung Workloads


```
[
  {
    "src": 9,
    "dst": 1,
    "size": 65291367,
    "start_time": 0
  },
  {
    "src": 7,
    "dst": 3,
    "size": 8542511,
    "start_time": 0
  },
  {
    "src": 9,
    "dst": 1,
    "size": 376722526,
    "start_time": 0
  },
  {
    "src": 6,
    "dst": 0,
    "size": 534911281,
    "start_time": 0
  },
  {
    "src": 4,
    "dst": 7,
    "size": 355232935,
    "start_time": 0
  },
```

```
{
  "src": 9,
  "dst": 2,
  "size": 214042533,
  "start_time": 0
},
{
  "src": 0,
  "dst": 7,
  "size": 123704110,
  "start_time": 0
},
{
  "src": 7,
  "dst": 0,
  "size": 210240840,
  "start_time": 0
},
{
  "src": 3,
  "dst": 4,
  "size": 65699680,
  "start_time": 0
},
{
  "src": 7,
  "dst": 5,
  "size": 48476602,
  "start_time": 0
},
{
  "src": 8,
  "dst": 2,
  "size": 82543711,
  "start_time": 0
},
{
  "src": 1,
  "dst": 9,
```

```
    "size": 182002504,
    "start_time": 0
  },
  {
    "src": 5,
    "dst": 4,
    "size": 973514499,
    "start_time": 0
  },
  {
    "src": 8,
    "dst": 5,
    "size": 205673709,
    "start_time": 0
  },
  {
    "src": 1,
    "dst": 6,
    "size": 98045742,
    "start_time": 0
  },
  {
    "src": 8,
    "dst": 0,
    "size": 41846930,
    "start_time": 0
  },
  {
    "src": 4,
    "dst": 7,
    "size": 159436120,
    "start_time": 0
  },
  {
    "src": 3,
    "dst": 1,
    "size": 211693811,
    "start_time": 0
  },
```

```
110   {
111     "src": 0,
112     "dst": 8,
113     "size": 741631780,
114     "start_time": 0
115   },
116   {
117     "src": 0,
118     "dst": 6,
119     "size": 475731258,
120     "start_time": 0
121   }
122 ]
```

Programmcode C.1: Zufällig generierter Traffic als JSON bestehend aus einzelnen Bulk-Transfers des ersten Experiments.

Eidesstattliche Versicherung

(Affidavit)

Huhn, Julian — 206598

Name, Vorname (surname, first name) — Matrikelnummer (student ID number)

☒ Bachelorarbeit (Bachelor's thesis) ☐ Masterarbeit (Master's thesis)

Titel (Title)

Optimierung rekonfigurierbarer Netzwerke für Bulk-Transfers mit gaengigen Scheduling-Algorithmen in Mininet

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Dortmund, 22.06.2023 — Julian Huhn

Ort, Datum (place, date) — Unterschrift (signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Dortmund, 22.06.2023 — Julian Huhn

Ort, Datum (place, date) — Unterschrift (signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**