Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Entwurf und Implementierung eines XDP-basierten Paket-Scheduling-Rahmenwerks zur Echtzeitkommunikation

Marcel Hafner

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Christian Becker
Betreuer/in:	Dr. rer. nat. Frank Dürr, Robin Laidig
Beginn am:	5. Oktober 2022
Beendet am:	5. April 2023

Kurzfassung

Die rasante Entwicklung von Netzwerktechnologien und Anwendungen hat zu immer höheren Anforderungen an Netzwerkleistung und -effizienz geführt. Der Linux eXpress Data Path (XDP) hat sich als eine leistungsstarke und flexible Technologie zur Verbesserung der Netzwerk-Performance im Linux-Umfeld etabliert. Die extended Berkeley Packet Filter (eBPF)-Technologie ermöglicht die Verwendung von XDP in der User-Space-Programmierung.

Diese Arbeit untersucht einen alternativen Ansatz zur Implementierung von Echtzeit-Scheduling-Mechanismen in einer User-Space-Anwendung. Ziel ist es, die Anwendbarkeit und Leistungsfähigkeit von AF_XDP (Application Flow XDP) zu untersuchen und ein generisches Software-Framework zu entwickeln, das verschiedene Scheduling-Verfahren implementieren und testen kann. Die Evaluierung der Leistung von AF_XDP wird im Hinblick auf Genauigkeit und Durchsatz durchgeführt. Der Completion-Ring von AF_XDP möglicht das sichere Versenden von Paketen, indem die Pakete nach dem Versenden in diesem automatisch geschrieben werden. Vorteil von diesem Ring ist das korrekte und zuverlässige Versenden von Paketen, jedoch beeinträchtigt dies den Durchsatz massiv.

Inhaltsverzeichnis

1 Einleitung **1**

2 Technischer Hintergrund **3**
2.1 Paketverarbeitung in einem Switch . . . 3
2.2 Paket-Scheduling . . . 5
2.3 Scheduling-Verfahren auf verschiedenen Schichten . . . 7
2.4 Ethernet Frames . . . 10
2.5 extended Berkeley Packet Filters (eBPF) . . . 12
2.6 eXpress Data Path . . . 13

3 Verwandte Arbeiten **17**
3.1 Netmap . . . 17
3.2 Data Plane Development Kit . . . 18
3.3 Software-Switch . . . 19

4 Systemmodell und Problemstellung **21**
4.1 Systemmodell . . . 21
4.2 Problemstellung . . . 21

5 Entwurf **23**
5.1 Systemarchitektur und Architektur des Software-Switches . . . 23
5.2 Endwurfentscheidungen . . . 23
5.3 Network-Switch Buffer Klasse . . . 25
5.4 Port Klasse . . . 27
5.5 Forward Klasse . . . 28
5.6 Classifier Klasse . . . 29
5.7 Packet Klasse . . . 30
5.8 Package Parsing Klasse . . . 31
5.9 Scheduler Klasse . . . 32
5.10 Thread Klasse . . . 36
5.11 Network-Switch Klasse . . . 38

6 Implementierung **43**
6.1 Aufbau des Testsystems . . . 43
6.2 Konfiguration des Testsystems . . . 43

7 Auswertung **47**
7.1 Durchsatz . . . 47
7.2 Korrektheit . . . 54

8 Zusammenfassung und Ausblick **59**

Literaturverzeichnis **61**

Abbildungsverzeichnis

2.1 Datenfluss des Eingangsportes 4
2.2 Datenfluss des Ausgangsportes 5
2.3 TSN_Queues 8
2.4 XDPFunktionsweise 14
2.5 XDP Ring UMEM 15

5.1 Software-Switch 24
5.2 Pufferpool Puffercache 26
5.3 Ethernet Klassifizierung 32

6.1 Aufbau 44

7.1 Packets per second (PPS) Sender und Empfänger Batching 49
7.2 Pakete Sender und Empfänger Batching 49
7.3 PPS Switch Batching 50
7.4 Pakete Switch Batching 50
7.5 PPS Sender und Empfänger mit Stop and Wait 51
7.6 Pakete Sender und Empfänger mit Warten 52
7.7 PPS Switch mit Warten 52
7.8 Pakete Switch mit Warten 53
7.9 Fehleranzahl beim Scheduling 57
7.10 Empfangene Pakete beim Scheduling 58

Tabellenverzeichnis

2.1 Ethernet II 11
2.2 Ethernet 802.3 11
2.3 Ethernet 802.3 Subnetwork Access Protocol (SNAP) 11
2.4 Ethernet II tagged 11

7.1 Durchschnitt und Empfänger der PPS Batching 51
7.2 Durchschnitt und Median der PPS mit Stop and Wait 53

Abkürzungsverzeichnis

API Application Programming Interface. 19

BCC BPF Compiler Collection. 12

BPF Berkeley Packet Filter. 12

CL Completion. 25

CL-Ring Completion-Ring. 14

CNAME Canonical Name. 10

CPU Central Processing Unit. 5

DiffServ Differentiated Services. 9

DPDK Data Plane Development Kit. 18

DSCP Differentiated Services Code Point. 9

E/A Eingang und Ausgang. 12

eBPF extended Berkeley Packet Filter. 13

FIFO First-In First-Out. 6

IEEE Institute of Electrical and Electronics Engineers. 10

IntServ Integrated Services. 9

JIT Just-In-Time Compiler. 13

MAC Media Access Control. 3

MPPS Million packets per second. 15

NIC Netzwerkkarte. 17

OSV Open vSwitch. 19

PCP Feld Priority Code Point. 7

PHY Physical Layer Receiver. 3

PPS Packets per second. vii

PTP Precision Time Protocol. 55

QoS Quality of Service. 7

RSVP Resource Reservation Protocol. 9

RTCP RTP Control Protocol. 10

RTP Real-Time Transport Protocol. 10

RX Receive. 14

SNAP Subnetwork Access Protocol. ix

SSH Secure Shell. 19

TAP Test Access Points. 43

TAS Time-Aware Shaper. 7

TOS Type of Service. 9

TSN Time-Sensitive Networking. 7

TX Transmit. 14

UMEM Datenpuffer im Speicher. 14

VALE Virtual Local Ethernet. 19

VLAN Virtual Local Area Network. 3

VM virtuelle Maschine. 12, 19

XDP eXpress Data Path. 1

1 Einleitung

In den letzten Jahren hat die rasante Entwicklung von Netzwerktechnologien und Anwendungen zu immer höheren Anforderungen an Netzwerk-Leistung und -Effizienz geführt. Moderne Multi-Core-Architekturen ermöglichen es, durch höhere Leistung die Paketverarbeitung in Software zu implementieren. Diese, auch unter dem Stichwort Software-Defined Networking (SDN) zusammengefassten Lösungen ermöglichen es, neue Funktionalität schnell und einfach hinzuzufügen, was große Flexibilität schafft [Sta15].
In diesem Zusammenhang hat sich der Linux eXpress Data Path (XDP) als eine leistungsstarke und flexible Technologie zur Verbesserung der Netzwerk-Performance im Linux-Umfeld etabliert. Die Kernidee von XDP ist es, eine schnelle, effiziente und flexible Möglichkeit zur Verarbeitung von Netzwerkdatenpaketen auf dem Kernel-Level bereitzustellen. Dies wird erreicht, indem die Pakete direkt in der Netzwerkkarte bearbeitet werden, bevor sie den Kernel erreichen [VCP+20].
Extended Berkeley Packet Filter (eBPF) ist eine Technologie in Linux die es ermöglicht, benutzerdefinierte Funktionen in Form von eBPF-Programmen in den Kernel einzufügen und auf Ereignissen wie Netzwerkverkehr, Systemaufrufe und Kernel-Events zu reagieren. eBPF-Programme werden in einer sicheren Umgebung ausgeführt und haben nur begrenzten Zugriff auf Ressourcen, was die Sicherheit und Stabilität des Systems gewährleistet [VCP+20].
Die eBPF ermöglichen die Verwendung von XDP in der User-Space-Programmierung [VCP+20]. Dennoch stellen zeitkritische Anwendungen weiterhin eine Herausforderung dar, insbesondere wenn es um die Übertragung von großen Datenmengen in Echtzeit, d.h. mit gegebenen Schranken bezüglich der Verzögerung geht. Echtzeit-Scheduling-Mechanismen werden benötigt um sicherzustellen, dass kritische und zeitkritische Aufgaben in Echtzeit ausgeführt werden können. Solche Aufgaben müssen innerhalb einer bestimmten Zeitspanne abgeschlossen werden, um eine ordnungsgemäße Funktion des Systems zu gewährleisten. Ohne Echtzeit-Scheduling-Mechanismen besteht das Risiko, dass kritische Aufgaben nicht rechtzeitig ausgeführt werden und dadurch das System fehlerhaft oder unbrauchbar wird. Ein Beispiel für zeitkritische Aufgaben ist die Verarbeitung von Audio- und Videodaten in der Unterhaltungsindustrie [Zöb20].

Diese Arbeit untersucht einen alternativen Ansatz zur Implementierung von Echtzeit-Scheduling-Mechanismen in einer User-Space-Anwendung. Im Rahmen dieser Aufgabenstellung wird das Ziel verfolgt, die Anwendbarkeit und Leistungsfähigkeit von AF_XDP (Application Flow XDP) zu untersuchen. Um die Anwendung von XDP-Sockets in verschiedenen Anwendungsbereichen zu erleichtern, ist die Entwicklung eines generischen Software-Frameworks von entscheidender Bedeutung. Ein solches Framework ermöglicht es Entwickelnden, schnell und einfach XDP-Sockets in verschiedenen Anwendungen zu implementieren, anzupassen und zu testen. Das Framework unterstützt verschiedene Scheduling-Algorithmen, um den Datenverkehr in Echtzeit zu priorisieren und zu steuern. Ein wichtiger Aspekt bei der Entwicklung von Netzwerkanwendungen ist die Leistungsfähigkeit und Effizienz des Systems. Ein Problem bei solchen Systemen, ist das Scheduling von Paketen in Echtzeit. Diese Systeme erfordern eine hohe Zuverlässigkeit und Zeitgenauigkeit, was bei der Entwicklung berücksichtigt werden muss. Ein weiteres Problem beim Scheduling von

Paketen in Echtzeit ist das Treffen der richtigen Entscheidung unmittelbar vor dem Weiterleiten des Pakets. Hier kommen diverse Puffer zwischen der Anwendung und der Netzwerkschnittstelle ins Spiel, die die Scheduling-Entscheidung erschweren. XDP soll hier helfen, indem es direkt nach dem Senden des vorausgehenden Pakets die Scheduling-Entscheidung für das nächste Paket trifft. Das Framework soll generisch sein und verschiedene Scheduling-Verfahren implementieren und testen können. Beispiele dafür sind zeitgesteuertes Scheduling gemäß dem IEEE 802.3Qbv-Standard [iee11] oder prioritätengesteuerte Scheduling-Verfahren [iee11].

Die Evaluierung der Leistung von AF_XDP wird im Hinblick auf Genauigkeit (Korrektheit) und Präzision der Planung und Durchsatz durchgeführt. Es werden hierfür vier vordefinierte Implementierungsvarianten von Scheduling-Algorithmen geprüft. Es gibt zwei Implementierungen, die beide auf der XDP-Technologie basieren. Die erste nutzt einen Stop and Wait-Ansatz, dabei werden alle vier Ringe verwendet. Die zweite Implementierung namens Batching setzt auf die Schnelligkeit von XDP und versucht, die Anzahl der Kontextwechsel auf ein Minimum zu reduzieren. Hierbei wird die Implementierung von eBPF und des implementierten Rahmens berücksichtigt, um eine möglichst realistische Verhalten zu ermöglichen.
Die abschließend durchgeführte Evaluierung zeigt, dass im Vergleich zum Batching-Ansatz der Stop-and-Wait-Ansatz zu einer besseren Scheduling-Entscheidung führt, durch ein spätes Treffen der Entscheidung unmittelbar vor dem Senden eines Pakets. Dadurch wird sichergestellt, dass die nächste Paketübertragung korrekt erfolgt. Jedoch verringert der Stop-and-Wait-Ansatz den Durchsatz des Switches signifikant im Vergleich zum Batching-Ansatz. Der erzielte Durchsatz von ca. 4,704 Mbps liegt weit entfernt von den Durchsatzraten moderner Gigabit- oder sogar Multi-Gigabit-Ethernet-Technologien.

Die Arbeit besteht aus insgesamt acht Kapiteln, die alle Aspekte des Projekts abdecken. Das zweite Kapitel erläutert den technischen Hintergrund und wichtige Begriffe, die in dieser Arbeit verwendet werden. In Kapitel drei werden verwandte Arbeiten vorgestellt, die in enger Verbindung zu diesem Projekt stehen. Das vierte Kapitel beschreibt das Systemmodell und die Problemstellung, die dieser Arbeit zugrunde liegen. Kapitel fünf ist dem Entwurf und Aufbau des Softwareswitches gewidmet, der das Ziel dieser Arbeit ist. In Kapitel sechs wird die Implementierung des Switches beschrieben, einschließlich eines Beispielaufbaus. Kapitel sieben ist der Auswertung der Arbeit gewidmet und stellt die Bewertungsergebnisse des Switches vor, die im Rahmen der Arbeit erzielt wurden. Das letzte Kapitel bündelt die Erkenntnisse und stellt Perspektiven vor, wie sich die vorliegende Arbeit fortsetzen lässt.

2 Technischer Hintergrund

Zu Beginn der Arbeit werden zunächst die technischen Grundlagen erläutert. Dafür wird im ersten Abschnitt dieses Kapitels die Paketverarbeitung in einem Switch erläutert. Anschließend wird sich mit den Scheduling-Methoden befasst, die später in dem Software-Framework implementiert werden sollen. In Abschnitt 2.3 werden unterschiedliche Scheduling-Verfahren auf den verschiedenen Layern des OSI-Modells aufgezeigt. Daran anknüpfend werden die für den weiteren Verlauf der Arbeit notwendigen technischen Grundlagen der Ethernet-Technologie vermittelt. Abschließend wird auf die eBPF näher eingegangen, sowie die XDP-Technologie näher erläutert.

2.1 Paketverarbeitung in einem Switch

In Abbildung 2.1 wird der Ablauf von einem Paket demonstriert, welches an einem Port eines Switch ankommt und wie dieses weiterverarbeitet wird. Ankommende Pakete werden zunächst gepuffert, bis diese vollständig angekommen sind (Physical Layer Receiver (PHY)). Nach dem vollständigen Empfangen durch das System wird die Media Access Control (MAC) der Rahmen auf seine Korrekt - und Gültigkeit überprüft [SE08].
Der Link-Aggregation-Kollektor implementiert die Frame-Sammelfunktion, indem er Frames, die von jeder zugrunde liegenden physischen Schnittstelle empfangen wurden, sammelt und der Klassifizierungsmaschine als einen einzigen, kombinierten Datenstrom präsentiert, wenn mehrere physische Schnittstellen zu einer einzigen logischen Schnittstelle zusammengefasst werden [SE08].
Die Klassifizierungsmaschine ist ein komplexes und leistungsintensives Modul eines Switches, das verwendet wird, um empfangene Frames zu analysieren und zu klassifizieren, basierend auf verschiedenen Kriterien und administrativen Richtlinien wie z.B. lokale Versenkung von reservierten Multicast-Adressen und Virtual Local Area Network (VLAN)-Eingangsregeln. Je nach Komplexität und den angewandten Regeln wird jedem Frame eine VLAN-Identifikationsnummer zugewiesen [SE08].
VLAN steht für die Abkürzung Virtual Local Area Network und in den Ethernet-Headers ist dafür ein zwei Byte großes Tag vorgesehen. Das 12 Bit lange VLAN-Tag wird zur Segmentierung des Datenverkehrs benötigt und ist im Tag-Feld enthalten. Darüber hinaus können im Feld des Ethernet-Headers drei Bit zur Priorisierung verwendet werden, um bis zu acht Prioritätsstufen zuzuordnen und Dienstklassen bzw. Class of Service zu definieren. Der VLAN-Tag sorgt dafür, dass nur Computer mit dem gleichen VLAN-Tag das Paket empfangen können. Der Einsatz von VLANs ist dann gegeben, wenn mehrere getrennte Netze benötigt werden und auf dem selben Switches laufen sollen [TW13].
Das VLAN Filter-Modul implementiert die beiden VLAN-Eingangsfilter. Der “Acceptable Frame Filter“ prüft, ob ein Frame explizit mit einem VLAN-Tag versehen ist, und der VLAN Ingress Filter akzeptiert oder verwirft Frames auf Grundlage der Portmitgliedschaft. Diese Filter können in der Lookup Engine integriert werden [SE08].

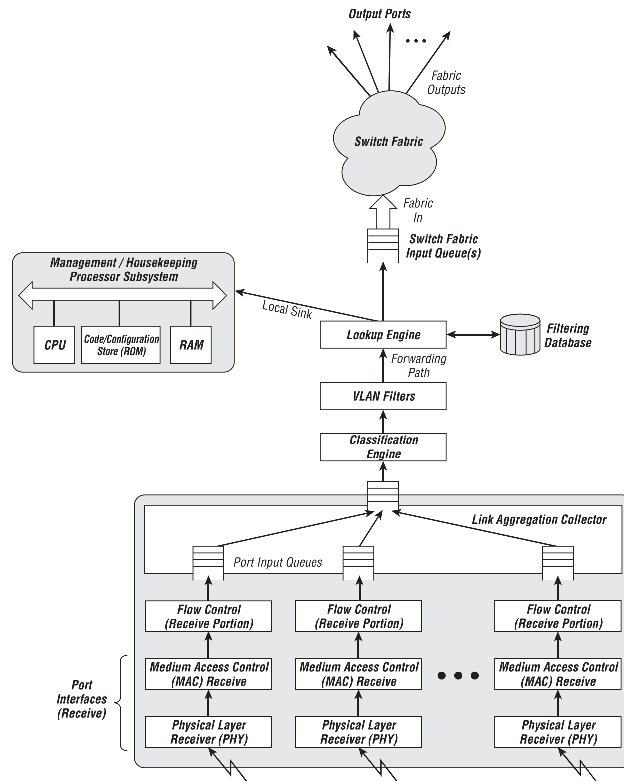

Abbildung 2.1: Datenfluss des Eingangsportes [SE08]

Die Lookup Engine ist das Herzstück des Switch-Weiterleitungsprozesses der entscheidet, welche Ausgangsports für eine Weiterverarbeitung und mögliche Übertragung eines Rahmens verwendet werden. Hierzu wird eine Tabellensuche gegen die Filterdatenbank durchgeführt, welche aktuelle Zuordnungen von Zieladressen und VLANs zu den Anschlüssen enthält. Frames werden entweder einem einzigen Ausgangsport für unicast- oder einem oder mehreren Ausgangsports für multicast-Ziele zugewiesen [SE08].
Die Aufgabe von Switch-Fabrics ist es zwischen den Dateneingangs- und -ausgangspfaden sitzen und Datenrahmen zwischen den Ports zu übertragen [SE08].

Der Ablauf wie ein Paket aus dem Switch versendet werden soll ist in Abbildung 2.2 gezeigt. Um einen Frame an einen bestimmten Ausgangs-Port zu übertragen, müssen zwei VLAN-bezogene Qualifikationstests durchgeführt werden um zu entscheiden, ob der Port Mitglied des VLANs ist und ob der Frame getaggt oder ungetaggt übertragen wird. Eine einfache Ausgangsfiltertabelle wird verwendet, um die oder ungetaggt übertragen Egress-Regeln und den Egress-Filter gleichzeitig anzuwenden und unerwünschte Übertragungen von Frames zu vermeiden. Dies wird in den "Output Filters" entschieden[SE08].
Die Pakete werden im nächsten Schritt nach verschiedenen Qualifikationstests in eine der Ausgangswarteschlangen des Ports platziert, die verschiedene Dienstgüten implementieren. Die Frames werden nach einer Prioritätsrichtlinie "de-queued" durch einen Scheduler und von diesem für die Übertragung in der gewünschten Reihenfolge auf den Ausgangsport geliefert. Die Pakete werden dann in dieser Reihenfolge von dem Port übertragen [SE08].

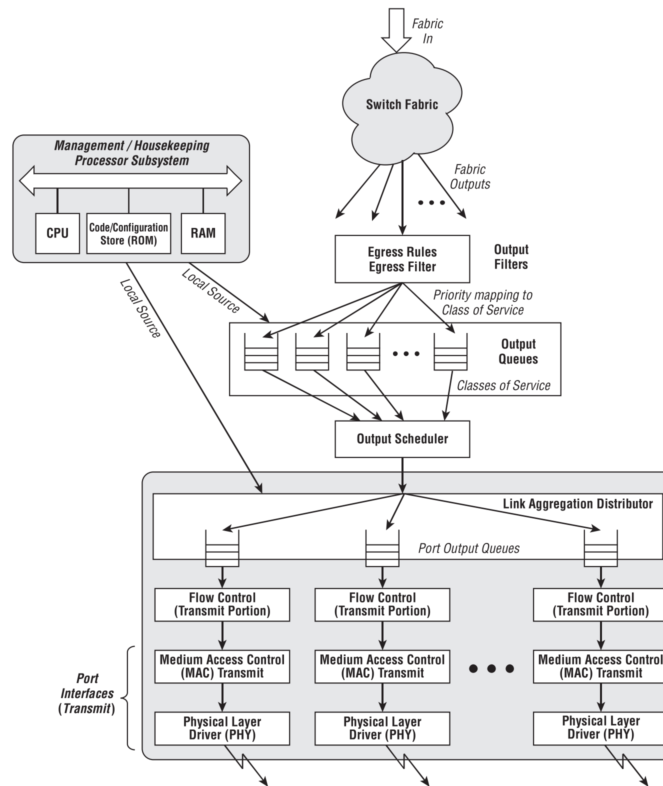

Abbildung 2.2: Datenfluss des Ausgangsportes [SE08]

2.2 Paket-Scheduling

Da die Bandbreite von Netzen beschränkt ist, muss diese auf die verschiedenen Datenströme aufgeteilt werden. Ein Paket-Scheduler ist ein Algorithmus, welcher die zeitliche Abfolge von gepufferten Paketen bestimmt und versendet. Sie werden auch Netzwerk-Scheduler genannt. Unterschiedliche Scheduler verfolgen verschiedene Ziele, z.B. die Priorisierung von wichtigen Paketen wie Echtzeitverkehr oder die faire Aufteilung der Bandbreite auf verschiedene Ströme. Wenn die Warteschlange des Puffers voll ist, kann es zu Wartezeiten für Pakete kommen, da sie darauf warten müssen, in die Warteschlange aufgenommen zu werden oder darauf, dass ein anderes Paket aus der Warteschlange ausgewählt wird. Diese Wartezeiten werden als Queuing-Delay bezeichnet und können je nach Auslastung des Netzwerks und der Geschwindigkeit des Scheduling-Verfahrens variieren. Ein langes Queuing-Delay kann sich auf die Netzwerklatenz auswirken und die Leistung des Netzwerks beeinträchtigen. Daher ist es wichtig, das Scheduling-Verfahren sorgfältig zu wählen und zu konfigurieren, um das Queuing-Delay zu minimieren und eine optimale Leistung zu erzielen. Aus diesen Grund können bestimmte Datenflüsse Bandbreite für sich reservieren. Eine weitere Ressource, welche endlich bei Software-Switches und Software-Router ist, ist die Central Processing Unit (CPU), welche nur eine bestimmte Anzahl an Paketen pro Sekunde verarbeiten kann. Bei Hardware-Switches und Hardware-Router ist die Hauptaufgabe die Weiterleitung von Paketen, sodass spezielle Hardware mit sehr kurzer Verzögerung hierfür bereitgestellt wird. CPU werden für die interne Bearbeitung von Pakete benötigt [TW13].
Die CPU ist der Hauptprozessor eines Computers und fungiert als "Gehirn"des Systems. Sie

ist für die Ausführung von Programmen, die Verarbeitung von Daten und die Steuerung der Systemressourcen verantwortlich. Die CPU besteht aus einer Vielzahl von Schaltkreisen und wird durch einen Taktgeber gesteuert, der den Rhythmus der Ausführung vorgibt [PH16].

2.2.1 FIFO

Der einfachste Paket-Scheduling-Algorithmus ist der First-In First-Out (FIFO)-Algorithmus. Dieser arbeitet nach dem Prinzip, wer zuerst kommt, wird als erstes wieder weiterverarbeitet bzw. weitergeleitet. Sollte der Puffer voll sein, verwirft dieser die neu ankommenden Rahmen. Dieses Verfahren wird als "Tail Drop" bezeichnet. In der Praxis gibt es andere Verfahren, welche effizienter mit dem Problem umgehen können. Der Algorithmus ist zwar sehr einfach zu implementieren, eignet sich in der Praxis für viele Fälle jedoch nicht. Ein Grund dafür ist, dass zwischen verschiedenen Datenflüssen nicht unterschieden werden kann, sodass nur eine einzige Datenquelle viel Bandbreite für sich reservieren kann. Sollten die Pakete einen Platz im Puffer bekommen, verzögert sich die Verarbeitung bis alle anderen großen Pakete verarbeitet wurden. Durch große Rahmen kann der Puffer jedoch schnell gefüllt werden, sodass viele ankommende Rahmen verworfen werden [TW13].

2.2.2 Weighted Fair Queuing

Weighted-Fair-Queuing ist ein Fairer-Paket-Scheduling-Algorithmus, welcher für jeden Datentrom eine eigene Warteschlange hat und fair ist. Im Round-Robin-Verfahren darf jede Warteschlange Bytes versenden. Innerhalb der Warteschlange wird meist das FIFO-Prinzip angewendet.
In machen Situationen ist es vorteilhaft, wenn mehr Bandbreite für Prozesse verfügbar sind. Streams brauchen zum Beispiel im Gegensatz zu einem Dateiserver schnellere Reaktionszeiten, was mehr Bandbreite bedeutet. Gewichtete Warteschlangen können dieses Problem lösen. Das Gewicht einer Warteschlange bestimmt die Anzahl an Bytes, die pro Runde versendet werden können. Es kann jeder Warteschlange, also eine individuelle Bandbreite zugeordnet werden. Sollte das Gewicht von allen Warteschlange gleich sein, werden alle Warteschlangen entsprechend auch gleich behandelt. Bei echtzeitfähigem Verkehr kann es Sinn ergeben, diesen in eine Warteschlange mit hohem Gewicht zuzuordnen, sodass diese schnell versendet werden [TW13].

2.2.3 Stict-Priority-Queuing

Ein Stict-Priority-Queuing-Algorithmus ist ein Algorithmus, welcher Warteschlangen mit hoher Priorität den Vorzug vor anderen Warteschlagen gibt. Verzögerungsempfindliche Pakete mit hoher Priorität wie Sprachverkehr mit minimaler Verzögerung werden schneller weitergeleitet. Es werden zuerst alle Pakete aus der Warteschlange mit der höchsten Priorität entnommen, bevor die nächste Warteschlange mit der zweithöchsten Priorität berücksichtigt wird. Wenn jedoch kontinuierlich Pakete mit hoher Priorität eintreffen, können die Pakete mit niedrigerer Priorität blockiert und vernachlässigt werden, da diese keine Chance haben, bedient zu werden, solange der Datenstrom mit höherer Priorität aktiv ist. Dies kann zu einem "Aushungern" von Datenströmen mit niedrigerer Priorität führen [iee11].

2.3 Scheduling-Verfahren auf verschiedenen Schichten

Das Software-Framework dieser Arbeit soll verschiedene Protokolle auf verschieden Layern unterstützen. Es werden aus diesem Grund jeweils ein Protokoll aus den Schichten zwei bis vier vorgestellt.

2.3.1 Layer 2: Time-Sensitive Networking

Time-Sensitive Networking (TSN) ist ein Scheduling-Verfahren, welches auf Schicht zwei des OSI-Modell arbeitet. Mit TSN soll es möglich sein, Netzwerkverkehr echtzeitfähig zu machen. Ethernet wurde ursprünglich nicht für Echtzeitkommunikation entwickelt und wurde erst durch verschiedene Erweiterungen und Verbesserungen echtzeitfähig gemacht. Vernetzte Regelungssysteme sind eine Art von Regelungssystemen, die aus mehreren Komponenten bestehen, die über ein Netzwerk miteinander verbunden sind. Die Komponenten können Sensoren, Aktoren, Regler und andere Geräte umfassen, die miteinander kommunizieren, um eine bestimmte Regelungsfunktion zu erfüllen. Diese Echtzeitfähigkeit soll der Industrie ermöglichen, dass Steuerungen nicht mehr an der Maschine stehen müssen, sondern in die Cloud auslagert werden können. Dies ist aber nur möglich, falls es ein Protokoll gibt, dass die Sensorwerte und die Aktorbefehle in Echtzeit austauschen kann. Es gibt bereits Echtzeit-Ethernet-Technologien, wie PROFINET [PRO23], die das möglich machen. Echtzeit und Nicht-Echtzeit Pakete können mit verschiedenen Prioritäten in einem Netz transportieren werden und dabei Garantien für den Echtzeitverkehr geben [DN16].

Im Folgenden wird sich besonders auf den IEEE 802.1Qbv-Standard (Time-Aware Shaper (TAS)) konzentriert. Das Feld Priority Code Point (PCP) ist ein 3-Bit-Feld im 802.1Q-Tag-Header-Feld des Ethernet-Frames, das zur Priorisierung des Datenverkehrs in VLAN-basierten Netzwerken verwendet wird. Das PCP-Feld kann Werte von 0 bis 7 annehmen und gibt die Priorität des Datenverkehrs an. Der PCP wird verwendet, um den Datenverkehr basierend auf seiner Priorität zu klassifizieren und zu steuern. Dies ermöglicht es, eine Quality of Service (QoS) für bestimmte Anwendungen oder Dienste bereitzustellen, indem der Datenverkehr je nach seiner Priorität priorisiert wird. Ein höherer PCP-Wert bedeutet, dass der Datenverkehr eine höhere Priorität hat. Wenn ein Switch oder Router ein Paket empfängt, das mit einem VLAN-Tag markiert ist, wird das PCP-Feld verwendet, um das Paket entsprechend seiner Priorität in der Warteschlange zu ordnen. Der Datenverkehr mit höherer Priorität wird zuerst übertragen, während der Datenverkehr mit niedrigerer Priorität warten muss. Die Quality of Service (QoS) bezieht sich auf die Fähigkeit eines Netzwerks, verschiedenen Arten von Datenverkehr zu priorisieren und zu steuern, um sicherzustellen, dass wichtiger Datenverkehr priorisiert und mit hoher Qualität bereitgestellt wird. QoS wird oft verwendet, um sicherzustellen, dass kritische Anwendungen wie Sprache, Video und Echtzeitdaten mit hoher Priorität behandelt werden, während weniger kritischer Datenverkehr wie E-Mail oder Dateiübertragungen mit niedrigerer Priorität behandelt werden können [iee11].

In einem Switch gibt es für jeden Port mehrere Warteschlangen. In einem TSN-Switch gibt es für jede Warteschlange ein Tor (“Gate“). Dieses Tor bestimmt, ob Pakete aus der Warteschlange entnommen werden können oder nicht. Dazu steuert ein Treiberprogramm, wann sich die Tore öffnen und schließen. Ein TSN-Switch hat mindestens eine Warteschlange für den echtzeitfähigen

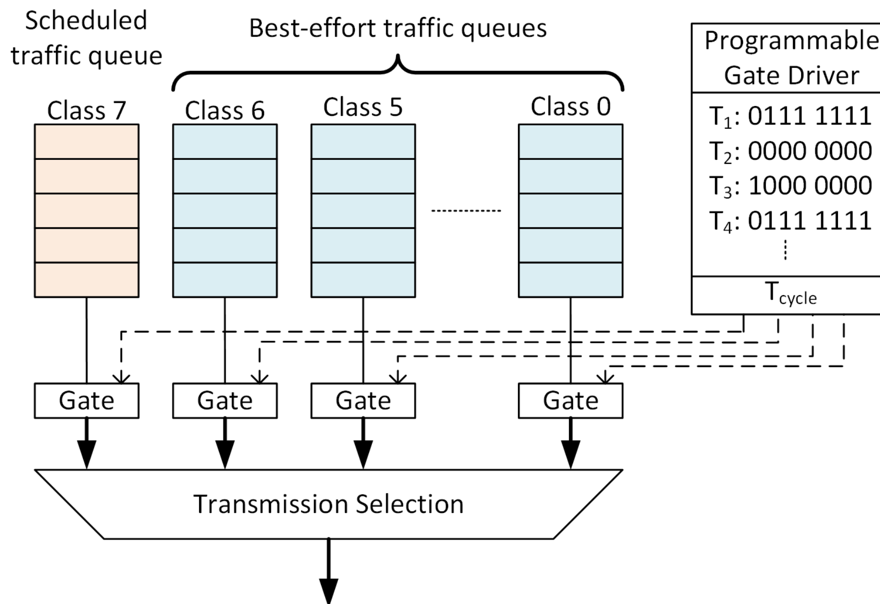

Abbildung 2.3: Architektur eines einzelnen Ports in einem TSN-Switch [DN16]

Netzwerkverkehr und mindestens eine andere Warteschlange für den Best-Effort-Verkehr[1]. Verschiedene Ströme mit verschiedenen Prioritäten können auch verschiedene Warteschlangen zugeordnet werden [DN16].
Eine Queue darf nur senden, wenn das entsprechende Gate offen ist. Wenn mehrere Gates offen sind, entscheidet der so genannte Transmission Selection Algorithmus, welche Queue senden darf. Ein Transmission-Selection-Algorithmus ist z.B. Strict Priority. In Abbildung 2.3 ist die Architektur eines einzelnen Ports in einem TSN-Switch dargestellt. Der Schedule wird aufgrund der Verkehrscharakteristika und -Anforderungen mit Hilfe von Planungsalgorithmen berechnet z.B. mit Hilfe von einem "No-Wait Packet"-Scheduling Ansatz. Der No-Wait Packet-Scheduling Ansatz ist eine Technik, die in der Computernetzwerktechnik verwendet wird, um die Latenzzeit der Paketübertragung zu reduzieren. Eine häufige Methode ist die Verwendung einer Prioritätswarteschlange für die Paketübertragung, bei der Pakete mit höherer Priorität sofort weitergeleitet werden [DN16].

Um Pakete innerhalb des Zeitfensters zu versenden, wird versucht alle TSN-Switch-Uhren miteinander auf die gleiche Uhrzeit zu synchronisiert [DN16].

[1] Best-Effort-Verkehr bedeutet, dass Netzwerkpakete so schnell wie möglich weitergeleitet werden, sofern Kapazitäten frei sind.

2.3.2 Layer 3: IntServ/DiffServ Protokoll

Integrated Services (IntServ) ist ein Verfahren um die QoS[2]-Eigenschaften für IP-Netze bereit zu stellen. Das Verfahren wird auf Schicht drei eingesetzt. Bei dem Verfahren werden Ressourcen explizit reserviert.

Es gibt drei Kategorien, welche durch IntServ identifiziert werden. Die erste Hauptklasse von IntServ ist der Best-Effort-Netzwerkverkehr, dieser hat keine QoS-Eigenschaften. Die nächste Klasse ist der garantierte Netzwerkverkehr (Guaranteed QoS), bei dem QoS-Eigenschaften zugesichert werden, beispielsweise wird ein Teil der Bandbreite für den Verkehr zugesichert. Die letzte Hauptklasse ist der Netzwerkverkehr mit kontrollierter Last ("Controlled Load"). Diese Klasse wird für Anwendungen verwendet, die eine garantierte Bandbreite und minimale Verzögerung benötigen. Sie ist auch als "best effort with admission control" bekannt, da der Netzwerkverkehr innerhalb der Kapazitätsgrenzen des Netzwerks gewährleistet wird, ohne dass eine vollständige Garantie für Bandbreite und Verzögerung besteht.

Für das Scheduling wird Weighted Fair Queuing eingesetzt. Die Klassifizierung erfolgt z.B. anhand der Stream-Adressen oder bei Differentiated Services (DiffServ) Type of Service Feld im IP-Header. Um die QoS-Eigenschaften einzuhalten wird das Resource Reservation Protocol (RSVP) benötigt. RSVP-Pakete werden entlang vom Sender zum Empfänger verschickt, um Ressourcen zu reservieren. Der Sender schickt eine "RSVP Path Message" an dem Empfänger. Die "RSVP Path Message" enthält eine detaillierte Beschreibung der gewünschten QoS-Parameter, die es dem Netzwerk ermöglichen, die erforderlichen Ressourcen bereitzustellen und sicherzustellen, dass der Datenstrom die gewünschte QoS erhält. Der Empfänger empfängt diese Nachricht und schickt eine "RESV Message" zurück an dem Sender, mit der RSPEC (R = Reservation) zu reservierende Bandbreite. Das maximale Queuing-Delay kann aufgrund der Parameter in "TSPEC" und "RSPEC" berechnet werden. Die zurückgesendete Nachricht zum Sender wird nun von jedem Knoten bearbeitet, welcher sich auf dem Pfad befindet. Kann ein einziger Knoten die Reservierung nicht tätigten, wird diese blockiert. Bei erfolgreicher Reservierung auf jedem Knoten müssen nun regelmäßig "PATH"- und "RESV" Messages gesendet werden, um die Reservierung aufrecht zu erhalten [MS99].

DiffServ wird zur Klassifizierung von IP-Paketen benutzt und arbeitet wie IntServ auf der dritten Schicht des OSI-Modells. Im Gegensatz zu IntServ werden bei DiffServ Ressourcen nicht explizit für Verbindungen reserviert, sondern der Verkehr in Klassen unterteilt und dann priorisiert. DiffServ hat im Vergleich zu IntServ keine Ende-zu-Ende-Signalisierungsmechanismus, um die QoS-Eigenschaften einzuhalten. Um die Klasse der IP-Pakete zu identifizieren gibt es im IP-Header ein Feld. Dieses Feld ist ein 8-Bit-Type of Service (TOS) Feld. Die ersten sechs Bits sind für Differentiated Services Code Point (DSCP) und die letzten zwei Bit sind unbenutzt. DSCP gibt also lediglich an, wie mit dem Paket weiter verfahren werden soll. Ein DiffServ-Netz benötigt drei Services: den Paketklassifizierer, ein Verkehrsprofil und den Verkehrskonditionäre. Der Paketklassifizierer identifiziert und klassifiziert Pakete entsprechend ihrer Priorität und den Anforderungen an die Übertragungsqualität. Der Verkehrsprofiler überwacht und misst die Netzwerkauslastung und den Datenverkehr, um ein Profil der Netzwerkbelastung zu erstellen. Der Verkehrskonditionierer gewährleistet, dass der Verkehr innerhalb der vom Netzwerk bereitgestellten Kapazität bleibt [MS99].

[2] Für die QoS gibt es in Rechnennetzen folgende Eigenschaften, welche dem Anwender ein Maß geben, wie gut die Verbindung ist: Latenzzeit, Paketverlustrate und Durchsatz. [iee11]

2.3.3 Layer 4: RTP/RTCP Protokoll

Real-Time Transport Protocol (RTP) ist ein Protokoll welches benutzt wird, um Daten in Echtzeit zu übertragen [Her20]. Im Gegensatz zu IntServ arbeitet es auf den Endsystemen und erfordert keine Unterstützung durch das Netzwerk. Live-Videos, interaktives Audio oder Simulationsdaten sind beispielsweise Anwendungsfälle von diesem Protokoll. Es unterstützt sowohl Unicast also auch Multicast-Anwendungen [SCFv03]. Es ist auf der Transportschicht verankert und erweitert das UDP-Protokoll. RTP kann im Gegensatz zu UDP Nachrichten erkennen, welche verloren gegangen sind. Die Pakete werden mit einer Sequenznummer gezählt und mit einem Zeitstempel versehen. Dieser gibt an, wann das Paket versendet wurde. Es kann außerdem Pakete wieder in die richtige Reihenfolge bringen. Die Erweiterungen werden im Header eines RTP-Paketes gespeichert [Her20]. RTP garantiert nicht, dass die Pakete rechtzeitig und korrekt beim Empfänger ankommen. Das Protokoll verlässt sich ganz auf die Schichten, die sich unter ihm befinden [SCFv03].

Neben RTP wird auch das RTP Control Protocol (RTCP) benötigt, welches die Informationen zur Verwaltung einer Sitzung überträgt [Her20]. RTCP ist ein Netzwerkprotokoll, um in einer Sitzung Kontrollpakete an alle Teilnehmer zu verteilen. Hauptaufgabe vom RTCP ist es Information über die Qualität der aktuellen Dienstgüte zu geben, mit der die Anwendung die Kommunikation anpassen kann. Eine weitere Aufgabe ist es, mittels seines Canonical Name (CNAME) die Teilnehmer einer Sitzung zu identifizieren. Die Anwendung muss sich aufgrund der durch RTCP ermittelten und kommunizierten Information anpassen, z.B. bedeutet das die Datenrate reduziert werden muss. Der Sender bekommt zusätzlich regelmäßig Kontrollnachrichten von den Empfängern mit Angaben zur Verbindungsqualität und der Anzahl der verlorengegangen Pakete. So können Fehler in der Verteilung vom Sender zum Empfänger diagnostiziert werden [SCFv03].

Für eine Anwendung von RTP und RTCP benötigt jeder Dienst einen eignen Port um Daten in eine Richtung auszutauschen [SCFv03].

2.4 Ethernet Frames

Ein Ethernet Frame übermittelt Daten auf der Sicherungsschicht des OSI-Modell. Im Laufe der Entwicklung des Internets sind dabei verschiedene Formate entstanden. Das am häufigsten verwendete Rahmenformat im Internet ist Ethernet II. In Tabelle 2.1 wird der Aufbau eines Ethernet II demonstriert. Die ersten Felder, Präambel und SFD, sind Signale für den Empfänger, damit dieser sich mit dem Sender synchronisiert und den Start der Daten erkennt. Die nächsten beiden Felder sind die Mac-Adresse des Empfängers (Zieladresse) und des Senders (Quelladresse). Welches Protokoll in der Nutzlast des Rahmens eingekapselt ist, wird im Typ angegeben. Dann kommen die eigentliche Daten, welche eine Mindestgröße von 46 Byte haben müssen und eine Maximalgröße von 1500 Byte haben dürfen. Das letzte Feld ist das Frame Check Sequence (FCS), welches eine Prüfsumme enthält, um Fehler bei der Übertragung zu erkennen. Eine Pause von 96 Bit-Zeiten wird als Inter Frame GAP bezeichnet und tritt auf, nachdem ein Frame gesendet wurde. Die Länge der Pause hängt dabei von der Datenrate ab [ele].
Der Ethernet 802.3 ist eine standardisierte Ethernet-Version vom Institute of Electrical and Electronics Engineers (IEEE). Der Rahmen, wie in Tabelle 2.2 zu sehen, ist fast gleich aufgebaut. Statt dem Typ-Feld hat der Ethernet 802.3 die Länge an der Stelle. Zusätzliche Felder sind DSAP, SSAP und Control, welche dem Frame die Logical Link Control hinzufügen. Im LAN Bereich ist

dieser Rahmen am Weitesten verbreitetet. Der Logical Link Control befindet sich ebenfalls auf der Sicherungsschicht des OSI-Modells [ele].
Tabelle 2.3 zeigt den erweiterten 802.3 Ethernet-Frame, bei welchem noch ein zusätzliches 5 Byte großes Feld (SNAP) hinzugefügt wird. SNAP steht für Subnetwork Access Protocol. Dieses Feld enthält einen Code, um den Rahmen einer Firma und ein Typenfeld zuzuordnen [ele]. Im Gegensatz zu Logical Link Control gewährleistet SNAP auch die Abwärtskompatibilität zu Ethernet II, da hier das Typenfeld zwei Byte groß ist.
Zusätzlich kann zu allen vorgestellten Rahmen noch ein 4 Byte großes Feld zugefügt werden. Dieses Feld wird als Tag bzw. VLAN Tag bezeichnet [ele]. Das Feld wird wie in Tabelle 2.4 beispielhaft nach der Quelladresse hinzugefügt.

Präambel	**SFD**	**Ethernet-Frame: min. 64 Byte / max. 1518 Byte**					**Inter Frame Gap**
101010..	**10101011**	**Zieladresse**	**Quelladresse**	**Typ**	**Daten**	**FCS**	
8 Byte		6 Byte	6 Byte	2 Byte	46 - 1500 Byte	4 Byte	96 Bitzeiten

Tabelle 2.1: Ethernet II [ele]

Präambel	**SFD**	**Ethernet-Frame: min. 64 Byte / max. 1518 Byte**								**Inter Frame Gap**
101010..	**10101011**	**Zieladresse**	**Quelladresse**	**Länge**	**DSAP**	**SSAP**	**Control**	**Daten**	**FCS**	
8 Byte		6 Byte	6 Byte	2 Byte	1 Byte	1 Byte	1 Byte	42 - 1497 Byte	4 Byte	96 Bitzeiten

Tabelle 2.2: Ethernet 802.3 [ele]

Präambel	**SFD**	**Ethernet-Frame: min. 64 Byte / max. 1518 Byte**									**Inter Frame Gap**
101010..	**10101011**	**Zieladresse**	**Quelladresse**	**Länge**	**DSAP**	**SSAP**	**Control**	**SNAP**	**Daten**	**FCS**	
8 Byte		6 Byte	6 Byte	2 Byte	0xAA	0xAA	0×03	5 Byte	38 - 1492 Byte	4 Byte	96 Bitzeiten

Tabelle 2.3: Ethernet 802.3 SNAP [ele]

Präambel	**SFD**	**Ethernet-Frame: min. 68 Byte / max. 1522 Byte**						**Inter Frame Gap**
101010..	**10101011**	**Zieladresse**	**Quelladresse**	**Tag**	**Typ**	**Daten**	**FCS**	
8 Byte		6 Byte	6 Byte	4 Byte	2 Byte	46 - 1500 Byte	4 Byte	96 Bitzeiten

Tabelle 2.4: Ethernet II tagged [ele]

2.5 extended Berkeley Packet Filters (eBPF)

eBPF ist eine Erweiterung von Berkeley Packet Filter (BPF). BPF ist ein Tool um die Leistung von Paketen zu erfassen. Erstmals wurde es im Jahre 1992 entwickelt. Nach einer Neuauflage 2013, wurde es 2014 in den Linux-Kernel aufgenommen [Gre19]. eBPF wird hauptsächlich für Anwendungen von Sicherheit, Vernetzung und der Beobachtung (“Tracing“) benutzt. Mit eBPF kann der komplette Software-Stack effektiv betrachtet werden. Es ist möglich, die Performance von geschriebenen Programmen zu überwachen, den Netzverkehr zu filtern, DDoS-Angriffe frühzeitig zu erkennen oder das gesamte System zu beobachten [Gre19].

Im Kernel gibt es nur eine Ausführung-Engine (eBPF-virtuelle Maschine (VM)), welche sowohl eBPF als auch BPF-Programme ausführen kann. Mit eBPF-Programmen können nicht nur Kernel-Entwickler den Kernel programmieren bzw. anpassen. In der eBPF-VM gibt es zum Speichern von Daten zwischen den Ausführungen oder zum Austausch mit dem Userspace BPF “Maps“. eBPF hat im Vergleich zu BPF mehr Register, welche als globale Datenspeicher fungieren [Cil21]. Die gewünschten Informationen werden mit eBPF in drei Schritten ermittelt. Im ersten Schritt werden im Kernel der Eingang und Ausgang (E/A) aktiviert und das eBPF-Programm hinzugefügt. Als Nächstes wird das BPF-Programm bei jedem Event ausgeführt und die Information in einer eBPF-Map gespeichert. Im letzten Schritt holt sich der Benutzer (Userspace) die Information aus der Map und gibt dieses aus. In einem normalen Userspace-Programm werden zwei weitere Schritte benötigt, außerdem müssten die Information mehrmals hin und her kopiert werden bis diese schlussendlich beim Nutzer ankommen. Vorteil hierbei ist, dass ein eBPF-Programm auf jedem Rechner ausgeführt werden kann, sobald es einmal kompiliert wurde. Das zu beobachtende Programm muss für die Überwachung mit BPF nicht neugestartet werden. Es kann einfach auf Produktivsystemen ohne großen Aufwand eingesetzt werden [Gre19]. Jedes eBPF-Programm braucht einen “Kernel-Hook“, definierte Hilfsfunktionen, welches es verwenden darf und den geschrieben Programmcode. Diese drei Punkte bestimmen seinen Typ. “Kernel-Hooks“ sind eine Schnittstelle um individuelle Programme zu registrieren, welche dann bei bestimmten Ereignissen ausgeführt werden [Cil21]. eBPF-Programm liefern einen Rückgabewert, der je nach verwendetem “Hook“ eine andere Bedeutung haben kann. So gibt beispielsweise der “Socket-Filter-Hook“ einen Rückgabewert, welche die Paketlänge angibt, zurück.

Da es relativ schwer ist BPF Befehle direkt zu schreiben, wurden Frameworks entwickelt. Eines dieser Frameworks ist BPF Compiler Collection (BCC) [IO 23], welches zum Tracing benutzt werden kann. Es kann in verschiedenen Programmiersprachen wie C, C++ oder Python geschrieben werden [Gre19].

2.5.1 eBPF-System

Um ein eBPF-Programm in den Kernel zu laden, wird der Quellcode meist in einer C-artigen-Sprache geschrieben. Diese Sprache ist eine Teilmenge von der Programmiersprache C und schließt manche Bibliotheken aus. So steht dem Programmierenden beispielsweise “pintf“ nicht zur Verfügung und es können keine nicht-statischen globalen Variablen definiert werden. Unsichere Funktionen oder Schleifen ohne eine Oberbegrenzung sind ebenfalls nicht erlaubt, da diese wichtige Threads blockieren könnten. Um eBPF-Maps zu manipulieren gibt es spezielle Hilfsfunktionen.
Ein Compiler übersetzt den Code in Objekt-Code und über einen Systemaufruf wird das Programm in den Kernel eingefügt. Der Compiler erstellt aus dem gegebenen Sourcecode direkte Anweisungen.

Während des Einfügens prüft der Verifier das eBPF-Programm, ob es korrekt ist und keinen Schaden am Kernel anrichten kann. Der Verifier führt eine statische Programmanalyse von den eBPF-Anweisungen durch und prüft die Korrektheit der oben genannten Einschränkungen. Es wird außerdem geprüft, ob das Programm zu groß ist und ob die Speicheradresse innerhalb des vorgegebenen Adressbereichen liegen. eBPF-Befehle werden von einer eBPF-VM, welche einen dynamischen Just-In-Time Compiler (JIT) und einen Interpreter enthält, im Linux-Kernel ausgeführt [Cil21].

2.5.2 Maps

Maps sind spezielle Datenstrukturen in eBPF-Programmen um Daten auszutauschen. Es handelt sich dabei um Key-Value-Speicher. Die Map kann individuell erstellt werden, so kann der Typ ein Array oder ein Hash sein. Weitere Attribute beim Erstellen sind die Anzahl der Bytes des Schlüssels und die Werte. Beim Erstellen der Map durch den Userspace wird im Kernel eine Variable angelegt. Diese Zahl repräsentiert die Anzahl an eBPF-Programmen, welche auf die Map zugreifen. Die Zahl wird inkrementiert sobald ein neues Programm drauf zugreift und dekrementiert, wenn es beendet wird. Sollte das erstellende Programm beendet werden, abstürzen oder der Wert der Zahl auf null fallen, wird der Speicher der Map freigegeben. Dieses Vorgehen ermöglicht eine sehr einfache gleichzeitige Nutzung der Map von mehreren eBPF-Programmen. Ein weiterer Vorteil von dieser Vorgehensweise ist, dass die Map nur so lange im Speicher verbleibt, wie sie gebraucht wird. Will der Programmierende die Map auch ohne ein laufendes Programm erhalten, kann er es an das eBPF-Dateisystem pinnen, sodass es nicht zerstört wird [Cil21].

2.5.3 Hilfsfunktionen

extended Berkeley Packet Filter (eBPF)-Programme können nicht jede Funktion im Kernel aufrufen, jedoch gibt es Hilfsfunktion, welche die eBPF-API bereitstellt, um mit der Einschränkung umzugehen. Diese Hilfsfunktion ermöglichen die Interaktion mit anderen Kernel-Einrichtungen. Aufgaben der Hilfsfunktionen sind die Ausgabe von Nachrichten aus dem Kernel, Interaktion mit Maps oder der Änderung von Paketen. Je nach verwendetem “Hook“ gibt es unterschiedliche Hilfsfunktionen, die zur Verfügung stehen. Das Hinzufügen neuer Hilfsfunktionen ist nicht möglich, da hier sonst der Kernel-Quellcode geändert werden müsste. Es sind nur die Funktionen verfügbar, welche vom Kernel bereitgestellt werden [Cil21].

2.6 eXpress Data Path

XDP ist eine programmierbare leistungsstarke Schnittstelle um Netzwerkpakete im Linux-Kernel zum frühestmöglichen Zeitpunkt zu verarbeiten [Cil21]. XDP befindet sich auf der untersten Schicht des Netzwerkstapels-Kernels. Mittels eines eingebauten “Hooks“ auf der Schicht können eBPF-Programme angebracht werden [VCP+20]. Diese eBPF-Programme werden immer dann ausgeführt, wenn das Paket aus dem Empfangsring in den Kernel geladen wird. Dies bringt den Vorteil, dass noch keine anderen kostspieligen Operationen ausgeführt wurden und im Moment des Laden der Pakete eine CPU zu Verfügung steht, die wiederum das eBPF-Programm ausführen kann [Cil21]. Diese Programme können schnell auf die Pakete reagieren und beispielsweise einen

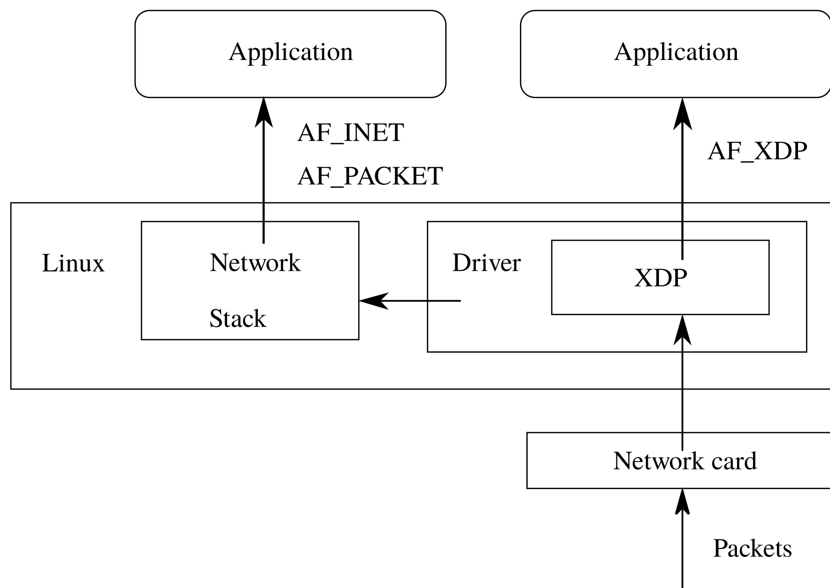

Abbildung 2.4: Funktionsweise von XDP und AF_XDP [KT18]

DDoS-Angriff erkennen und abschwächen [VCP+20]. Nach der Beendigung des XDP-Programms gibt es eine Entscheidung, wie das Paket weiter verarbeitet werden soll. Das Paket kann entweder verworfen, an den Kernel weitergegeben, an eine andere Netzwerkkarte, an einen AF_XDP-Socket oder an den Userspace, weitergeleitet werden [VCP+20]. In Abbildung 2.4 wird das Funktionsprinzip von XDP auf verschiedene Wege gezeigt, von der Weiterleitung der Pakete über die Netzwerkkarte und den Kernel in den Userspace.

2.6.1 AF XDP

Mit AF_XDP wird ein Socket erzeugt mit dem es möglich ist, Pakete an eine Userspace-Anwendung weiterzuleiten. Das Ziel des Sockets ist es, so schnell wie möglich, effizient und robust, Pakete von dem Treiber an den Userspace zu übergeben.
Auf dem Socket kann ein Receive (RX) und ein Transmit (TX)-Ring registriert und dimensioniert werden, dabei muss aber mindestens Einer registriert sein. Mit dem RX lassen sich Pakete empfangen und mit dem TX können Pakete verschickt werden. Um ein Paket nicht zwischen den Ringen hin und her kopieren zu müssen, können sich RX und TX den gleichen Datenpuffer im Speicher (UMEM) teilen. UMEM ist ein zusammenhängender virtueller Speicher, welcher unterteilt ist in gleich große Rahmen. Speicher bekommt der UMEM aus dem Userspace zugewiesen. Der UMEM hat zusätzlich zwei Ringe, den COMPLETION-Ring und den FILL-Ring. Vom Userspace werden Adressen aus dem FILL-Ring gesendet, die der Kernel mit RX-Paketen füllen soll. Sobald ein neues Paket empfangen wurde, taucht ein Verweis im RX-Ring auf. Im Completion-Ring (CL-Ring) sind hingegen die Pakete, die über den TX-Ring erfolgreich gesendet wurden. Ein Deskriptor ist ein Zeiger, der innerhalb des Ringes auf die Adresse des Rahmens zeigt [KT18].

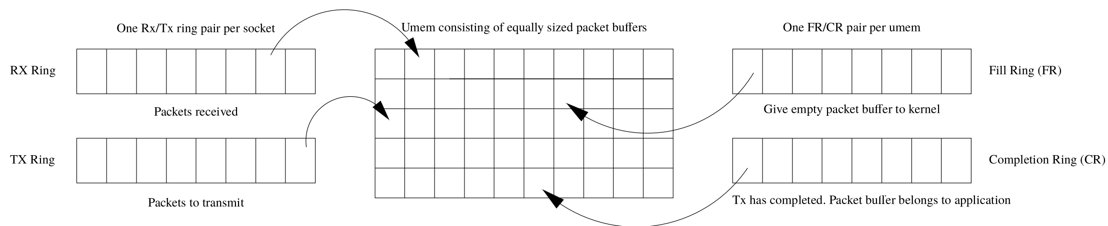

Abbildung 2.5: Funktionsweise von XDP-Ringen und UMEM [KT18]

Der Ablauf, wenn ein Paket an der Netzwerkkarte ankommt und an einen AF_XDP-Socket weitergeleitet wird, ist Folgender: Das Paket trifft an der Netzwerkkarte ein und wird auf der physischen Ebene verarbeitet. Wenn das Paket für einen AF_XDP-Socket bestimmt ist, wird es an den XDP-Programmierbereich weitergeleitet, wo ein vorher definiertes XDP-Programm ausgeführt wird. Das XDP-Programm verarbeitet das Paket auf der Ebene der Netzwerkkarte und trifft Entscheidungen, wie das Paket weiterverarbeitet werden soll, z.B. ob es fallen gelassen wird oder an eine bestimmte Anwendung oder einen Socket weitergeleitet werden soll. Wenn das Paket an einen AF_XDP-Socket weitergeleitet wird, wird es direkt an den Speicherbereich des Sockets im Benutzerbereich weitergeleitet, ohne dass es durch den Kernel geht. Das AF_XDP-System verwendet eBPF-Maps, um den AF_XDP-Socket-Ring mit dem XDP-Programm zu verbinden. Die Anwendung, die auf den AF_XDP-Socket hört, empfängt das Paket und kann es entweder weiterverarbeiten oder darauf reagieren, z.B. indem es eine Antwort an den Absender sendet [KT18]. In Abbildung 2.4 wird der Ablauf erst von der Netzwerkkarte zum Socket demonstriert und in Abbildung 2.5 wird das Zusammenspiel von UMEM und den XDP-Ringen gezeigt.

Jeder AF_XDP kann nur mit einem UMEM verbunden sein, jedoch kann jeder UMEM mehrere AF_XDP haben [TB].
Es gibt drei verschiedene Modi, in denen der Socket betrieben werden kann. Der schnellste Modus ist der Zero-Copy-Modus, bei welchem zwischen dem Kernel und dem Userspace nur Adressen ausgetauscht wird und nicht kopiert wird [KT18].

Der Socket ermöglicht einen Durchsatz von ca. 20 Million packets per second (MPPS) pro Anwendungskern für 64-Byte-Pakete. Mit verschiedene Optimierung sollen sich laut [KT18] bis zu 40 MPPS auf dem RX und 70 MPPS für TX erreichen lassen. In der Regel werden für ein AF_XDP zwei Kerne benötigt, einen für die Userspace-Anwedung und einen für Sende und Empfangsvorgänge. Es gibt aber auch die Möglichkeit alles auf einem Kern laufen zu lassen, um die Performance zu steigern [KT18].

3 Verwandte Arbeiten

In den folgenden Abschnitten werden Forschungsthemen und Projekte vorgestellt, die in engem Zusammenhang mit dieser Arbeit stehen. Zunächst wird in Abschnitt 3.1 ein Überblick über Netmap gegeben. Anschließend wird in Abschnitt 3.2 ein Blick auf Data Plane Development Kit geworfen. Abschnitt 3.3 zeigt drei verschiedene Software-Switches.

3.1 Netmap

Netmap [lui23] ist ein Framework für Hochgeschwindigkeits-Paket-E/A, welches für FreeBSD entwickelt wurde, aber auch für Linux verfügbar ist. Es ermöglicht Anwendungen auf dem Userspace einen schnellen Zugriff auf Netzwerkpakete. Der Zugriff ist dabei auf der Empfangs- als auch auf der Sendeseite möglich. Ein großer Vorteil von Netmap ist, dass es Hardwareunabhängig ist [RCC12]. Aktionen, die das Betriebssystem schaden könnten, sind nicht möglich, da das System Aktionen bzw. Programmcode im Vorhinein validiert. Hierfür werden kostspielige Systemaufrufe und unnötige Laufzeitkosten entfernt. Die Netmap API erstellt hierfür Kopien der Netzwerkkarte (NIC)-Ringe und verwendet diese um die E/A von Paketen zu unterstützen. Diese Kopien werden in Netmap-Ringe gespeichert, welche einfache Warteschlangen von Puffern im gemeinsamen Speicher, von Kernel und von allen Benutzerprozessen sind. Synchronisierung zwischen Anwendung und Kernel erfolgt durch Systemaufrufe, z.B. "poll", sodass immer nur eine Seite (Anwendung, Kernel) auf gemeinsame Speicherbereiche zugreift kann. Dies ermöglicht zwischen den Schnittstellen eine "Zero-Copy-Übertragung". Wichtiges Konzept, das zu hoher Performance führt ist "Batching". Mit einem Systemaufruf können viele Pakete vom Kernel an die Anwendung oder umgekehrt übergeben werden, daher gibt es einen geringeren "Overhead" für Kontextwechsel. Um Netmap nutzen zu können, muss eine kleine C-Header-Datei, in welcher alle nötigen Strukturen und Makros sind, in seinen Quellcode eingebunden werden [RCC12].
Netmap ändert die Netzwerktreiber ab, um die Pakete zu empfangen. Es werden hierfür zwei Funktion implementiert, um die Pakete und die Reinitialisierung der Ringe im Netmap-Modus zu erzielen [RCC12].
Die Übertragung eines einzelnen Paketes dauerte bei einem Experiment mit dem Frameworks weniger als 70 CPU-Taktzyklen. Hochgerechnet auf einen einzelnen 900 MHz Kern sind das 14,88 MPPS, die mit einer 10-Gbit/s-Verbindung erreicht werden können [RL12].

Im Vergleich mit dieser Arbeit haben XDP und Netmap das Ziel Netzwerkpakete dem Userspace zur Verfügung zu stellen, die geschieht aber auf unterschiedlichen Wegen. Während diese Arbeit auf Raw-Sockets setzt, welche die Treiber so abändert, dass die Pakete nach der Netzwerkkarte abgegriffen und in Ringe speichern werden, ändert Netmap hingegen die Netzwerktreiber so ab, dass die Pakete an eine eigens entwickelte Datenstruktur senden, wo sie der Userspace abgreifen

kann [RCC12]. Wesentlicher Unterschied zwischen XDP und Netmap ist, dass Netmap nur schnell Pakete an den Userspace weitergeben kann, während XDP tatsächlich Pakete im Kernel filtern kann, welche an die Anwendung weitergegeben werden sollen.

3.2 Data Plane Development Kit

Data Plane Development Kit (DPDK) [DPD] ist ein von Intel entwickeltes Framework zur schnellen Verarbeitung von Netzwerkpaketen. Es ist Open-Source und wird seit 2019 von der Linux Foundation verwaltet. Das Framework besteht aus verschieden Bibliotheken und kann auf vielen CPU-Architekturen ausgeführt werden [DPD]. Die Bibliotheken sind für hohe Leistungen optimiert und erfüllen grundlegende Netzwerkaufgaben, ähnlich wie der Linux-Netzwerkstack. Beispielsweise übernimmt DPDK die Pufferung oder die Speicherzuweisung von Netzwerkpakete [Sch14].
Um eine hohe Verarbeitungsgeschwindigkeit zu erreichen, nutzt DPDK die Kernel-Bypass-Technologie. Die Kontrolle über die Netzwerkkarte wird hierfür von DPDK, mittels eines eigens entwickelten User-Space-NIC-Treiber, übernommen. Der Treiber verhindert nun nach dem Laden, dass das System noch auf die Netzwerkkarte zugreifen kann. Durch die direkte und einzige Kommunikation zwischen DPDK und Netzwerkkarte, können eingehende Netzwerkpakete direkt an den Userspace geliefert werden. Die hohe Performance von DPDK kommt, wie auch bei Netmap, von Batching, was zur Minimierung von Systemaufrufe und Kontextwechsel führt. Außerdem implementiert DPDK sogenannte "Poll-Mode-Treiber", die mit "Busy-Waiting" auf die NIC zugreifen und so die Latenz reduzieren. DPDK besitzt einen vereinfachten Netzwerkstapel, der keine unnötigen Module benötigt. Die Netzwerkpakete werden von DPDK direkt von dem Stapel der Netzwerkkarte abgerufen. Dies ermöglicht es mehrere Pakete gleichzeitig abzuholen und verhindert gleichzeitig teure Systemaufrufe im Kernel [FOC+22]. Für die Verwaltung von Warteschlangen gibt es Ringe, die nach dem FIFO-Prinzip befüllt und entnommen werden. Implementiert ist der Ring als Tabelle von Zeigern. Durch die festgelegt Größe zu Beginn der Programmierung, ist der Ring sehr performant und vermeidet kostspielige Speicherzuweisungsprozeduren. Die Ringe können zur Pufferung von Netzpaketen und zur Kommunikation zwischen Anwendungen benutzt werden [Sch14].
Ohne Kernel muss DPDK Funktionen implementieren, welche eigentlich schon im Kernel vorhanden sind, wie Routing-Tabellen, Protokolle oder Gerätetreiber. Standardwerkzeuge, wie Tcpdump[1], sind durch die Isolation von DPDK mit der Netzwerkkarte und dem Betriebssystem nicht mehr nutzbar. Sicherheitsmechanismen, wie der kontrollierte Hardware-E/A oder Prozessisolierung, müssen ebenfalls von DPDK bereitgestellt werden [FOC+22].

Werden XDP, was in dieser Arbeit genutzt wird, mit DPDK verglichen, haben beide Technologien das gleiche Ziel, die Netzwerkpaketverarbeitung zu beschleunigen. Beide Technologien nutzen Ringe um die Netzwerkpakete zwischenzuspeichern. DPDK ersetzt den kompletten Kernel für die Netzwerkkommunikation, indem es den Treiber der Netzwerkkarte ändert, während XDP bzw. AF_XDP auf das Betriebssystem setzt. DPDK muss separat auf dem System installiert werden, was XDP nicht muss. Wie schon in Netmap erwähnt, ermöglicht die Kombination aus XDP und BPF die Vorfilterung von Paketen, welche die Userspace-Anwendung erreichen sollen, was in DPDK nicht möglich ist.

[1]Tcpdump ist ein Programm, um Netzwerkverkehr im System aufzuzeichnen.

3.3 Software-Switch

Software-Switche oder auch Softswitches genannt sind Switche, welche mit Hilfe von Software die Paketvermittlung in Netzwerken übernehmen. Es wird dabei trotzdem Hardware benötigt, wie ein System auf dem die Software läuft, sowie eine Netzwerkkarte zum Annehmen von Paketen. Eine weitere Anwendung von Software-Switche ist die Übernahme der Kommunikation zwischen VMs und dem Betriebssystemen. In Abschnitt 3.3.1 wird ein Switch vorgestellt der auf Basis von dem Netmap-Framework arbeitet. Anschließend wird in Abschnitt 3.3.2 ein Blick auf Open vSwitch geworfen. Abschließend wird ein Simulations-Framework vorgestellt.

3.3.1 Netmap Vale

Virtual Local Ethernet (VALE) [LG] ist ein virtueller Software-Switch, welcher über einen Port mit dem Netmap-Application Programming Interface (API) kommunizieren kann. VALE wird in der Regel zur Kommunikation zwischen Prozessen wie VMs und dem System genutzt. Es kann sich mit Hardware-Geräten verbinden, um mit externen Systemen zu kommunizieren. Der Switch kann hohe Paketraten erreichen, mit einem Durchsatz von 17 MPPS . Es ist als Kernelmodul implementiert und erweitert das Netmap-Framework. Innerhalb von einem System können mehrere Switches erstellt werden. Jede Instanz lernt wie bei einem normalen Switch, bei der Übertragung von Paketen, wie diese weitergeleitet werden müssen [Riz12].

Während VALE das Ziel hat zwischen Prozessen wie VM zu kommunizieren, will diese Arbeit ein Software-Framework für Scheduling-Algorithmen bereitstellen.

3.3.2 Open vSwitch

Open vSwitch (OSV) [A L16] ist ein virtueller Multilayer-Software-Switch, der häufig in Rechenzentren als Hypervisor-Switch anzutreffen ist [OTCK20]. Hypervisor ist ein Programm um virtuelle Maschine (VM) auf einem System zu erstellen und zu verwalten. OSV isololiert das Betriebssystem von den virtuelle Maschine und nimmt die restlichen Ressourcen als ein Ganzes und teilt es dann gleichmäßig auf [Red20]. Die Paketweiterverarbeitung in OSV wird von zwei Services übernommen. Der erste Service ist der Datenpfad (Fastpath), der für die Weiterleitung der Pakete verantwortlich ist. Im Falle von OSV-DPDK ist dieser Service im Userspace implementiert. "Ovs-vswitchd" (Slowpath) ist die zweite Komponente, welche anhand einer Tabelle bestimmt, wie der Fastpath mit eingehenden Paketen weiter verfahren soll. [OTCK20]

Wie auch bei Netmap Vale ist das Ziel die Kommunikation zwischen VM und dem Betriebssystem und nicht das Scheduling.

3.3.3 Mininet

Mininet [BN22] ist ein Open-Source-Projekt, um Netzwerknetze auf einem Rechner zu emulieren. Router, Switch, Host oder auch Links können auf einem einzigen Linux-Kernel ausgeführt werden. Mininet-Host verhält sich wie ein echter Rechner, sodass es sich auf die Virtuellen Hosts mit Secure Shell (SSH) eingeloggt werden kann und beliebige Programme ausführt werden können. Über eine

emulierte Ethernet-Schnittstelle können diese Programme dann Pakete senden. Die Pakete werden dann von emulierten Switches und Router verarbeitet, bis diese an ihr Zieladresse kommen. Da Mininet nur von einem einzigen Linux-Kernel ausgeführt wird, können keine Programme ausgeführt werden, welche den Betriebssystem-Kernel benötigen. Mininet ist schnell, sodass das Starten eines einfachen Netzwerks nur wenige Sekunden dauert. Die Weiterleitung von Paketen können in den Switches mittels des OpenFlow-Protokoll[2] angepasst werden [BN22].

Mininet simuliert ganze Netzwerke, wohingegen das Ziel dieser Arbeit die Implementierung einzelner Switche auf physisch vernetzten Rechner ist, die z.B. Teil eines Testbeds sein können.

[2] Mit OpenFlow können Weiterleitungstabellen angepasst werden [BN22].

4 Systemmodell und Problemstellung

4.1 Systemmodell

Das Systemmodell besteht aus mehreren Komponenten die zusammenarbeiten, um die Datenübertragung innerhalb des Netzwerks zu ermöglichen. Das System umfasst einen Sender und einen Empfänger-Prozess, die auf einem Host-System laufen. Die beiden Prozesse kommunizieren über das Netzwerk und tauschen Datenpakete aus.
Das Netzwerk, das auf einer Ethernet-Technologie basiert, wird durch einen Software-Switch realisiert. Der Software-Switch ist eine Anwendung, die auf den Hosts läuft und die Weiterleitung und das Scheduling von Paketen durchführt. Der Switch wird verwendet, um den Datenverkehr im Netzwerk zu regeln und zu priorisieren, damit ein optimaler Datendurchsatz erreicht wird.
Die Hosts, auf denen der Software-Switch ausgeführt wird, unterstützen XDP und eBPF. Diese Technologien werden verwendet, um den Datendurchsatz und die Verarbeitungseffizienz zu verbessern. XDP ist eine Technologie die es ermöglicht, eingehende Pakete direkt an die Anwendungsebene bzw. den Prozess weiterzuleiten, der den Software-Switch implementiert.
Mithilfe von XDP können Pakete schneller verarbeitet werden, da sie nicht mehrere Schichten im Netzwerkstack durchlaufen müssen. Stattdessen können sie direkt an die Anwendungsebene weitergeleitet werden. Mit eBPF können Anwendungen den Datenverkehr im Netzwerk steuern und verwalten, um die Verarbeitungseffizienz und die Sicherheit zu verbessern.
Ein weiterer Vorteil von XDP ist der CL-Ring, der von der Anwendung genutzt werden kann um Feedback vom Kernel über die versendeten Pakete zu erhalten. Dies ermöglicht der Anwendung auf Netzwerkereignisse in Echtzeit zu reagieren und ihre Leistung zu optimieren.

4.2 Problemstellung

Das Problem der Korrektheit bei der Übertragung von Daten besteht in der Pufferung von Paketen. Wenn mehrere Pakete von einem Sender an einen Empfänger übertragen werden, müssen sie in der Regel in einer bestimmten Reihenfolge ankommen, um die Integrität der übertragenen Daten zu gewährleisten. Ein gängiges Verfahren, um dies zu erreichen, ist das sogenannte Scheduling.
Das Ziel eines korrekten Schedulings ist es, die Scheduling-Entscheidung möglichst spät zu treffen, sobald ein "Vorgängerpaket" versendet wurde und unmittelbar bevor das nächste Paket gesendet wird. Der Grund dafür liegt darin, dass sich die Netzwerkbedingungen stetig ändern können und eine Entscheidung zu einem frühen Zeitpunkt möglicherweise zu Fehlern führt.
Ein Beispiel hierfür ist eine Videokonferenz, bei der mehrere Teilnehmer Daten über das Netzwerk senden. Wenn ein Teilnehmer zu früh entscheidet, welches Paket als nächstes gesendet werden soll, kann dies dazu führen, dass das Paket aufgrund von Netzwerküberlastung oder anderen Störungen nicht rechtzeitig ankommt. Dies kann zu Fehlern bei der Wiedergabe führen und die Qualität der Konferenz beeinträchtigen.

Um dieses Problem zu lösen, muss das Scheduling sorgfältig durchgeführt werden, um sicherzustellen, dass die Entscheidungen so spät wie möglich getroffen werden, ohne dabei die Integrität der übertragenen Daten zu gefährden.

Ziel dieser Arbeit ist der Entwurf und die Implementierung eines Frameworks zur Implementierung der Software-Switches auf Anwendungsebene, die mit Hilfe von eBPF und XDP Pakete direkt vom Kernel empfangen und senden. Ein weiteres Ziel ist es, ein generisches und erweiterbares Framework für Scheduling-Verfahren zu schaffen. Hierbei soll gezeigt werden, dass das Framework leicht um neue Scheduling-Verfahren erweitert werden kann. Als Proof-of-Concept sollen ein zeitgesteuertes und ein prioritätengesteuertes Scheduling-Verfahren implementiert werden, um die Anwendbarkeit auf verschiedene Klassen von Scheduling-Algorithmen zu zeigen.
Ein weiteres Ziel des Frameworks ist es, die Leistung in Bezug auf den Paketdurchsatz zu optimieren. Es soll also sichergestellt werden, dass das Framework effektiv und effizient arbeitet, um eine maximale Anzahl von Paketen durchzuschleusen.
Ein wichtiges Ziel des Frameworks ist auch ein korrektes Scheduling-Verhalten. Hierbei ergibt sich das zentrale Problem, dass die Scheduling-Entscheidung möglichst nahe am Sendezeitpunkt des Pakets getroffen werden sollte. Dadurch wird sichergestellt, dass das nächste Paket gemäß dem Scheduling-Verfahren weitergeleitet wird. Dies wird jedoch durch die Pufferung auf verschiedenen Ebenen im System erschwert. Aus diesem Grund soll ein auf XDP basierender Ansatz entworfen und evaluiert werden, der dieses Problem weitestmöglich löst.

5 Entwurf

In diesem Kapitel wird der Entwurf eines Software-Switches präsentiert, welcher nicht nur die Systemarchitektur beschreibt, sondern auch die Konzepte zur Erfüllung der gestellten Anforderungen aufzeigt - darunter generische Erweiterbarkeit, korrektes Scheduling und hohe Leistung in Bezug auf den Durchsatz.

Das Framework nutzt die Bibliothek "xdp-tools" [Tok], um grundlegende Funktionalität von XDP bereitzustellen. Die "XSK"-Klasse, welche hauptsächlich genutzt wird, initialisiert den UMEM, die Network-Switches und die Ringe. Automatisch lädt das Framework ein eBPF-Programm in den Kernel, welches alle Pakete an eine eBPF-Map weiterleitet. Dort wird es vom Network-Switch empfangen.

5.1 Systemarchitektur und Architektur des Software-Switches

Der ganze Ablauf ist in Abbildung 5.1 zu sehen. Pakete werden von der Netzwerkkarte angenommen und vom Kernel an die eBPF-VM weitergegeben. Das eBPF-Programm fängt die Pakete ab und leitet diese weiter an den Userspace mittels einer eBPF-Map. Der RX-Thread greift auf diese Map zu, entnimmt die Pakete und speichert diese im UMEM, sowie in der geteilten Liste von RX-Thread und Forward-Thread. Die Pakete werden von dem Forward-Thread an den richtigen Absendeport geleitet und in die richtige Warteschlange des Ausgangsports klassifiziert. Der TX-Thread nimmt die Warteschlangen und "scheduled" diese. Für den Sendevorfang werden die Pakete aus dem UMEM genommen und an den TX-Ring des Sockets weitergegeben, welcher das Paket über eine eBPF-Map an den Kernel, welcher wiederum dieses an die Netzwerkkarte weitergibt. Sobald das Paket die Netzwerkkarte verlassen hat, wird wieder über eine eBPF-Map in den CL-Ring des Sockets geschrieben. Wenn das Paket in den CL-Ring geschrieben wurde, nimmt der TX-Thread jenes und gibt es aus dem UMEM frei. Der Ansatz, bei dem auf eine Bestätigung des Versendens des Pakets gewartet wird, wird als Stop-and-Wait-Ansatz bezeichnet. Im Gegensatz dazu gibt es noch den Batching-Ansatz bei dem so viele Pakete wie möglich an den Kernel gegeben werden, um so wenig wie möglich Kontextwechsel zwischen Anwendung und Kernel zu erreichen.

5.2 Endwurfentscheidungen

Der implementierte Software-Switch verwendet Multithreading um den Empfang (RX), die Weiterleitung, die Klassifizierung (Forward) und das Scheduling und das Versenden (TX) gleichzeitig auszuführen. Dies hat mehrere Vorteile. Es können mehrere Aufgaben gleichzeitig ausführt werden, wodurch die Leistung insgesamt verbessert wird. Dies ist besonders nützlich in Umgebungen mit hohem Datenverkehr, in denen eine schnellere Verarbeitung erforderlich ist. Durch die Verwendung von Multithreading kann der Software-Switch besser skalieren und eine höhere Anzahl

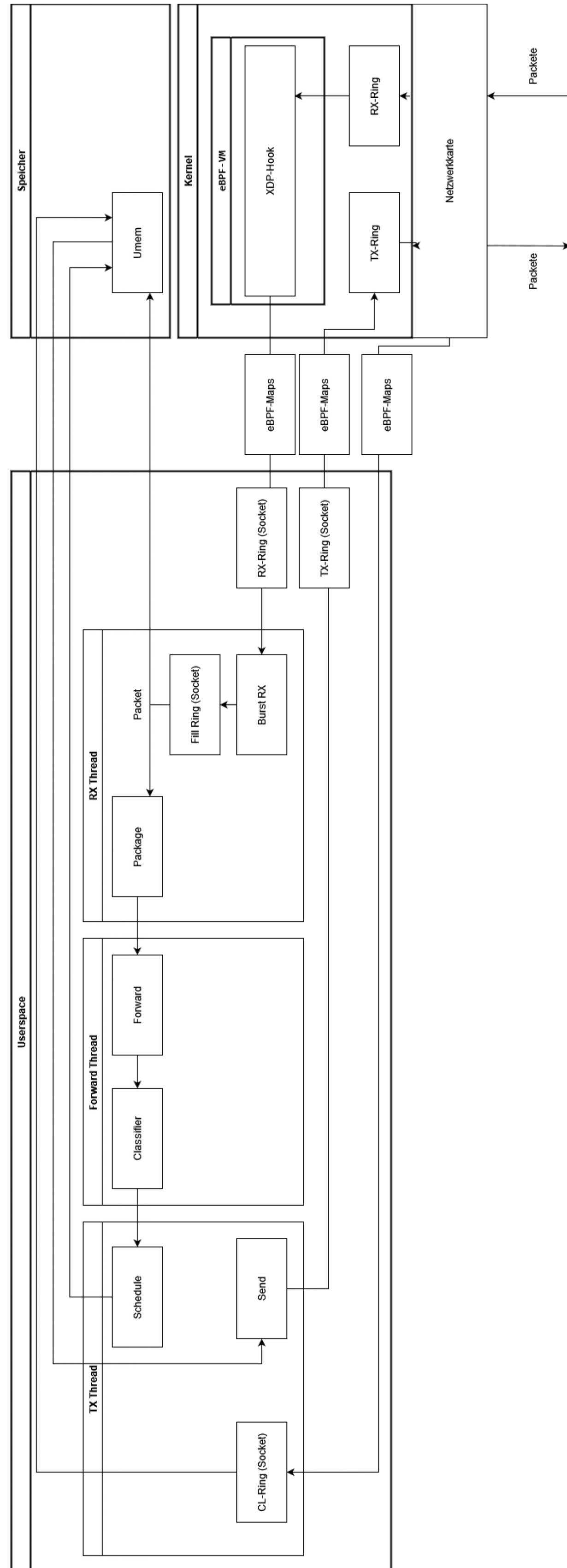

Abbildung 5.1: Software-Switch

von Verbindungen gleichzeitig verwalten, ohne dass dies zu einem Leistungsengpass führt. Die Verwendung von separaten Threads für verschiedene Aufgaben verringert das Risiko, dass ein Fehler in einem Thread den gesamten Switch beeinträchtigt. Insgesamt kann der Software-Switch, der Multithreading verwendet dazu beitragen, die Leistung, Skalierbarkeit und Zuverlässigkeit des Netzwerks zu verbessern.

Das Framework implementiert drei generische Klassen: die Forward -, Classifier - und Scheduler Klasse. Generische Klassen können dazu beitragen, die Skalierbarkeit des Software-Switches zu erhöhen, da sie es Entwickelnden ermöglicht, schnell neue Funktionen hinzuzufügen oder bestehende Funktionen zu erweitern, indem generische Klassen verwendet werden, anstatt jede Funktion von Grund auf neu zu schreiben.

Für die Zusammensetzung des Switches wurde bewusst eine Klasse entwickelt, die Network-Switch Klasse. Die Verwendung dieser Klasse zur Erstellung des Software-Switches ermöglicht eine modulare Struktur, in der Code einfach anpasst oder erweitert werden kann, ohne die gesamte Anwendung ändern zu müssen. Dies erleichtert die Wartung und Erweiterung der Anwendung. Es kann die Testbarkeit des Codes verbessern, da die einzelne Komponenten des Switches isoliert und separat getestet werden können, anstatt den gesamten Code der Anwendung testen zu müssen.

Für das Versenden in der richtigen Reihenfolge wurde auf mehr Durchsatz verzichtet. Das Warten bis die Netzwerkkarte das Paket in den Completion (CL)-Ring geschrieben hat, macht die richtigen Reihenfolge beim Versenden aus und ermöglicht den zuverlässige Empfang aller Pakete. Die Zuverlässigkeit ist bei einem Software-Switch, welcher zum Testen von Scheduling-Algorithmen entwickelt wurde, essenziell.

5.3 Network-Switch Buffer Klasse

Die Network-Switch Buffer Klasse ist eine Pufferverwaltung für Netzwerkschnittstellen. Die Klasse initialisiert und verwaltet einen Buffer-Pool, in dem die Puffer (Buffers) für die Sockets erstellt werden, um den Datenfluss zwischen der Anwendung und dem Netzwerk zu implementieren.
Diese Klasse ist aus der C-Datei "AF_XDP-forwarding" [Int22] aus den "bpf-examples" entstanden. Die Methoden und Structs sind hierfür lediglich kopiert worden.

In der Regel werden für die Weiterleitung von Paketen die Paketpuffer aus dem Pool für den Paketempfang zugewiesen und nach Abschluss der Übertragung wieder freigegeben, um sie weiter verwenden zu können.

Da der Pufferpool von mehreren Threads gemeinsam genutzt wird, kann es zu Zugriffslatenzen kommen. Um dies zu minimieren bzw. um die Leistung zu optimieren, erstellt jeder Thread einen Puffer-Cache, der nur für den Thread selbst privat ist und somit nicht mit anderen Threads geteilt wird. Der gemeinsame Pool wird nur dann genutzt, wenn der Cache entweder leer ist und aufgefüllt werden muss oder voll ist und zurück in den Pool geleitet werden muss.

Bei der Weiterleitung von Paketen kann ein empfangenes Paket je nach Konfiguration an jeden Ausgangsport gesendet werden. Um dies bei AF_XDP-Sockets mit Null-Kopie der Paketpuffer zu ermöglichen, muss der Pufferpoolspeicher in einem von allen Sockets gemeinsam genutzten UMEM-Bereich passen.

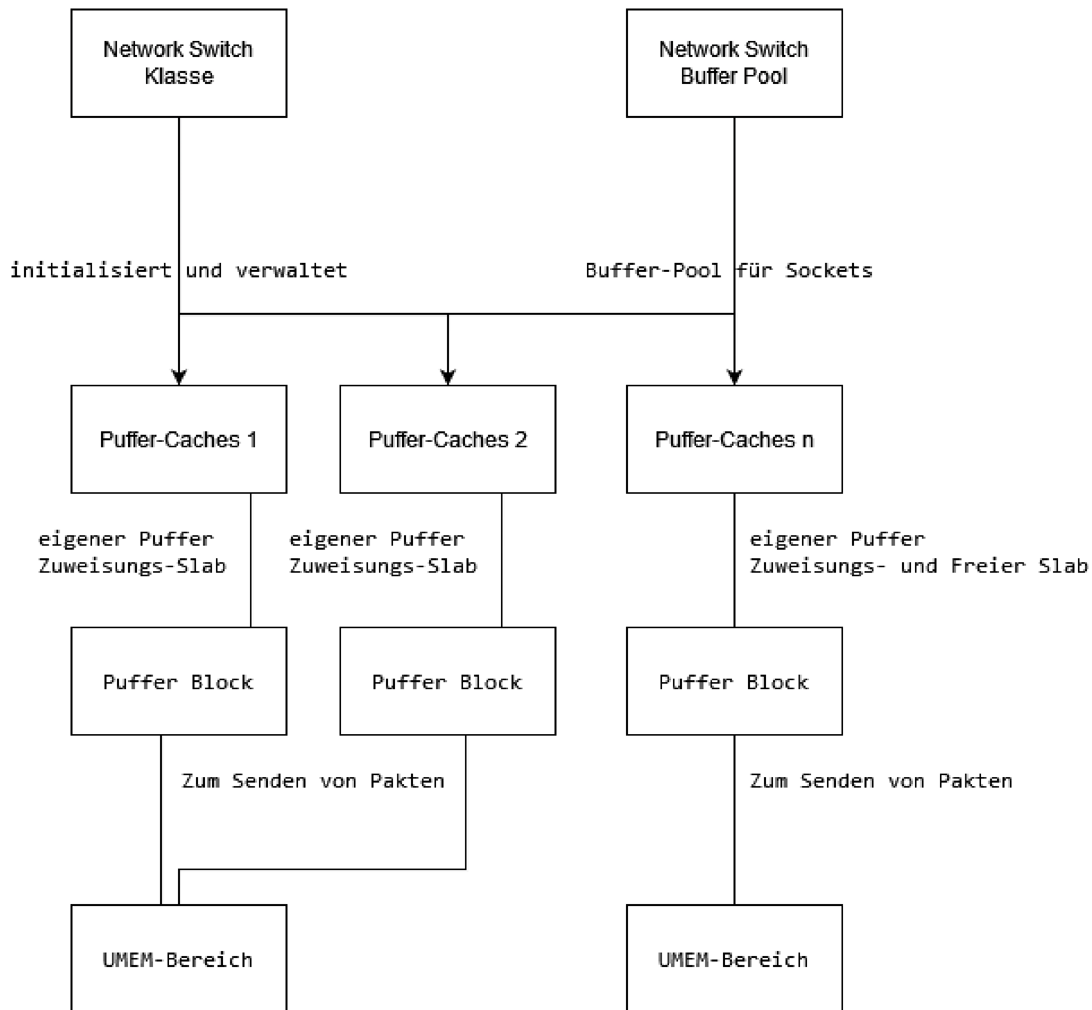

Abbildung 5.2: Pufferpool und Puffercache

Die Implementierung des Pufferpools organisiert die Puffer in gleichgroße Slabs mit einer bestimmten Anzahl von Puffern pro Slab. Ein Slab ist eine zusammenhängende Gruppe von Speicherbereichen, die alle die gleiche Größe haben. Im Falle des Pufferpools werden die Puffer in gleichgroße Slabs organisiert, um die Speicherverwaltung zu optimieren. Zu Beginn gibt es eine bestimmte Anzahl von vollen Slabs im Pool.

Jeder Puffer-Cache hat einen Slab für die Pufferzuweisung und einen Slab für freie Puffer. Wenn der Zuweisungsblock des Caches leer wird, wird er mit einem verfügbaren vollen Slab aus dem Pool getauscht. Wenn der freie Slab des Caches voll wird, wird er gegen einen leeren Slab aus dem Pool ausgetauscht, was garantiert erfolgreich ist. In Abbildung 5.2 ist zusätzlich eine Abbildung zum Pufferpool und Puffercache zu finden.

Bpool_params ist eine Struktur, die die Parameter des Buffer-Pools enthält, wie die Anzahl der Buffers, ihre Größe und die maximale Anzahl von Benutzenden des Pools. UMEM_cfg ist eine Struktur, die die Konfiguration der User-Mode DMA-Umgebung definiert, einschließlich der Größe der Füll- und Übertragungsräume sowie der Größe der Bufferrahmen. Buffer_pool ist eine Variable

vom Typ bpool, die den zugewiesenen Buffer-Pool speichert. Bpool_params und UMEM_cfg sind vorinitialisierte Structs, welche standardmäßig auf die Maximalwerte konfiguriert werden. Die statischen Structs können aber nach belieben mit eigenen Werten überschrieben werden.

Insgesamt stellt die Network-Switch Buffer Klasse eine einfache Verwaltung von Buffers für Sockets dar. Die Methoden zur Speicherverwaltung sind statisch und werden von der Klasse Network-Switch beim Zusammensetzen des Switch aufgerufen. Die Klasse stellt auch eine Abstraktionsschicht für User-Mode Direct Memory Access (DMA) auf Linux bereit, um eine schnelle Netzwerkkommunikation zu ermöglichen.

5.4 Port Klasse

Die Port-Klasse ist einer der wichtigsten Klassen im ganzen Switch. Diese Klasse ist eine Implementierung eines Netzwerkports und Teil eines größeren Systems, das auf Netzwerkverkehr reagiert. Die Klasse definiert Methoden, die die Interaktion mit dem Netzwerkport steuern, wie beispielsweise das Hinzufügen oder Entfernen von Paketen aus der Senderliste, das Abrufen von VLAN-IDs oder das Setzen des Klassifizierers.
Schließlich gibt es auch einige Hilfsmethoden, die die Verwaltung des Netzwerkports erleichtern, wie beispielsweise das Überprüfen von VLAN-IDs, das Abrufen der Geschwindigkeit des Netzwerkports oder das Ausgeben von Informationen über den Netzwerkport.

Die wichtigsten Eigenschaften des Netzwerkports ist das Interface und dessen "Interface-Queue", an welches sich dieser binden soll. Das sind zwei von drei Attributen, welche eine Port-Klasse beim Initialisieren haben muss. Die dritte Eigenschaft ist die Geschwindigkeit in Megabyte pro Sekunde, mit welcher der Port empfangen und senden kann diese Eigenschaft wird beim Scheduling benötigt. Optional können pro Port einzelne bestimmte Mac-Adressen, IP-Adressen oder Ethernet Typen bei der Klassifizierung bevorzugt werden. Falls dies gewünscht ist, muss jegliche eine Liste hiervon im Konstruktor übergeben werden. Standardmäßig werden acht Warteschlagen initialisiert, diese können jedoch vom Benutzenden auf eine verringert oder nach belieben erhöht werden.

Ein Port ist entweder kein VLAN-Access-Port, dann wird dem Port die nullte Adresse gegeben, oder ein VLAN-Access-Port, bei welchem die VLAN-Adresse nicht NULL sein darf. Es kann auch ein Trunk-Port definiert werden, welcher keine VLAN-Adresse hat, aber trozdem VLAN-Pakete empfangen und die entsprechenden VLAN-Ports verteilen kann. Um eine Statistik für den Port zu erhalten, gibt es drei private Variablen, die die Anzahl an Paketen zählt, welche empfangen (RX), versendet (TX) und bestätigt (CL) wurden.

Standardmäßig wird im Standard-Konstruktor intern die Ringgröße RX und TX vom Network-Switch festgelegt. Ein weiteres Attribut in dem Struct ist der "bind_flags". Sofern im bind-Aufruf die entsprechende Option gesetzt ist , wird das "need_wakeup"-Flag aktiviert. Dieses Flag weckt den Kernel durch einen Systemcall auf, um die weitere Verarbeitung von Paketen sicherzustellen [TB]. Das Struct kann jedoch vom Nutzenden neu gesetzt werden. Eine weitere Option für das "bind_flags" wäre "XDP_ZEROCOPY" oder "XDP_COPY". Beim Initialisieren ist "XDP_USE_NEED_WAKEUP" angegeben, da nicht jede Netzwerkkarte das "XDP_ZEROCOPY-Flag" unterstützt. Falls Zero-Copy nicht unterstützt wird, wird auf den Kopiermodus zurückgegriffen, bei dem alle Pakete in den Userspace kopiert werden. Durch Übergeben des "XDP_COPY-Flags" an den Bind-Aufruf wird der Socket in den Kopiermodus gezwungen. Falls der Kopiermodus

nicht verwendet werden kann, schlägt der Bind-Aufruf fehl. Im Gegensatz dazu erzwingt das "XDP_ZEROCOPY-Flag" den Socket in den Zero-Copy-Modus oder schlägt fehl. Das Zero-Copy-Flag bei AF-Sockets steht für eine Funktion, bei der Daten direkt zwischen dem Netzwerk-Stack und Anwendungen ausgetauscht werden können, ohne dass die Daten durch Kopieren zwischen den Speicherbereichen der Anwendung und dem Netzwerk-Stack verschoben werden müssen. [TB].

Jeder Port speichert seine Warteschlagen in einer Map. Der erste "Key" ist dabei die Priorität der Warteschlange und der Wert ist dann die Liste mit den Paketen. Die Liste entnimmt die Element nach dem FIFO-Prinzip. Zusätzlich dazu gibt eine identisch aufgebaute Liste mit Mutex. Damit es nicht zu Speicherzugriffsfehlern kommt, muss erst der Zugriff für andere Threads blockiert werden, bevor auf die eigentliche Daten zugegriffen werden kann. Nach dem Zugreifen muss der Mutex unbedingt wieder freigegeben werden, da es sonst zu einer Verklemmung kommt.

Bei Initialisierung des Sockets bekommt der Port die vier Ringe, den Buffer und das "xsk_socket"-Struct zugewiesen. Die Variable "UMEM_fq_initialized" wird dabei auf wahr gesetzt, sollte das nicht geschehen sein, wird der Vorgang abgebrochen und das Switch-Programm beendet sich komplett. Beim Versenden von Paketen bekommt die Standard C-Operation "sendto" den File-Descriptor von dem Struct übergeben.

5.5 Forward Klasse

Die Klasse "Forward" ist eine abstrakte Klasse, die als Schnittstelle für die Weiterleitung von Paketen in einem Netzwerk dient. Die Klasse enthält die Forwarding-Tabelle zum Speicherung von MAC-Adressen und Ports sowie abstrakte Methoden, die in abgeleiteten Klassen implementiert werden müssen, um sicherzustellen, dass Pakete ordnungsgemäß weitergeleitet werden. Der Benutzende kann eine eigene Fordward-Klasse programmieren, um Pakete an den richtigen Port weiterzuleiten, indem er die abstrakte "Forward"-Klasse erbt und die abstrakte Methode "forward" überschreibt. Außerdem gibt es eine weitere statische Klassevariable "lock", die für die Synchronisation von Threads genutzt werden kann und ebenfalls öffentlich zugänglich ist.

Die erste öffentliche, statische Datenmitgliedsvariable der Klasse ist eine Liste von Mac-Adressen. Diese Variable wird verwendet, um eine Zuordnung zwischen MAC-Adressen und Ports herzustellen. Sie speichert die MAC-Adressen als Schlüssel und die entsprechenden Port-Objekte als Werte. Die zweite öffentliche Datenmitgliedsvariable der Klasse ist ein Iterator für die Liste. Mit diesem Iterator kann auf die in der Liste gespeicherten Daten zugegriffen werden.

Die Klasse definiert eine virtuelle Methode "forward", die für das Weiterleiten von Paketen aufgerufen wird und eine reine virtuelle Methode "clone", die für das Erzeugen von Kopien der Klasse genutzt wird.

Zusätzlich besitzt die Klasse "Forward" eine Methode "broadcast", welche dazu dient, ein Paket an alle angeschlossenen Ports eines Switches oder Routers zu senden. Diese Methode nimmt als Parameter das zu sendende Paket, den Empfängerport, die Liste der Senderports sowie die VLAN-ID des Pakets entgegen.

5.5.1 Forward Backward Learning Klasse

Die Klasse “Forward_Backward_Learning“ ist eine konkrete Klasse, die von der abstrakten Klasse “Forward“ abgeleitet ist. Die Klasse implementiert die “Forward_Backward_Learning“-Klasse die abstrakte Methode “forward“ der Basisklasse “Forward“ und fügt der Methode spezifische Funktionalitäten hinzu, um sicherzustellen, dass Pakete ordnungsgemäß weitergeleitet werden. Die Klasse repräsentiert einen klassische Forward–Backward-Learning-Algorithmus, welcher mit Hilfe von Mac-Adressen Pakete weiterleitet. Der Algorithmus arbeitet also auf Schicht zwei des OSI-Models.

Die Methode “forward“ der “Forward_Backward_Learning“-Klasse implementiert die Weiterleitung von Paketen an den richtigen Empfänger-Port. Die Methode nimmt drei Argumente: package ist das zu sendende Paket, receiver ist der Port, auf dem das Paket empfangen wurde, und transmitter ist ein Vektor aller anderen Ports, an die das Paket weitergeleitet werden soll. Die Methode ruft die Package Parsing-Hilfsfunktionen auf, um die Quell- und Ziel-MAC-Adressen sowie das Protokoll und die VLAN-ID aus dem Paket-Header zu extrahieren. Dann wird der gemeinsam genutzte Forwarding-Tabelle mit einem Lock gesperrt, um sicherzustellen, dass der Zugriff atomar ist. Es wird überprüft, ob der Eintrag für die Quell-MAC-Adresse in der Mac-Tabelle vorhanden ist, und wenn nicht, wird der Eintrag mit dem aktuellen receiver-Port erstellt.

Daraufhin wird überprüft, ob der Eintrag für die Ziel-MAC-Adresse in der Mac-Tabelle vorhanden ist. Wenn ja, wird das Paket an den entsprechenden Port weitergeleitet. Wenn das Protokoll ARP ist, wird das Paket an alle Ports (außer dem receiver-Port) über die broadcast-Methode weitergeleitet. Wenn das Protokoll IPV4 ist, wird überprüft, ob das VLAN auf dem Ziel-Port übereinstimmt und das Paket wird nur an diesen Port weitergeleitet. Andernfalls wird das Paket an den receiver-Port weitergeleitet. Wenn das Protokoll nicht ARP oder IPV4 ist, wird das Paket an alle Ports weitergeleitet. Wenn die Ziel-MAC-Adresse nicht in der Forwarding-Tabelle gefunden wird, wird das Paket an alle Ports im Transmitter-Vektor gesendet.

5.6 Classifier Klasse

Die Klasse “Classifier“ definiert eine abstrakte Basisklasse, die als Schnittstelle für alle Klassifikatoren dient. Ein Klassifikator ist eine Einheit, die eingehende Pakete kategorisiert und in die richtige Warteschlange einordnet. Die Klasse “Classifier“ stellt eine grundlegende Struktur zur Implementierung von Klassifikationsalgorithmen bereit, die von verschiedenen Netzwerkkomponenten genutzt werden können, wie beispielsweise Switches oder Router. Der Benutzende kann eine eigene Classifier-Klasse programmieren, um die Pakete der richtigen Warteschlange zuzuordnen. Hierfür muss er die abstrakte “Classifier“-Klasse erben und die abstrakte Methode “classify“ überschreiben.

Die Methode “classify“ ist eine reine virtuelle Methode, die von abgeleiteten Klassen implementiert werden muss. Diese Methode erhält als Parameter den Port, auf dem das Paket eingegangen ist, das Paket selbst und ein Package-Objekt, das weitere Informationen über das Paket enthält.

Die Klasse hat auch eine Methode “clone“, die ebenfalls virtuell ist und in abgeleiteten Klassen implementiert werden muss. Diese Methode erstellt eine Kopie des aktuellen Klassifikators und gibt einen Zeiger (Pointer) auf das neue Objekt zurück.

5.6.1 Classifier_L2 Klasse

Diese Klasse ist die Implementierung eines Klassifizierers namens “Classifier_L2“, der von einer Basisklasse “Classifier“ abgeleitet ist. Die Klasse ist ein Layer zwei Klassifizierer.

Die Methode “classify“ akzeptiert einen Pointer auf ein Port-Objekt, einen Pointer auf ein Paket, das von der Netzwerkkarte empfangen wurde, sowie ein Package-Objekt, das das Paket und seine Metadaten enthält. Die Methode klassifiziert das Paket anhand von Filterregeln und legt es in einer der Queues des Ports ab.

Zunächst werden die Quell- und Ziel-MAC-Adressen des Pakets extrahiert. Dann wird jedes Element in der PrioMac-Liste des Ports durchlaufen. Wenn die MAC-Adresse des Pakets entweder der Quell- oder Zieladresse der Priorisierungsliste entspricht, wird das Paket der vorgegebenen Priorität der entsprechenden Queue des Ports hinzugefügt.

Wenn das Paket ein VLAN-Tag enthält, wird es weiter aufgeteilt und die entsprechende Filterregel aus der PrioEthertype-Liste ausgewählt, basierend auf dem Typ des aufgeteilten Pakets. Wenn das Paket kein VLAN-Tag enthält, wird die entsprechende Filterregel aus der PrioEthertype-Liste basierend auf dem Ethertype-Feld im Header ausgewählt. Wenn keine passende Filterregel gefunden wird, wird das Paket der Queue mit der Priorität eins hinzugefügt, dies entspricht dem “Best Effort Level“ von der Unterteilung des PCP-Feld vom Standard IEEE 802.1Q [iee11].

Die Package-Struktur enthält auch Informationen über die Priorität des Pakets, die basierend auf dem VLAN-Tag mittels des PCP-Feld des Pakets bestimmt wird. Wenn das Paket keine Prioritätsinformationen enthält, wird es der Queue mit der Priorität eins hinzugefügt. Wenn das Paket eine Priorität von eins hat, wird es der Queue mit der Priorität null hinzugefügt. Die beiden Zuordnungen kommen von der Unterteilung des PCP-Feldes vom Standard IEEE 802.1Q [iee11]. Ansonsten wird das Paket der Queue mit der entsprechenden Priorität basierend auf dem PCP-Feld des Pakets hinzugefügt.

In der Methode werden Mutex-Locks verwendet, um kritische Abschnitte des Codes zu schützen, in denen mehrere Threads gleichzeitig auf dieselbe Ressource, wie zum Beispiel auf die Queue zugreifen können. Das pthread_mutex_lock-Systemaufruf wird verwendet, um den Mutex zu sperren, während die Queue geändert wird und pthread_mutex_unlock wird verwendet, um den Mutex zu entsperren, wenn die Queue-Änderungen abgeschlossen sind.

Insgesamt ist die Klasse “Classifier_L2“ ein Klassifizierer, der Pakete anhand ihrer MAC-Adresse und ihres Ether-Typs priorisiert und in die passende Queue auf einem Port einreiht.

5.7 Packet Klasse

Die Klasse Packet ist eine Klasse, die zwei private Variablen enthält: “addr“ und “len“. “addr“ ist ein 64-Bit Ganzzahltyp, der die Adresse des Pakets repräsentiert, während “len“ ein 32-Bit Ganzzahltyp ist, der die Länge des Pakets darstellt. Die Klasse Package wird in der Regel verwendet, um Informationen über ein Paket zu speichern und abzurufen. Durch die Speicherung von Adresse und Länge des Pakets kann ein Empfänger das Paket ordnungsgemäß empfangen, verarbeiten und später weitersenden.

5.8 Package Parsing Klasse

Die Klasse namens Package Parsing beinhaltet verschiedene Funktionen und Methoden, die im Zusammenhang mit Netzwerkpaketen verwendet werden können. Die Klasse ist in verschiedene Abschnitte unterteilt, die jeweils unterschiedliche Aufgaben erfüllen.

Die Funktion "isARPProtocol" überprüft, ob das Protokoll im Ethernet-Header mit ARP (Address Resolution Protocol) übereinstimmt, indem sie den Wert von Protokol mit der Ethernet-Protokollnummer von ARP (ETH_P_ARP) vergleicht. Die Funktion gibt wahr zurück, wenn die Protokolle übereinstimmen, und falsch sollte dies nicht der Fall sein.

Die Funktion "isVlanTagged" überprüft, ob das Paket mit einem VLAN-Tag versehen ist. Die Funktion verwendet die Ethernet-Header-Struktur, um das Ethernet-Protokoll im Header zu extrahieren (h_proto) und vergleicht es mit der VLAN-Protokollnummer (ETH_P_8021Q). Die Funktion gibt wahr zurück, wenn das Paket mit einem VLAN-Tag versehen ist, und falsch sollte kein VLAN-Tag vorhanden sein.

Die Funktion "isEthernetII" überprüft, ob ein Paket einem Ethernet II-Frame entspricht, indem sie das Ethernet-Protokoll im Header extrahiert und prüft, ob dieser Wert größer oder gleich 1536 ist. Ein solcher Wert kennzeichnet einen Ethernet II-Frame. Die Funktion gibt wahr zurück, wenn das Paket einem Ethernet II-Frame entspricht, und falsch andernfalls.

Die Funktion "isVlanEthernetII" ist ähnlich zu "isEthernetII", prüft jedoch auf VLAN-Tagging wie in "isVlanTagged".

Durch Extraktion des Ethernet-Protokolls im Header prüfen die Funktionen "is802_3_LLC" und "isVlan802_3_LLC", ob ein Paket dem 802.3-Standard mit LLC-Protokoll entspricht. Dabei wird geprüft, ob der Wert kleiner 1536 ist, welcher ein LLC-Paket kennzeichnet. Die Funktion gibt wahr zurück, wenn das Paket dem 802.3-Standard mit LLC-Protokoll entspricht, und falsch andernfalls.

Die Funktionen "is802_3_SNAP" und "isVLAN802_3_SNAP" überprüfen, ob das Paket dem 802.3-Standard mit SNAP-Protokoll entspricht, indem sie die 802.3 LLC-Header-Struktur verwenden und die Werte der SSAP, DSAP und Control-Felder prüfen, um sicherzustellen, dass sie dem SNAP-Protokoll entsprechen. Die Funktion gibt wahr zurück, wenn das Paket dem 802.3-Standard mit SNAP-Protokoll entspricht, und falsch andernfalls.

Die Funktion "parseL2" extrahiert den Ethernet-Header aus einem Paket, egal welchen Ethernet-Frame es hat und gibt einige seiner Felder, falls gewünscht, auf der Konsole aus.

Die Methode "getProtocol" analysiert das gegebene Netzwerkpaket und gibt das Protokoll zurück, dass es verwendet. Für die Klassifizierung des Ethernet-Frame werden verschiedene Felder ausgewertet. Der Entscheidungsbaum ist auch in Abbildung 5.3 zu sehen. Als erstes werden die Daten des Paketes auf das Ethernet II Struct übertragen. Handelt es sich bei dem Typ um einen VLAN-Tag (0x8100), werden bei den weiteren Entscheidungen die VLAN-Structs verwendet, was die weiteren Entscheidungen jedoch nicht beeinflusst. Es werden nur die Structs mit VLAN verwendet. Im Framework finden sich die erstellten Structs zu "802.3 LLC", "802.3 LLC VLAN", "802.3 SNAP", "802.3 SNAP VLAN" und "Ethernet II VLAN". Hat der Typwert beim Ethernet II Struct einen Wert über oder gleich 1536, handelt es sich um einen EtherType und das Paket ist ein Ethernet II Frame. Ist der Wert kleiner als 1536 handelt es sich um ein Längen Feld und

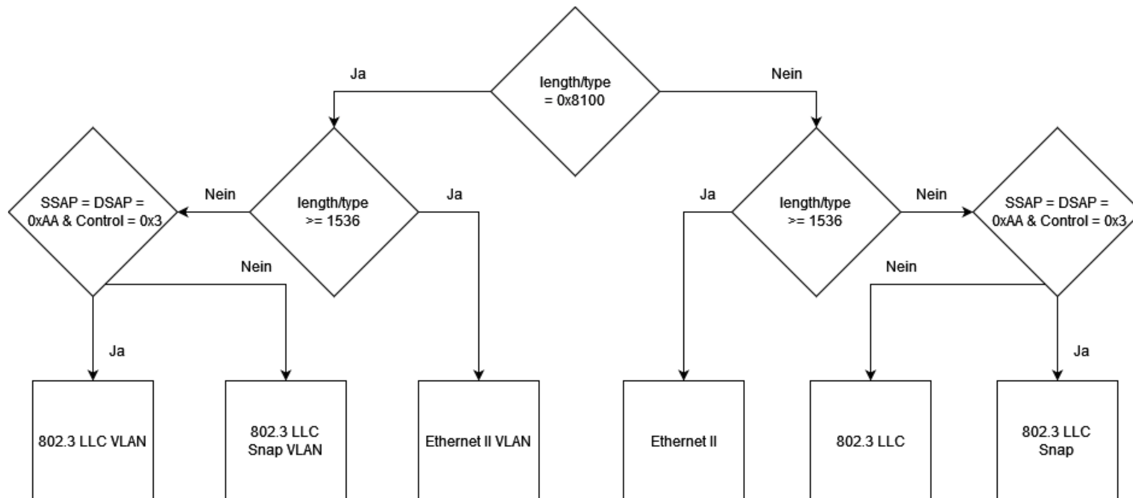

Abbildung 5.3: Ethernet Klassifizierung

somit um einen 802.3 LLC Frame. Die Daten von dem Paket wurden nun auf das 802.3 LLC Struct übertragen. Ist der Wert von SSAP und DASP gleich 0xAA und der Control 0x3, handelt es sich um einen 802.3 SNAP Frame, sonst um einen 802.3 LLC Frame.

Die nächste Methode namens “getPCP“ gibt die Prioritätsstufe des gegebenen Pakets zurück. Wenn das Paket ein VLAN-Paket ist, wird eine weitere Methode “getPCP“ aufgerufen, die die Prioritätsstufe bitweise aus dem verschlagworteten Ethernet-Header extrahiert. Wenn das Paket kein VLAN-Tag hat, wird 0 zurückgegeben.

Die Methoden “getDei“ und “getVLANID“ funktionieren ähnlich wie “getPCP“, indem sie die Drop-Eligibility-Indikation und die VLAN-ID des Pakets zurückgeben, sollte es mit einem VLAN-Tag versehen sein.

Die Methoden “getSourceMac“ und “getDestMac“ extrahieren die Quell- und Ziel-MAC-Adresse aus dem Ethernet-Header des gegebenen Pakets und geben sie als String zurück.

Schließlich gibt es eine Methode namens “parseIPPackge“, die das IP-Paket des gegebenen Netzwerkpakets analysiert und einen Zeiger auf den IP-Header zurückgibt. Die Methode überprüft zuerst, ob das Protokoll des Ethernet-Headers das IP-Protokoll ist. Wenn ja, wird der IP-Header extrahiert und zurückgegeben. Wenn der print-Parameter auf wahr gesetzt ist, wird der Inhalt des IP-Header ausgegeben.

Insgesamt bietet die Klasse Package Parsing verschiedene nützliche Methoden zur Analyse von Netzwerkpaketen und Extrahierung von Informationen aus Ethernet-Headern und IP-Headern.

5.9 Scheduler Klasse

Die Klasse “Scheduler“ definiert eine Schnittstelle zum Scheduling von Paketen, die von einem Netzwerkport gesendet werden sollen. Die Klasse definiert eine abstrakte Basisklasse.

Die Klasse definiert eine reine virtuelle Funktion “schedule“, die von abgeleiteten Klassen implementiert werden muss. Die Funktion nimmt einen Zeiger auf einen Port als Parameter und plant die Warteschlangen entsprechend dem übergebenen Prinzip.

Die Klasse hat auch zwei konkrete Methoden. Die erste Methode ist “round_robin“, bei dem Pakete gemäß dem Packet-by-Packet-Round-Robin an die Netzwerkkarte weitergeleitet werden. Die Funktion nimmt einen Zeiger auf einen Port und die maximale Anzahl der zu sendenden Pakete als Parameter entgegen. Die zweite Methode ist “fifo“, bei der Pakete gemäß dem FIFO-Prinzip an die Netzwerkkarte weitergeleitet werden. Die Funktion nimmt nur einen Zeiger auf einen Port als Parameter.

Abgeleitete Klassen können weitere Methoden definieren, um andere Planungsprinzipien zu implementieren. Die abgeleitete Klassen “scheduler_round_robin“ und “scheduler_fifo“ importieren die von ihrere Elternklasse implementierten Funktionen. Für das eigene Erstellen von Scheduler-Klassen muss lediglich die abstrakte Methode “schedule“ überschrieben werden.

5.9.1 Scheduler SPQ Klasse

Die Klasse ist eine Unterklasse der Klasse “Scheduler“ und wird als “Scheduler_SPQ“ bezeichnet. Scheduler_SPQ implementiert das Scheduling-Verfahren Strict-Priority-Queuing.

Die Funktion “schedule“ ist die Hauptfunktion dieser Klasse. Sie erhält einen Pointer auf ein Objekt der Klasse “Port“ als Argument und verwendet eine Map-Datenstruktur, um die Queues des Ports zu durchlaufen. Die Funktion “schedule“ beginnt damit, einen Pointer auf ein Objekt der Klasse “Port“ als Argument zu erhalten. Sie verwendet dann eine “map“-Datenstruktur, um die Queues des Ports in umgekehrter Reihenfolge zu durchlaufen, beginnend mit der höchsten Queue-Nummer. In jeder Schleifeniteration Funktion, ob die aktuelle Queue nicht leer ist, indem sie die “empty“-Funktion der Queue aufruft. Wenn die Queue nicht leer ist, wird der Inhalt der Queue mit der Funktion “port_tx_burst_one“ auf den Port übertragen. Diese Funktion wird aufgerufen und erhält drei Argumente: den Port-Pointer, eine Referenz auf die aktuelle Queue und die Queue-Nummer. Wenn der Inhalt der aktuellen Queue erfolgreich auf den Port übertragen wurde, setzt die Funktion die Schleifenvariable “iter“ auf den Anfang der Queues des Ports zurück, um sicherzustellen, dass immer die Pakete mit der höchsten Priorität versendet werden. Andernfalls geht die Funktion einfach zur nächsten Queue weiter.

Diese Klasse ist ein Teil eines Netzwerksystems und für das Scheduling von Datenpaketen in verschiedenen Queues verantwortlich.

5.9.2 Scheduler ST Klasse

Die Klasse Scheduler_ST ist eine Subklasse der abstrakten Klasse Scheduler. Sie implementiert eine einfache “Scheduled-Traffic-Logik“. Ein Scheduler_ST Objekt hat einen round_robin_index, einen “realtime“ und einen “best_efford“ Timer und eine “cycle“ Zeit, die sich aus der Summe von “realtime“ und “best_efford“ ergibt.

Der Konstruktor Scheduler_ST initialisiert den round_robin_index mit null und setzt “realtime“, “best_efford“ und “cycle“ auf die übergebenen Parameterwerte.

Die Methode “schedule“ plant den Paketversand auf einem gegebenen Port p. Zunächst wird die aktuelle Systemzeit abgerufen und in “now“ gespeichert. Dann wird “time_left“ berechnet, indem “cycle“ von “now“ subtrahiert wird, um zu ermitteln, wie viel Zeit noch bis zur nächsten Planungsrunde verbleibt. Die Schleife überprüft, ob die verbleibende Zeit größer als null ist. Innerhalb der Schleife wird erneut die aktuelle Zeit abgerufen und in “now“ gespeichert, um zu überprüfen, ob sich die verbleibende Zeit seit dem letzten Abrufen geändert hat. Wenn von der Zykluszeit des echtzeitfähigen Netzwerkverkehr noch Zeit übrig ist, wird die Warteschlange mit der höchsten Priorität ausgewählt und wenn diese nicht leer ist, wird die “port_tx_burst_one“ Methode aufgerufen, um das Paket zu senden. Wenn keine Zykluszeit mehr für den Echtzeitfähigen Netzwerkverkehr ist, kann der Best-Efford-Verkehr senden. Hier wird die Warteschlange mit dem nächsten Index “round_robin_index“ ausgewählt und wenn diese nicht leer ist, wird berechnet, wie viel Zeit benötigt wird, um das Paket zu senden. Wenn die verbleibende Zeit, um das Paket zu senden, größer als null ist, wird “port_TX_burst_one“ aufgerufen, um das Paket zu senden und “round_robin_index“ wird um eins erhöht. Falls die Warteschlange leer ist, wird “round_robin_index“ ebenfalls um eins erhöht. Bei dem Scheduling-Algorithmus handelt es sich um Length-aware Scheduling, bei dem die Länge oder Dauer von Aufgaben berücksichtigt wird, wenn diese geplant werden. Der Echzeitfähigverkehr kann dabei über den Zyklus senden, während beim Best-Fffort immer geprüft wird, ob das Paket noch versendet werden darf. Der beschriebene Algorithmus ist in Listing 5.1 zu sehen.

Insgesamt implementiert die Klasse “Scheduler_ST“ einen einfachen “Scheduled-Traffic“, die in der Lage ist, Pakete auf einem Port zu priorisieren und sie nach einer bestimmten Zeitplanungsrunde zu senden.

5.9.3 Scheduler WFQ Klasse

Die Klasse ist ein Scheduler mit dem Weighted Fair Queueing (WFQ) Algorithmus, der auf der Klasse namens “Scheduler“ basiert. Die Klasse hat einen Standardkonstruktor und einen parametrisierten Konstruktor. Die Funktion “schedule“ ist verantwortlich für die Planung des Sendens von Paketen über den Port.

Der Standardkonstruktor initialisiert eine Map namens “weights“, die verwendet wird um Gewichte für die unterschiedlichen Queues festzulegen. Das Gewicht bestimmt die Anzahl der Pakete, welche pro Runde versendet werden dürfen. In diesem Fall werden die Queues mit dem gleichem Gewicht initialisiert, die in der Map mit einer for-Schleife gesetzt werden.

Der parametrisierte Konstruktor ruft den Standardkonstruktor auf und setzt Gewichte für bestimmte Queues, die vom Benutzenden als Parameter übergeben wurden. Es werden alle Gewichte durchlaufen und in der Map aktualisiert.

Die Funktion “schedule“ ist der wichtigste Teil des Schedulers und durchläuft alle Queues auf dem Port. Für jede Queue wird überprüft, ob sie nicht leer ist. Wenn sie nicht leer ist, wird die Anzahl der Pakete berechnet, die gesendet werden sollen, basierend auf dem Gewicht, das der Queue zugeordnet ist. Es wird ein “Burst“ von Paketen aus der Queue an den Port gesendet. Die Anzahl der Pakete, die gesendet werden sollen, wird durch die Anzahl von Schleifendurchläufen festgelegt, wobei jede Schleife ein Paket sendet.

```
void Scheduler_ST::schedule(Port *p)
{
  // current time
  auto now = std::chrono::system_clock::now().time_since_epoch();
  // current time + cycle = maximum time for the cycle
  auto time_left = std::chrono::duration_cast<std::chrono::microseconds>(now + this->cycle);
  // remaining time for the cycle
  auto microseconds = std::chrono::duration_cast<std::chrono::microseconds>(time_left - now);
  while (microseconds.count() > 0)
  {
    // current time
    now = std::chrono::system_clock::now().time_since_epoch();
    // remaining time for the cycle
    microseconds = std::chrono::duration_cast<std::chrono::microseconds>(time_left - now);
    // remaining time for real_time traffic
    auto burst_real_time = std::chrono::duration_cast<std::chrono::microseconds>(microseconds
- this->best_effort);
    if (burst_real_time.count() > 0)
    {
      int queue = p->getQueues().size() - 1;
      std::queue<Packet> &v = p->getQueues().at(queue);
      if (!v.empty())
      {
        Network_Switch::port_tx_burst_one(p, v, queue);
      }
    }
    else
    {
      std::queue<Packet> &v = p->getQueues().at(this->round_robin_index);
      if (!v.empty())
      {
        // protection window
        // byte * 8 bit/byte / mb * 8 mbit/mb * 10^6 bit/s = s
        auto protection_windows = std::chrono::microseconds(v.back().getLength() * 8) / (p->
getSpeed() * 8 * 10 ^ 6);
        auto time_left_to_send = std::chrono::duration_cast<std::chrono::microseconds>(
microseconds - protection_windows);
        // enough time to send the packet
        if (microseconds.count() > 0)
        {
          Network_Switch::port_tx_burst_one(p, v, this->round_robin_index);
          this->round_robin_index = (this->round_robin_index + 1) % (p->getQueues().size() -
1);
        }
      }
      else
      {
        this->round_robin_index = (this->round_robin_index + 1) % (p->getQueues().size() - 1);
      }
    }
  }
}
```

Listing 5.1: Scheduled Traffic

Insgesamt ist die Klasse Scheduler_WFQ eine Implementierung des “Weighted Fair Queueing Algorithmus“, der in der Lage ist, den Datenverkehr von mehreren Queues fair und effizient zu behandeln, wobei die Gewichte an die Anforderungen der Nutzenden angepasst werden können.

5.10 Thread Klasse

Die Klasse “Thread“ definiert eine abstrakte Basisklasse für Thread-spezifische Daten. Die Klasse enthält private Variablen und Funktionen, sowie öffentliche Funktionen zur Verwaltung dieser Variablen.

Innerhalb der privaten Sektion der Klasse befinden sich drei Komponenten: Die Variable “cpu_core_id“ vom Typ “uint32“, welche die ID des CPU-Kerns speichert, auf dem der Thread ausgeführt werden soll, die Variable “shouldJoin“ vom Typ bool, welche angibt, ob der Thread beendet werden soll oder nicht, und die Methode “init“, welche bei der Erstellung eines Objekts der Klasse aufgerufen wird, um die Variablen zu initialisieren.

Die Methode “join“ setzt die Variable “shouldJoin“ auf wahr, was bedeutet, dass der Thread beendet werden soll. Die Methode “isJoining“ gibt den Wert von “shouldJoin“ zurück.

Die abstracte Methode “execute“ ist eine rein virtuelle Funktion, welche von abgeleiteten Klassen implementiert werden muss. Diese Funktion definiert, was im Thread ausgeführt werden soll.

Zusammenfassend kann gesagt werden, dass diese Klasse als Grundlage für Thread-spezifische Daten dient und Funktionen zur Verwaltung dieser Daten enthält. Sie bietet eine einfache Möglichkeit, Threads zu steuern und zu überwachen.

5.10.1 Thread RX Klasse

Die Klasse “Thread_RX“ repräsentiert einen Datenverarbeitungs-Thread, der Daten von einem Port empfängt.

Die Klasse verfügt über zwei Konstruktoren. Der erste Konstruktor nimmt einen Port als Parameter und ruft den Konstruktor der übergeordneten Klasse Thread auf, während der zweite Konstruktor auch eine Variable “cpu_core_id“ erhält und sie an den Konstruktor der übergeordneten Klasse weitergibt.

Die Methode “getTransmitter“ gibt eine Referenz auf einen Vektor von Port zurück. Die Methode “setTransmitter“ nimmt einen Vektor von Port als Parameter und ruft die Methode “addTransmitter“ für jeden Port im Vektor auf, welche dem Port dem Transmitter-Vektor hinzu, sofern der Port nicht das gleiche Interface wie der Empfänger hat. Die Methode “getReceiver“ gibt den Empfangs-Port zurück.

Die Methode “execute“ führt die Thread-Ausführung durch. Diese hat einen Parameter vom Typ “Thread_Safe_Queue<Package>“, der eine Referenz auf eine sichere Warteschlange von Paketen darstellt. Sie legt die CPU-Affinität auf die vom Thread zugewiesene CPU-Core-ID fest. Die Methode ruft dann die “port_rx_burst“-Funktion auf, um Daten vom angegebenen Port zu empfangen, solange der Thread nicht zum Beenden markiert wurde.

Die Klasse ist zusammenfassend eine Klasse welche für den Empfangs-Port die Aufgabe des Empfangens von Paketen übernimmt.

5.10.2 Thread Forward Klasse

Diese Klasse namens “Thread_Forward“ ist eine Klasse, die von der Klasse “Thread“ abgeleitet ist und zwei Parameter enthält: ein Objekt der Klasse “Thread_RX“ und ein Objekt der Klasse “Forward“.

Die Konstruktoren der Klasse sind überladen und akzeptieren entweder zwei oder drei Argumente: das erste ist ein Zeiger auf ein Objekt der Klasse “Thread_RX“, das zweite ist ein Zeiger auf ein Objekt der Klasse “Forward“ und das dritte ist eine 32-Bit-Unsigned-Integer-Variable, die die ID des CPU-Kerns enthält.

Die Funktion “execute“ dieser Klasse hat einen Parameter, der eine Referenz auf eine “Thread_Safe_Queue“ des Typs “Package“ ist. Diese Funktion führt eine Schleife aus, die so lange ausgeführt wird, bis das “isJoining“-Flag des “Thread“-Objekts gesetzt ist. Innerhalb der Schleife wird die Größe der Queue ermittelt und dann wird jedes “Package“ aus der Queue entfernt und mit der Funktion “forward“ des “Forward“-Objekts weitergeleitet, wobei auch das Empfänger- und Sender-Objekt aus dem “Thread_RX“-Objekt verwendet wird.

Die Funktion “print“ der Klasse gibt Informationen über das Objekt aus, insbesondere die CPU-Kern-ID und die Schnittstelle (Interface) des Empfängers.

Die Klasse bzw. der Thread übernimmt die Aufgabe des Weiterleiten an den Sendeports und startet die Klassifizierung des Paketes, um dieses in die richtige Warteschlange einzuordnen.

5.10.3 Thread TX Klasse

Die Klasse ist eine Unterklasse der Klasse “Thread“ und wird als “Thread_TX“ bezeichnet. Die Aufgabe der Klasse definiert einen Thread, der zum Senden von Daten über einen Port und Scheduler verwendet wird.

Die Klasse hat zwei Konstruktoren. Der erste Konstruktor hat zwei Parameter vom Typ “Port*“ und “Scheduler*“, der zweite Konstruktor hat drei Parameter vom gleichen Typ wie der erste, aber auch einen Parameter mit der Bezeichnung “cpu_core_id“.

Die Klasse enthält auch zwei Methoden, “getSender“ und “execute“. Die “getSender“-Methode gibt einen Pointer auf einen “Port“ zurück, der als Parameter an den Konstruktor übergeben wurde. Die “execute“-Methode setzt zuerst den Thread an den spezifischen CPU-Kern, der in der Variablen “cpu_cores“ gespeichert ist. In einer While-Schleife wird überprüft, ob der Thread beendet werden soll oder nicht, indem die Methode “isJoining“ aufgerufen wird. Diese Methode gibt einen booleschen Wert zurück, der angibt, ob der Thread beendet werden soll oder nicht. In jeder Schleifeniteration wird der “scheduler“ aufgerufen und der “sender“ als Parameter übergeben.

Zusammenfassend bindet die Methode “execute“ den Thread an einen bestimmten CPU-Kern und plant den Sender in den Scheduler ein, solange der Thread nicht beendet werden soll. Der Thread hat die Aufgabe, die von ihm erhaltenen Pakete zu versenden.

5.10.4 Thread Safe Queue Klasse

Die Klasse “Thread_Safe_Queue“ ist eine Implementierung einer threadsicheren Warteschlange, die von mehreren Threads gleichzeitig verwendet werden kann. Diese Klasse bietet eine sicherere Möglichkeit, Daten zwischen Threads zu übertragen, ohne dass es zu “Race Conditions“ oder ähnlichen Problemen kommt.

Die Klasse ist als “Template“ definiert und kann mit jedem Datentyp verwendet werden. Sie enthält eine private Instanzvariable “m_queue“ des Typs “queue<T>“, die die eigentliche Warteschlange darstellt. “m_mutex“ ist eine Instanzvariable des Typs mutex, die für die Thread-Synchronisation verwendet wird, um sicherzustellen, dass nur ein Thread zu einem bestimmten Zeitpunkt auf die Warteschlange zugreifen kann. “m_cond“ ist eine Instanzvariable des Typs “condition_variable“, die für das Signalisieren von Threads verwendet wird.

Die Methode “push“ verwendet einen “unique_lock“ mit dem Mutex, um sicherzustellen, dass nur ein Thread gleichzeitig auf die Warteschlange zugreift. Sobald ein Element erfolgreich hinzugefügt wurde, wird ein Signal an einen wartenden Thread gesendet, indem die “notify_one“-Methode aufgerufen wird.

Die Methode “pop“ verwendet ebenfalls einen “unique_lock“ mit dem Mutex, blockiert aber bis die Warteschlange mindestens ein Element enthält, indem sie auf ein Signal von der “condition_variable“ wartet. Sobald ein Element verfügbar ist, wird es aus der Warteschlange entfernt und zurückgegeben.

Die Methoden “size“ und “ empty“ sind einfach und verwenden auch den Mutex um sicherzustellen, dass keine anderen Threads gleichzeitig auf die Warteschlange zugreifen können, während sie die Größe oder den Status der Warteschlange abfragen.

Zusammenfassend kann gesagt werden, dass diese Klasse eine threadsichern Warteschlange implementiert.

5.11 Network-Switch Klasse

Die Klasse Network-Switch ist eine Implementierung eines Netzwerk-Switches und dient zur Weiterleitung von Netzwerkverkehr zwischen verschiedenen Ports. Sie nutzt eine Instanz der Klasse Port zur Interaktion mit den physikalischen Netzwerk-Ports, eine Instanz der Klasse “Classifier“ zur Klassifizierung des Netzwerkverkehrs und eine Instanz der Klasse “Forward“ zur Weiterleitung des Verkehrs an den entsprechenden Port. Es gibt auch eine Instanz der Klasse Scheduler um zu entscheiden, welches Paket als nächstes gesendet wird. Die Klasse “Network-Switch“ verfügt auch über ein Flag für die Erfassung von Statistiken und ein Flag für die Protokollierung dieser Statistiken in einer Datei.
Die Klasse wurde durch Kopieren oder Modifizieren der Methoden und Structs aus der C-Datei “AF_XDP-forwarding“ [Int22] der “bpf-examples“ erstellt.

Der Konstruktor der Klasse “Network-Switch“ initialisiert alle notwendigen Member-Variablen der Klasse. Der Konstruktor nimmt eine Referenz auf einen Vektor von Port-Objekten, einen Zeiger auf eine Classifier-Instanz, einen Zeiger auf eine Forward-Instanz, einen Zeiger auf eine

Scheduler-Instanz, ein Flag für die Statistiken und ein Flag für die Protokollierung dieser Statistiken in einer Datei als Parameter. Die Methode ruft dann die private Methode “setup“ auf, um die Ports und den Buffer-Pool für den Netzwerkverkehr zu initialisieren.

Diese Methode “setup“ initialisiert mehrere Ports (Eingangs- und Ausgangsports) für eine Netzwerk-Kommunikation. Der Code verwendet einige benutzerdefinierte Klassen und Funktionen, die in anderen Teilen des Programms definiert sind. Als Erstes wird einen Pufferpool für die Ports durch Aufrufen einer Funktion namens “Network_Switch_Buffer::bpool_init“ initialisiert. Wenn die Initialisierung fehlschlägt, setzt die Methode “error“ auf eins und beendet die Ausführung. Im nächsten Schritt wird für jeden Eingangsport der Pufferpool gesetzt und jedem Port eine Portnummer zugewiesen. Außerdem wird ein Klassifizierer für jeden Eingangsport festgelegt und die Anzahl der empfangenen und gesendeten Pakete wird auf null gesetzt. Schließlich wird die Funktion “port_init“ aufgerufen, um den Port zu initialisieren. Wenn die Initialisierung fehlschlägt, wird die Methode beendet und “error“ auf eins gesetzt. Für den Empfang von Daten wird für jeden Eingangsport ein Thread und ein Thread für das Weiterleiten von Daten (Forwarding) von jedem Empfangsthread zu einem Ausgangsport erstellt. Der Empfangsthread wird einem bestimmten CPU-Kern zugewiesen, während der Forwarding-Thread einem anderen zugewiesen wird. Zur “port_tx“-Liste wird jeder Eingangsport hinzugefügt. Diese Liste wird später zum Entfernen von Duplikaten und zur Bestimmung des Ports mit der niedrigsten Warteschlange verwendet. Es werden aus der Liste alle Duplikate entfernt und diese behält nur den Port mit der niedrigsten Warteschlange, falls die Liste mehrere Einträge mehrerer Ports mit dem gleichen Interface aber unterschiedlichen Warteschlangen enthält.

Die Methode “startThreads“ initialisiert und startet die Threads für das Empfangen, Weiterleiten und Senden von Paketen über die Netzwerkports. Zu Beginn wird überprüft, ob die Statistiken in eine Datei geschrieben werden sollen und ob beim Erstellen der Ports ein Fehler aufgetreten ist. Danach wird für jeden Empfangsthread ein passender Senderthread zugeordnet. Wenn ein Empfangsthread über einen Trunk-Port (ein Port, der mehrere VLANs unterstützt) läuft, wird ihm der einzige Senderthread zugewiesen, der an diesem Trunk-Port angehängt ist. Andernfalls wird für jeden Senderport und jedes VLAN, das der Empfangsthread unterstützt, ein passender Senderthread ausgewählt. Für jeden Empfang- und Forwardsthread wird außerdem eine Thread-Safe-Queue erstellt. Dann werden die Threads für das Empfangen und Weiterleiten von Paketen für jeden Empfangs- und Senderthread gestartet. Der Start eines Empfangs-Threads ruft “Thread_Rx::execute“ auf und der Start eines Sender-Threads ruft “Thread_Forward::execute“ auf. Danach werden Threads für das Senden von Paketen für jeden Senderport gestartet. Der Start eines Sendethreads ruft “Thread_Tx::execute“ auf. Nachdem alle Threads erfolgreich erstellt wurden, wird die Überwachung der Statistiken für die Ports gestartet. Dazu wird ein Signal-Handler für die Signale SIGINT, SIGTERM und SIGABRT gesetzt und eine Schleife gestartet, die Statistiken für jeden Port in regelmäßigen Abständen ausgibt. Wenn “stats“ auf wahr gesetzt ist, werden die Statistiken über die Standardausgabe ausgegeben, andernfalls werden sie in eine Datei geschrieben. Die Schleife läuft so lange, bis das “quit-Flag“ auf wahr gesetzt wird. Wenn das Flag gesetzt ist, werden die Threads freigegeben und die Methode endet.

Die Methode “port_init“ initialisiert einen einzelnen Netzwerk-Port. Die Methode beginnt damit, die Größe des UMEM-FQ (User Memory Fill Queue) des Ports zu ermitteln. Diese wird benötigt, um die Puffer des Ports entsprechend zu füllen. Anschließend initialisiert die Methode den Puffer-Cache des Ports und überprüft erfolgreiches Gelingen. Als Nächstes wird ein XSK-Socket (eXpress Kernel Socket) erstellt, indem “xsk_socket__create_shared“ aufgerufen wird. Dieser Socket ist ein

Linux-Kernel-Objekt, das eine Netzwerkverbindung zu einem bestimmten Netzwerkgerät bereitstellt. Schließlich wird der “User Memory Fill Queue“ (UMEM-FQ) des Ports konfiguriert, indem Puffer aus dem Puffer-Cache entnommen und in den UMEM-FQ eingefügt werden. Die Methode verwendet dabei Funktionen wie “xsk_ring_prod__reserve“ und “xsk_ring_prod__submit“ aus der XSK-API. Wenn die Port-Initialisierung erfolgreich ist, wird die Methode “umemFqIsDefinded“ aufgerufen, um anzuzeigen, dass die Initialisierung des UMEM-FQ des Ports abgeschlossen ist. Andernfalls wird port_free aufgerufen, um den Port und alle damit verbundenen Ressourcen freizugeben.

Die Methode “port_rx_burst“ ist eine Funktion, die einen Port “p“ und eine Warteschlange “queue“ als Eingabeparameter annimmt und eine 32-Bit-Integer als Ausgabe zurückgibt. Die Funktion beginnt damit, eine lokale Variable “n_pkts“ mit dem Wert “MAX_BURST_RX“ zu initialisieren, was die maximale Anzahl von Paketen ist, die in einem Burst empfangen werden kann. Dann wird die Methode “Network_Switch_Buffer::bcache_cons_check“ aufgerufen, um zu prüfen, ob der Puffercache genügend Pakete zum Empfangen bereit hat und die Anzahl der empfangenen Pakete wird auf “n_pkts“ aktualisiert. Wenn “n_pkts“ gleich Null ist, gibt die Funktion 0 zurück, da es keine Pakete zum Empfangen gibt. Als nächstes liest die Funktion den Index des Empfangsringpuffers (RX) mithilfe der Methode “xsk_ring_cons__peek“ aus. Wenn die Anzahl der Pakete, die im RX verfügbar sind gleich NULL ist, prüft die Funktion, ob der RX-Ring aufgeweckt werden muss, um neue Pakete zu empfangen. Wenn ja, wird eine Poll-Operation ausgeführt, um das Socket zu überwachen, bis neue Pakete verfügbar sind und die Funktion gibt null zurück. Wenn es Pakete zum Empfangen gibt, liest die Funktion die Adressen der Einträge im RX und speichert sie in einem Package-Objekt, das der Warteschlange “queue“ hinzugefügt wird. Nachdem alle Pakete aus dem RX gelesen wurden, gibt die Funktion sie frei, indem sie die Methode “xsk_ring_cons__release“ aufruft. Als nächstes reserviert die Funktion Speicherplatz im User Memory (UMEM) mit der Methode “xsk_ring_prod__reserve“, um die empfangenen Pakete zu speichern. Wenn die Anzahl der reservierten Einträge gleich “n_pkts“ ist, wird die Schleife beendet. Wenn das UMEM-FQ aufgeweckt werden muss, um Speicherplatz zu reservieren, wird eine Poll-Operation ausgeführt, um das Socket zu überwachen, bis genügend Speicherplatz verfügbar ist. Sobald genügend Speicherplatz im UMEM reserviert wurde, füllt die Funktion die Adressen der Pakete in das UMEM-FQ mit der Methode “xsk_ring_prod__fill_addr“. Dann werden alle Pakete an das UMEM-FQ übermittelt, indem die Methode “xsk_ring_prod__submit“ aufgerufen wird. Schließlich wird die Anzahl der empfangenen Pakete “n_pkts“ zurückgegeben und die Methode “addReceivedPackages“ des Ports wird aufgerufen, um die Anzahl der empfangenen Pakete zu aktualisieren.

Die “port_tx_burst_one“ Methode dient dazu, ein einzelnes Paket aus einer Warteschlange von Paketen in den Sendepuffer zu schreiben und anschließend auf den Netzwerkadapter zu übertragen. Die Methode nimmt als Parameter einen Port “p“, eine Warteschlange von Paketen “Packages“ und eine Queue-Nummer “queue_number“. Zunächst wird überprüft, ob die Warteschlange leer ist, in diesem Fall wird die Methode abgebrochen. Ansonsten wird versucht, Speicherplatz im Sendepuffer (TX-Ring) zu reservieren, um das Paket hineinzuschreiben. Wenn kein Speicherplatz verfügbar ist, wird geprüft, ob der TX-Ring aufgeweckt werden muss, um die Übertragung fortzusetzen. Wenn der TX-Ring bereits aufgeweckt wurde und immer noch kein Speicherplatz verfügbar ist, wird die Methode beendet. Wenn genügend Speicherplatz verfügbar ist, wird das Paket aus der Warteschlange entfernt und in den TX-Ring geschrieben. Anschließend wird das Paket dem TX-Ring übergeben, und es wird erneut geprüft, ob der TX-Ring aufgeweckt werden muss, um weitere Übertragungen zu ermöglichen. Als Nächstes liest die Methode die Indexeinträge aus dem UMEM-Completion-Ring um festzustellen, ob das Paket erfolgreich gesendet wurde. Wenn ja, wird das Paket aus dem UMEM (User-space Memory) freigegeben, und die Methode endet.

Die Klasse beinhaltet die Methode “port_tx_burst_max“ arbeitet fast genauso wie die Methode “port_TX_burst_one“. Unterschied zwischen beiden Methode ist, dass zunächst so viele Pakete aus dem UMEM freigegeben werden wie möglich. Ein weiterer Unterschied ist, dass zuerst die versendeten Pakete aus dem CL-Ring und dem UMEM entfernt werden. Es wird hier nicht auf das Versenden der Pakete aktiv gewartet.

Die Methode “freeAF“ wird aufgerufen, wenn das Programm beendet wird. Sie gibt eine Meldung auf der Konsole aus, signalisiert den Threads dass sie gestoppt werden sollen, wartet darauf dass die Threads beendet werden, gibt die Ports frei, löscht die Puffer im Speicher und entfernt das XDP-Programm aus dem Kernel. Die Methode “remove_xdp_program“ entfernt das XDP-Programm aus dem Kernel für alle Empfangs-Ports, die von der Socket-Klasse geöffnet wurden. Die Methode “port_free“ wird verwendet, um den Speicher für einen gegebenen Port freizugeben. Sie löscht den Socket für den Port und gibt den Speicher für den Puffer frei, der für den Port verwendet wird.

Die Klasse “Network-Switch“ ist eine Klasse, die mehrere Ports für die Netzwerk-Kommunikation initialisiert. Sie verwendet benutzerdefinierte Klassen und Funktionen, um den Pufferpool zu initialisieren, Eingangs- und Ausgangsports zu konfigurieren, Threads für den Datenempfang und das Weiterleiten einzurichten und Duplikate in der Portliste zu entfernen.

6 Implementierung

Dieses Kapitel der Arbeit beschäftigt sich mit den Implementierungsdetails des Switches und einer beispielhaften Implementierung (Proof-of-Concept), die auch für die Messungen in Kapitel 7 verwendet wird. In Abschnitt 6.1 wird der Aufbau demonstriert, welcher später für die Auswertung benötigt wird. Abschnitt 6.2 zeigt die Grundlage wie der Switch konfiguriert wird und erweitert werden kann.

6.1 Aufbau des Testsystems

Der Aufbau des Testsystems des Switches ist in Abbildung 6.1 zu finden. In dem PC gibt es zwei Netzwerkkarten mit jeweils vier Ports, in Summe werden sechs Stück benötigt. Dem Software-Switch sind hiervon zwei Ports zugeordnet. Für den Server und dem Client werden jeweils ein Port benötigt. Zusätzlich gibt es in dem Aufbau ein Test Access Points (TAP).
Ein TAP ist ein Gerät, mit dem eine exakte Kopie des gesamten Datenverkehrs zwischen zwei Endpunkten in einem Netz erstellt wird. Das Monitoring der Pakete kann so während des Live-Betriebes geschehen. Es gibt hierfür einen Eingangs- und Ausgangsport für die eigentliche Übertragung, sowie einen Port um von A nach B und umgekehrt die übertragenen Pakete zu empfangen.
Die beiden Überwachungsports werden ebenfalls an die Netzwerkkarte angeschlossen.
Besonderheit an der Intel 4-Port i350 Netzwerkkarte ist, dass die Übertragungsgeschwindigkeit zwischen 10 mb/s, 100 mb/s und 1000 mb/s eingestellt werden kann. Pakete, die diese Netzwerkkarte erreichen werden mit einem "Hardware-Zeitstempel" versehen, sodass die genaue Ankunftszeit ausgelesen werden kann. Der Zeitstempel erfolgt in der Hardware auf der Grundlage einer Uhr in der Netzwerkkarte. In der Netzwerkkarte hat jeder Port eine eigene Uhr, jedoch laufen die Uhren in der Regel nicht synchron. Das Programm "phc2sys" dient dazu, zwei Uhren im System miteinander zu synchronisieren. Sein primärer Anwendungszweck besteht darin, die Systemuhr mit einer PTP-Hardwareuhr (PHC) zu synchronisieren [die].

6.2 Konfiguration des Testsystems

Sollte ein ähnlicher Aufbau wie in Abschnitt 6.1 genutzt werden, in welchem Server, Switch und Empfänger auf einem Rechner laufen sollen, müssen "Namespaces" konfiguriert werden. Ohne diese Maßnahme würde der Kernel die verschickten Pakete nie über die Leitung versenden, sondern intern verteilen.
Die Funktion der Namespaces im Linux-Kernel besteht darin, Kernel-Ressourcen so aufzuteilen, dass verschiedene Gruppen von Prozessen unterschiedliche Gruppen von Ressourcen sehen. Ein Namespace wird für eine Gruppe von Ressourcen und Prozessen verwendet, aber diese Namespaces

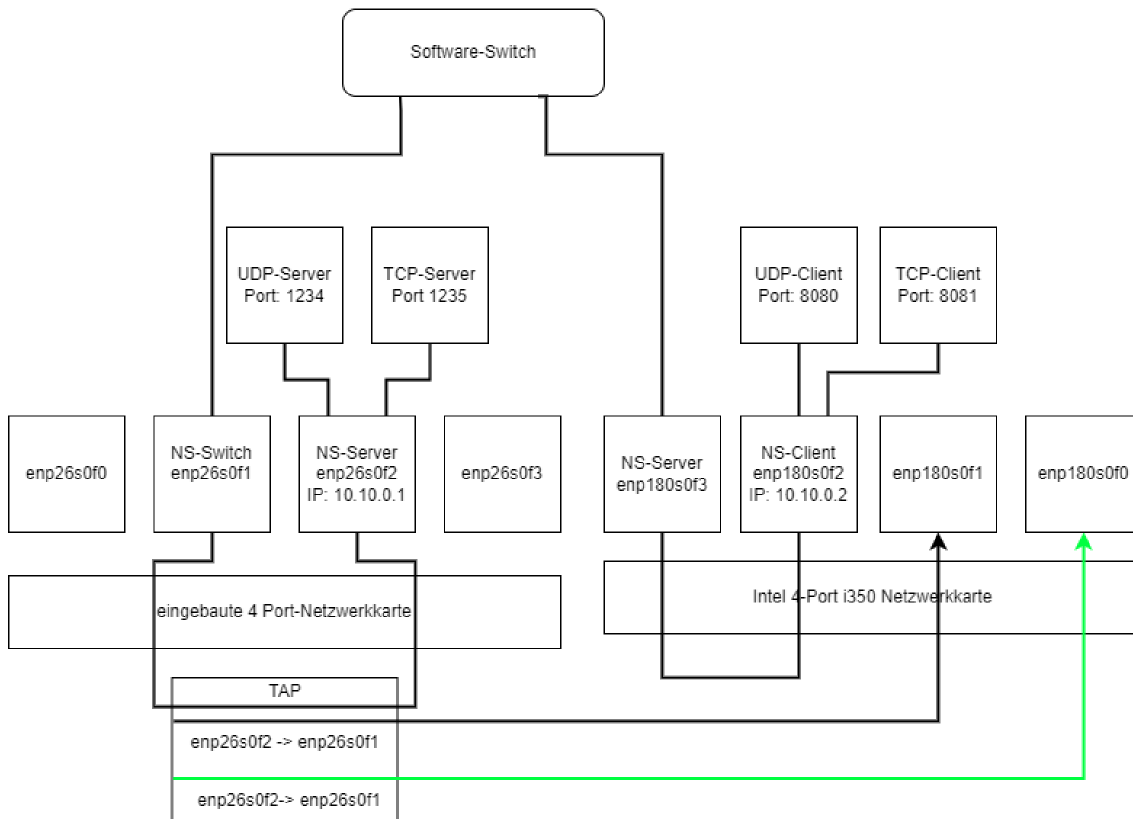

Abbildung 6.1: Aufbau

können auf verschiedene Ressourcen verweisen, die in mehreren Räumen existieren. Einige Beispiele für solche Ressourcen sind Prozess-IDs, Host-Namen, Benutzer-IDs, Dateinamen und Namen im Zusammenhang mit dem Netzzugang und der Kommunikation zwischen Prozessen [The20].

Normalerweise weist Linux bei der Initialisierung einer Netzwerkkarte pro Kern ein Paar von RX- und TX-Warteschlangen zu. In einem System mit 8 Kernen werden die Warteschlangen-IDs von null bis sieben zugewiesen, wobei jede Warteschlange einem Kern zugeordnet wird. Wenn die Klasse Port initialisiert wird, wird eine Warteschlange benötigt, an die man sich binden will. Es wird auf dem Socket nur Verkehr empfangen, der an diese Warteschlange gerichtet ist. Beispielsweise wird der Port an die Warteschlange null gebunden, empfängt der Port keinen Verkehr, der an die Warteschlangen eins bis sieben gerichtet ist. Es besteht die Möglichkeit, dass der Verkehr dennoch gesehen werden kann, aber normalerweise wird er in einer der Warteschlangen landen, wie zuvor angegeben. Mittels des Befehls “sudo ethtool -L <interface> combined 1“ kann die Anzahl der Warteschlangen auf eine reduziert werden, sodass der ganze Netzwerkverkehr an den Switch gesendet wird [TB].

Für den Aufbau des Switches werden verschiedene Komponenten benötigt, welche der Benutzende angeben muss. In Listing 6.1 sieht man beispielhaft eine Konfiguration mit allen Komponenten. Der Switch benötigt zuerst zwei Ports, auf welchen dieser empfangen und senden kann. Weitere wichtige Komponenten sind der “Forward“ und der “Classifier“, welche die Pakete zu dem vorgesehenen Port und Warteschlangen weiterleiten. Als letzte wichtige Klasse wird die Scheduler Klasse benötigt, welche bestimmt, wie die Pakete versendet werden. Sollte im Port acht Warteschlangen

konfiguriert worden sein und ein FIFO-Scheduler benutzt werden, werden die Warteschlangen eins bis acht nie entleert werden. Die Software_Switch Klasse bekommt alle Konifgurationsklasse, zusätzlich kann bestimmt werden, ob die Anzahl an Paketen auf der Konsole angezeigt und oder in eine CSV-Datei geschrieben werden. Als letzten Schritt werden alle Threads für den Switch gestartet.

```
int main(int argc, char **argv)
{
  Forward *forward = new Forward_Backward_Learning();
  Classifier *classifier = new Classifier_L2();
  Scheduler *scheduler = new Scheduler_FIFO();
  std::vector<Port *> ports;
  ports.push_back(new Port("enp180s0f3", 0, 1000, 1));
  ports.push_back(new Port("enp26s0f1", 0, 1000, 1));
  Network_Switch socket(ports, classifier, forward, scheduler, false, true);
  socket.startThreads();
}
```

Listing 6.1: Switch Konfiguration für Fifo-Scheduling

7 Auswertung

Der Software-Switch wurde auf seinen Durchsatz und die Korrektheit der implementierten Scheduling-Algorithmen untersucht. Zunächst wird in Abschnitt 7.1 ein Überblick über den Durchsatz des Switches gegeben. Anschließend werden in Abschnitt 7.2 die Ergebnisse der Korrektheitsanalyse vorgestellt.

7.0.1 Testaufbau

Der beschriebene Aufbau des Testsystems von Abschnitt 6.1 wurde für die Experimente genutzt. Der Aufbau enthält einen Sender- und einen Empfängerport, sowie zwei Ports für den Switch. Der Sender und der Empfänger sind jeweils mit einem Port des Switches verbunden. Alle verwendeten Ports sind Gigabit-Ports. Die Sender schickt jeweils 60 Byte große UDP-Pakete an den Empfangsport vom Switch. Bei einer Paketgröße von 60 Bytes und einem Linerate von 1 Gbps können alle Ports ungefähr 1.666.666 PPS übertragen. Die Linerate gibt an, wie viele Bits oder Pakete pro Sekunde übertragen werden können. Es wurden bei der Untersuchung immer zwei Ansätze miteinander verglichen. Der ersten Ansatz ist der Batching-Ansatz, welcher das Batchen von Paketen an den Kernel gibt, ohne auf Versandbestätigungen zu warten. Der zweite Ansatz ist der Stop and Wait-Ansatz, bei welchem ein Paket versendet wird und auf die Versandbestätigungen gewartet wird.

7.1 Durchsatz

Damit der erstellte Sender bei beiden Ansätzen nicht der begrenzende Faktor ist, wurde das Server-Programm drei Mal gleichzeitig gestartet und die Programme auf unterschiedliche CPUs und Ports aufgeteilt.
Der erstellte Sender ist ein Raw-Socket, welcher ein UDP-Paket inklusive Ethernet-Header selbst erstellt. In einer While-Schleife werden nun 1 Million Pakete ohne Unterbrechung an den Switch gesendet. Das gewählte Scheduling-Verfahren für das Experiment ist FIFO. Bei allen drei Programmen werden in einem extra erstellen Thread alle 50 Millisekunden die Anzahl der versendeten und oder empfangenen Pakete sowie die Zeit gespeichert. Aus diesen zwei Größen werden dann die Pakete pro Sekunde ermittelt. Außerdem wurde die Anzahl der verlorenen, d.h. vom Sender gesendeten aber nicht vom Empfänger empfangenen Pakete ermittelt, die einen Indikator für einen überlasteten Switch darstellen, der aufgrund von überlaufendem Puffern Pakete verwirft.

Abbildung 7.1 und Abbildung 7.5 visualisieren die Paketrate, die vom Sender versendet und vom Empfänger empfangen wurden. Abbildung 7.2 und Abbildung 7.6 zeigen die Anzahl der Pakete die vom Sender versendet bzw. vom Empfänger empfangen wurden. Die Pakete, die pro Sekunde vom Switch versendet bzw. empfangen werden, werden in Abbildung 7.3 und Abbildung 7.7 dargestellt.

Abbildung 7.4 und Abbildung 7.8 veranschaulichen die Anzahl der Pakete, die von dem Switch im RX-Ring empfangen, vom Switch an den Kernel weitergegeben wurden (TX-Ring) und vom System tatsächlich versendet (CL-Ring) wurden.

7.1.1 Batching

Zu Beginn steigert sich beim Sender eins und drei die Rate deutlich von null bis ca. 450.000 PPS und bei Sender von null bis ca. 275.000 PPS. Nach dem Anstieg bleibt die Rate bei allen Server auf diesem Niveau. 2,1 Sekunden nach dem Start fällt die Rate von Server eins und drei auf null ab, da diese die Million Pakete versendet haben. Die Rate von Sender zwei steigert sich daraufhin auf 450.000 PPS, da nun mehr Kapazitäten für das Programm frei sind. Eine Sekunde später fällt auch hier die Rate auf null, nachdem alle Pakete versendet wurden. Die Kontinuität zeigt sich ebenfalls in Abbildung 7.1, in welcher die Kurve vom Sender bis zur 2.1 Sekunde eine lineare Steigung hat. Die Steigung flacht danach ab, da zwei Sender das Senden abgeschlossen haben. In den ersten beiden Sekunden senden alle drei Sender zusammen mit einer Linerate von ca. 1,1 Milionen PPS.
Die Linerate bezieht sich auf die Geschwindigkeit, mit der Daten durch ein Netzwerk oder ein Gerät fließen. Tabelle 7.1 zeigt die dazugehörigen Werte.

Die ersten Pakete, welche den Empfänger in Abbildung 7.1 erreichen, sind im Vergleich zum Sender ein paar Mikrosekunden später. Erst steigt der Empfänger bis auf 200.000 PPS und pendelt dann über die komplette Zeit zwischen 190.000 und 420.000 PPS. Die Empfangs-Kurve sieht sehr ähnlich zur Sender-Kurve des Switch aus, mit dem Unterschied, dass die Switch-Sender-Kurve nicht so hohe Ausschläge nach unten und oben vorweist. Tabelle 7.1 stellt sehr gut dar, dass beim Sender Switch und beim Empfänger der Median und Durchschnitt sowie die dazugehörigen Konfidenzintervalle sehr nah beieinander liegen.

Abbildung 7.4 zeigt, dass von den 3 Mio. Paketen nur 1.950.538 Pakete beim Switch tatsächlich angekommen sind. Die verlorenen gegangenen Pakete musste der Switch verwerfen, da der definiert Puffer nur 1.034.240 Pakete á 60 Byte zwischenspeichern kann. Bis zur 2,1 Sekunde empfängt der Switch die Pakete konstant, danach verringert sich die Steigung, da nur noch ein Sender sendet. Nachdem die Sender bei der 3,1 Sekunde aufgehört haben zu senden, flacht die Empfangskurve des Switch komplett ab und die zwischengespeicherten Pakete werden noch versendet. Die RX-Kurve ist knapp über der CL-Kurve, da Pakete erst versendet werden müssen und dann der Versand bestätigt werden kann. Am Ende befinden sich beide auf ca. 1.950.000 versendeten und bestätigten Paketen. Zwischen dem Switch und dem Empfänger sind weitere 75.000 Pakete verloren gegangen. In dem Zeitraum des Sendens wurden Pakete vom TX-Ring, weil dieser überlaufen ist, vom Kernel oder dem Interface verworfen. Die maximale Empfangsrate vom Switch, falls die Puffer leer sind, liegt bei ca. 800.000 PPS und die Senderate ist unabhängig davon bei ca. 315.000 PPS, wie der Tabelle 7.1 entnommen werden kann.

[1] 95%-Konfidenzintervall des Medians
[2] 95%-Konfidenzintervall des Durchschnitts

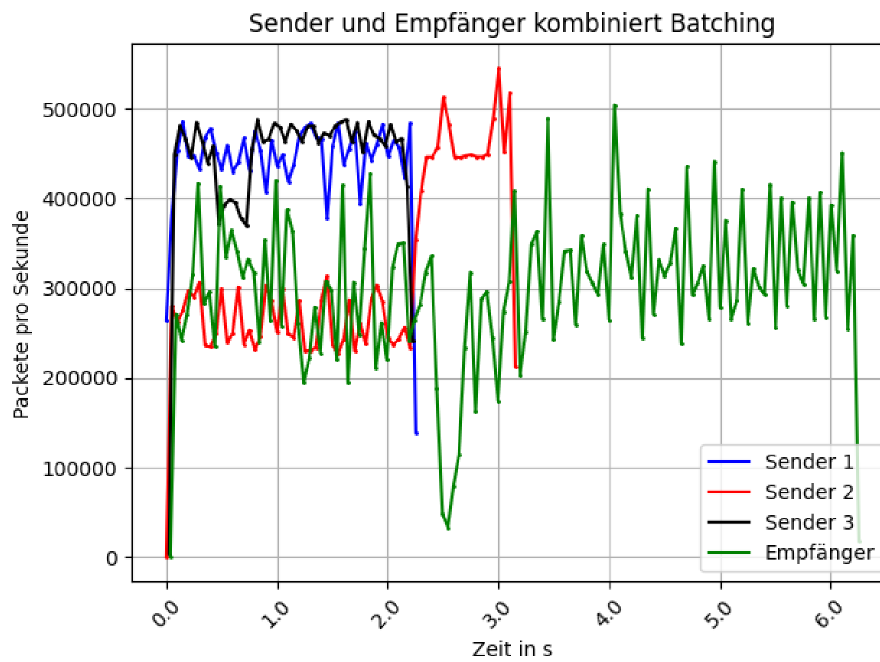

Abbildung 7.1: PPS Sender und Empfänger Batching

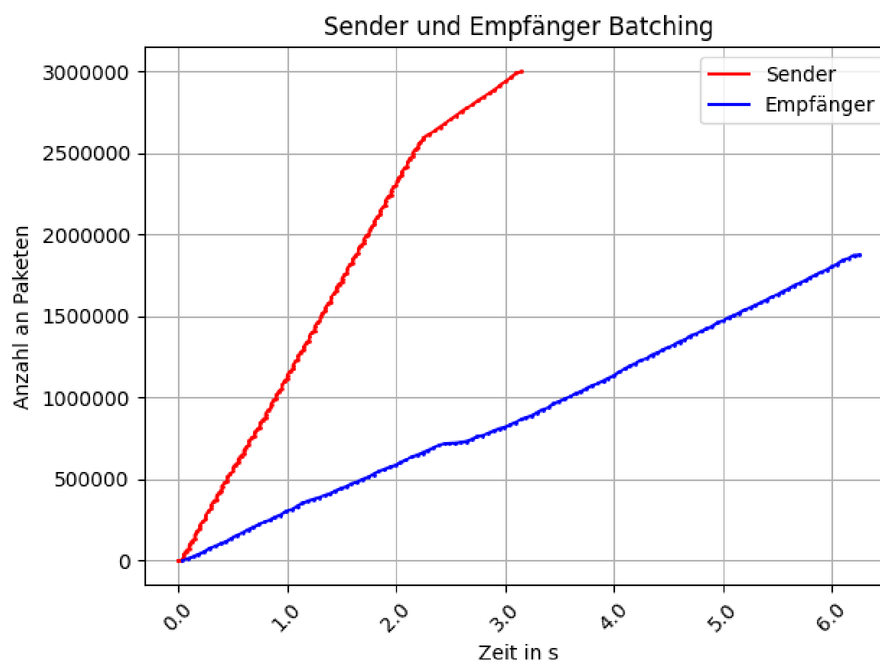

Abbildung 7.2: Pakete Sender und Empfänger Batching

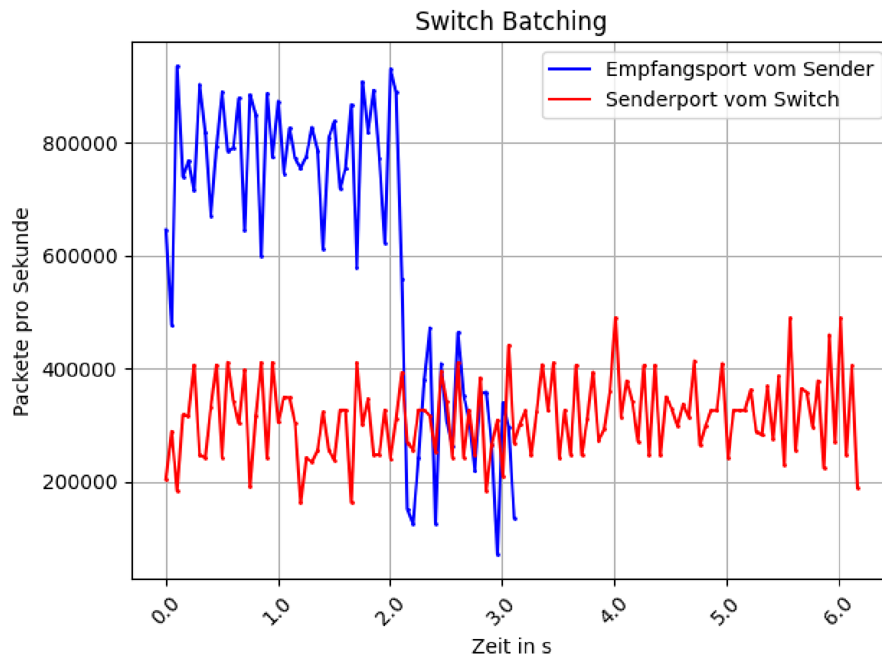

Abbildung 7.3: PPS Switch Batching

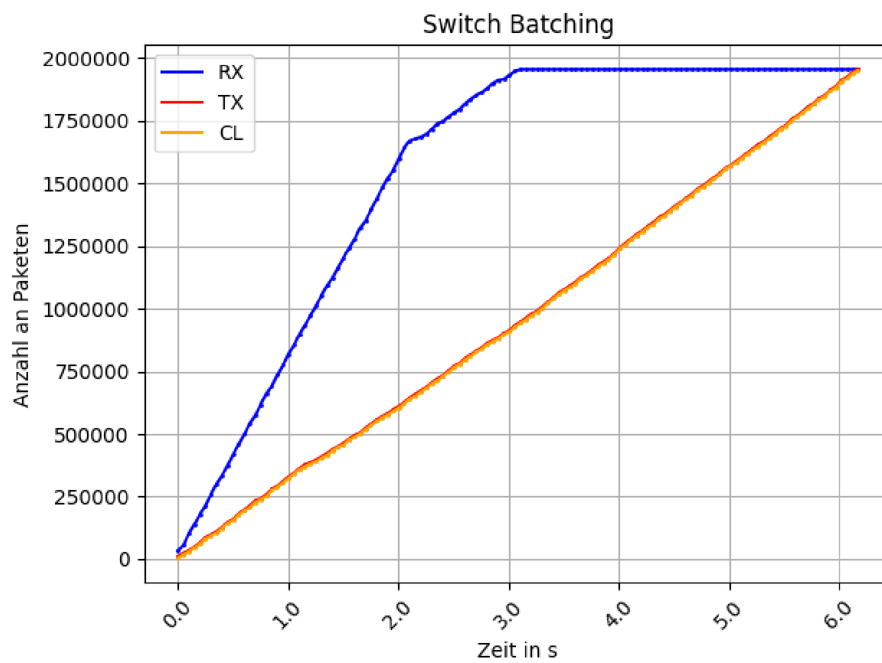

Abbildung 7.4: Pakete Switch Batching

pps	Median	Intervall[1]	Durchschnitt	Intervall[2]
Sender 1	453.785	(436.784, 470.785)	439.670	(422.669, 456.671)
Sender 2	285.293	(260.164, 310.422)	312.068	(286.939, 337.198)
Sender 3	463.991	(440.359, 487.622)	441.995	(418.364, 465.626)
Switch (Empfänger)	739.084	(675.726, 802.441)	619.100	(555.743, 682.458)
Switch (Sender)	315.760	(303.286, 328.233)	314.154	(301.680, 326.627)
Empfänger	304913	(289.561, 320.264)	299074	(283.722, 314.425)

Tabelle 7.1: Durchschnitt und Empfänger PPS Batching

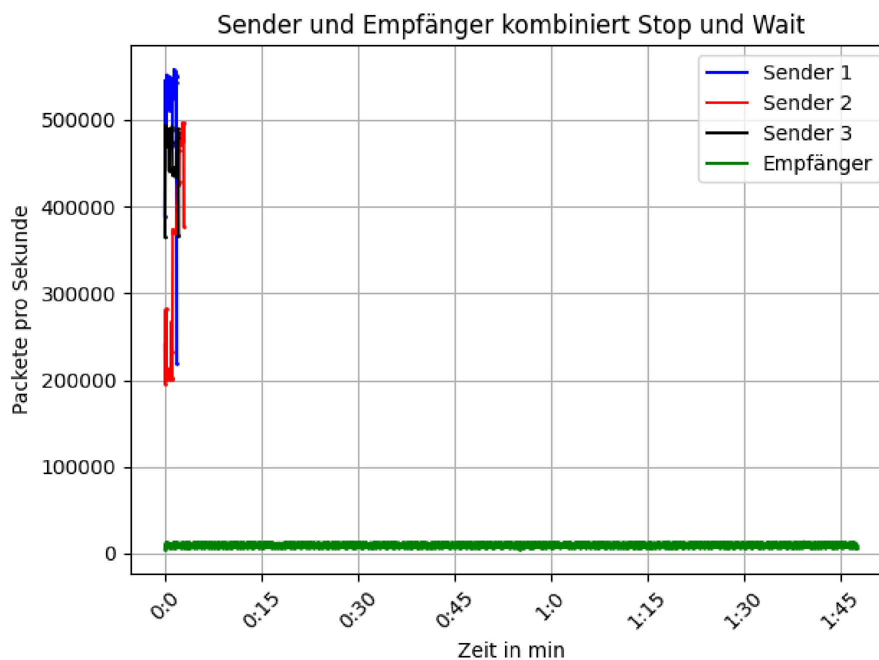

Abbildung 7.5: PPS Sender und Empfänger mit Stop and Wait

7.1.2 Stop & Wait

Die Linerate und die Aufteilung von den PPS von den Sendern ist sehr ähnlich zu der in Abschnitt 7.1.1, da hier nichts verändert wurde. Das zeigt auch der Vergleich von den sechs Median von Sender eins, zwei und drei aus den Tabelle 7.2 und Tabelle 7.1.

Die Empfängerrate fängt ebenfalls bei null und geht bis auf ca. 9.800 PPS hoch, was deutlich unter der Senderate liegt. Hier bleibt die Rate sehr konstant, was in Abbildung 7.5 zu sehen ist.

Die Kurven vom Sendeport des Switches und des Empfänger in Abbildung 7.7 und Abbildung 7.8, welche die PPS zeigen, sind fast identisch, dies zeigt, dass die Senderate vom Switch abhängt. Anzumerken ist, dass in Abbildung 7.8 die Werte von RX und CL genau übereinander liegen, da durch das Stop and Wait keine weiteren Pakete auf dem gleichen Port verschickt werden können.

[3] 95%-Konfidenzintervall des Medians

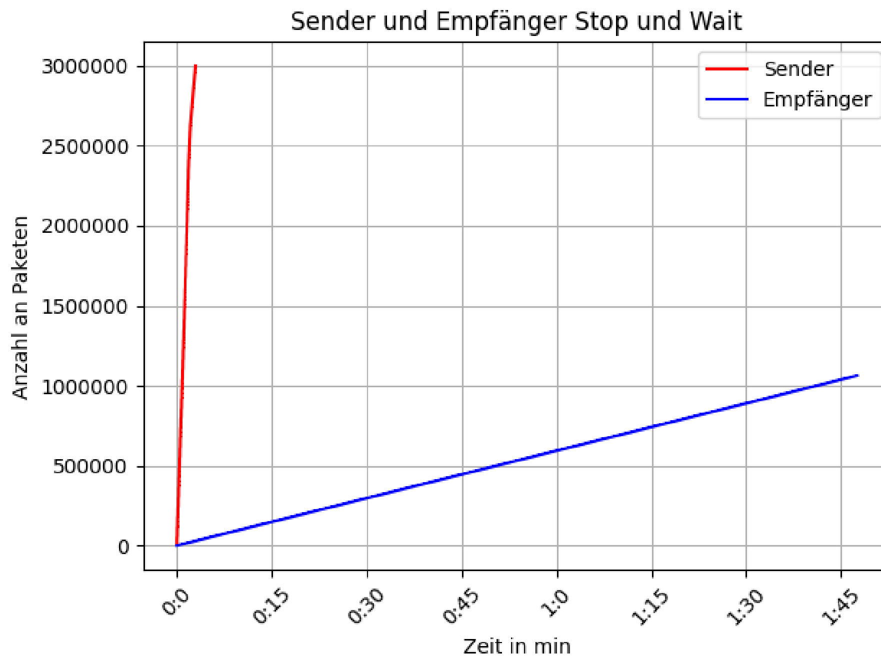

Abbildung 7.6: Pakete Sender und Empfänger mit Stop and Wait

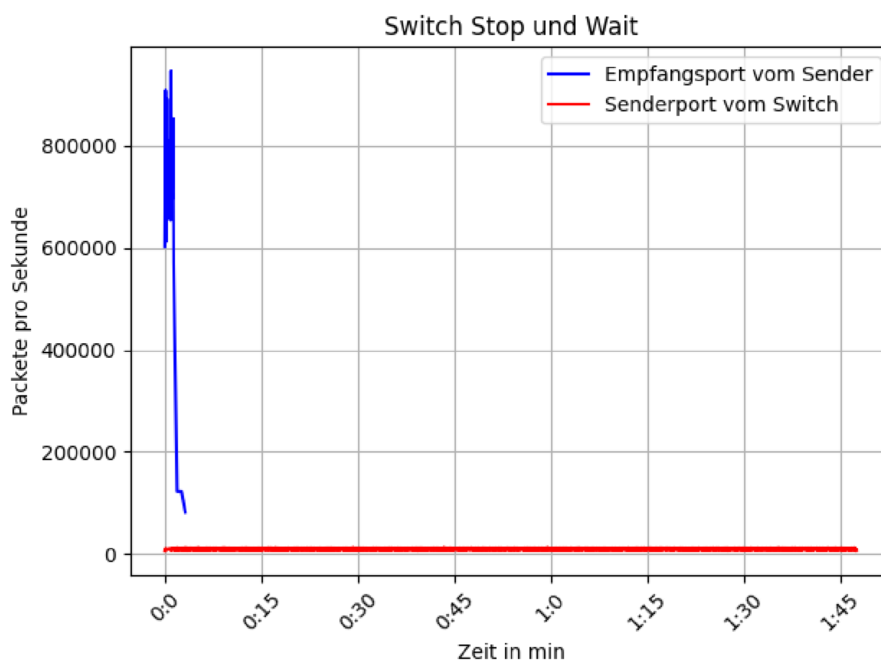

Abbildung 7.7: PPS Switch mit Stop and Wait

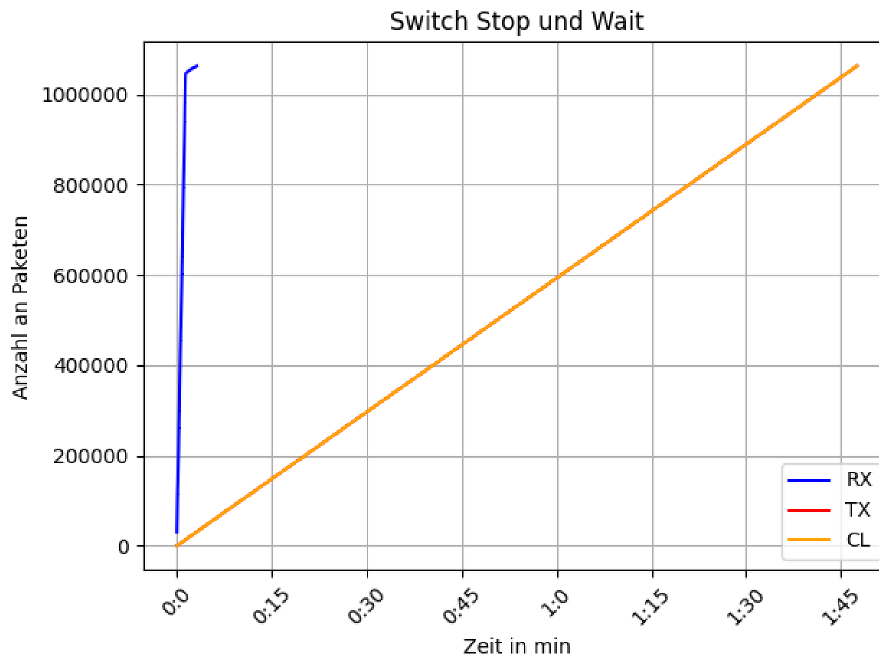

Abbildung 7.8: Pakete Switch mit Stop and Wait

pps	Median	Intervall[3]	Durchschnitt	Intervall[4]
Sender 1	544.253	(525.176, 563.330)	525.160	(506.083, 544.236)
Sender 2	370.996	(342.233, 399.758)	338.482	(309.720, 367.244)
Sender 3	476.369	(467.268, 485.469)	464.175	(455.074, 473.275)
Switch (Empfänger)	735.708	(662.026, 809.389)	683.506	(609824, 757.188)
Switch (Sender)	9.884	(9.808, 9.959)	9.879	(9.804, 9.954)
Empfänger	9.885	(9.809, 9.960)	9.877	(9.801, 9.953)

Tabelle 7.2: Durchschnitt und Median der PPS Stop and Wait

Tabelle 7.2 demonstriert den Median und den Durchschnitt sowie deren Konfidenzintervalle, dass der Sender und Empfangsport des Switch sowohl beim Median als auch beim Durchschnitt und das dazugehörige Konfidenzintervall sehr nah beieinander liegen, sodass der Switch kurzfristig mit Empfangsraten von ca. 1,1 Mio. PPS bei einem Socket-Sender zurecht kommt. Der Sendeport des Switch und der Empfänger liegen beim Median und beim Durchschnitt noch enger beieinander, sodass der Switch bei diesem Szenario mit einer Senderate von ca. 9.800 PPS senden kann.

Da der definiert Puffer nur 1.034.240 Pakete á 60 Byte zwischenspeichern kann und der Switch die Pakete nicht schnell genug wieder versenden kann, werden Pakete verworfen. Insgesamt beläuft sich der Paketverlust bei 3 Mio. versendeten Paketen auf 1.937.088 Pakete, welche nicht beim

[4] 95%-Konfidenzintervall des Durchschnitts

Empfänger angekommen sind. Dies kann in Abbildung 7.7 beobachtet werden, da der RX-Ring nicht weiter ansteigt als die 1.062.912 insgesamt versendeten Pakete. Die Anzahl wird auch vom Empfänger empfangen, was in Abbildung 7.6 zu sehen ist.

7.1.3 Vergleich

Durch das Stop and Wait, also das Warten auf das Versenden, verringert sich die Senderate im Vergleich zum Batching von 315.000 PPS auf maximal 9.800 PPS. Ein weiterer Grund für die verringerte Rate ist, dass beim Batching versucht wird so viele Pakete wie möglich an den Kernel weiterzugeben. Dies führt dazu, dass mehr Kontextwechsel zwischen Kernel und Userspace gemacht werden müssen. Die kleine Senderate sorgt jedoch dafür, dass alle vom Switch empfangenen Pakete erst nach 110s angekommen, während beim Batching innerhalb von 6,5 Sekunden fast 2 Millionen Pakete ankommen sind. Ein Vorteil von der Implementierung von Stop and Wait hat jedoch, dass wirklich alle Pakete, die beim Switch zwischengespeichert werden können beim Empfänger ankommen. Beim Batching sind von 3 Mio. angekommenen Pakete beim Switch 1.049.462 Pakete verloren gegangen. Beim Stop and Wait konnten dafür 1.937.088 Pakete gar nicht im Switch zwischengespeichert werden.

Insgesamt kann der Switch schneller versenden, was die zuverlässigen Zustellung der Pakete jedoch massiv beeinflusst. Es kann nur durch den CL-Ring die zuverlässige Versendung gewährleistet werden, was die Geschwindigkeit jedoch massiv drosselt, sobald auf diesen gewartet wird. Das Stop and Wait sorgt außerdem, dass die Senderate weniger Schwankungen hat als beim maximal Burst. Beim Empfangen kann der Switch eine Rate von 72 Mb/s erreichen, beim Versenden ohne Warten 18,9 Mb/s und beim Versenden mit Warten verringert sich das Rate auf 0.588 Mb/s.

Bei beiden Szenarien muss der Switch Pakete verwerfen sobald dieser komplett gefüllt ist.

7.2 Korrektheit

Die vorimplementierten Scheduling-Algorithmen Fair-Queuing, Weighted-Fair-Queuing, Strict-Priority-Queuing und Scheduled Traffic werden hierbei auf ihre Korrektheit untersucht. Von den im Switch gepufferten Paketen soll jeweils das als Nächstes über die Netzwerkschnittstelle gesendet werden, das der Scheduling-Algorithmus als nächstes Paket definiert. Eine zu frühe Scheduling-Entscheidung bzw. Pufferung nach der Scheduling-Entscheidung führt zu inkorrektem Scheduling. Daher versucht der Stop and Wait-Ansatz, die Scheduling-Entscheidung so spät wie möglich (kurz vor dem tatsächlichen Senden) zu treffen. Hier soll die Effektivität dieses Stop and Wait-Ansatzes gezeigt werden. Es handelt sich um einen Blackbox Test, der auf die Verarbeitung im inneren schließen soll.

Für die Korrektheitanalyse wurden zwei Sender erstellt, welche VLAN Pakete erstellen, damit der Switch mittels des PCP-Feldes die Priorität unterscheiden kann. Sender eins hat dabei Pakete mit Priorität null (PCP = eins) und Sender zwei mit Priorität null (PCP = sieben). Beide Sender haben jeweils 25.000 Pakete an den Switch gesendet und wurden zu ähnlichen Zeiten gestartet. Damit sich im Switch eine Warteschlange bildet wurde die Ausgangsport und der Empfängerport auf 10 Mb/s beschränkt.
Es wird TCPDUMP verwendet, um die Pakete auf einem zusätzlichen Port zu empfangen. Dadurch

kann die genaue Ankunftszeit der Pakete (Hardware Timestamping) sowie ihre Eigenschaften erfasst werden.
Die Uhren jedes Anschlusses im Experiment werden mit dem Precision Time Protocol (PTP) synchronisiert, um eine konsistente Testumgebung zu gewährleisten. PTP versucht, zwei Uhren so zu synchronisieren, sodass der Versatz zwischen beiden minimal ist und die Frequenzen annähernd gleich sind [iee20].
Mittels des TAP wird die Eingangsleitung und beim Empfängerport wird die Ausgangsleitung beobachtet.

7.2.1 Methodik zur Bestimmung der Scheduling-Fehler

Mittels eines Python-Scripts wurden die generierten Daten aus den Experimenten automatisch ausgewertet. Das erste Vorgehen war dabei bei allen gleich, die beiden Log-Datei von TCPDUMP auslesen. Es wurde aus den Log-Daten die Warteschlange beim Switch zum Sendezeitpunkt eines Pakets vom Switch rekonstruiert. Entspricht das tatsächlich gesendete Paket dem erwarteten Paket bzgl. des Scheduling-Algorithmus, so gilt das Paket als korrekt gescheduled, ansonsten als falsch gescheduled.

Fair-Queuing

Hier gibt es eine Variable, die den Round-Robin-Index darstellt, solange eine der beiden Listen komplett leer ist, ist diese inaktiv. In dieser Zeit kann es nur zu einem Fehler kommen, falls ein Paket vom Switch versendet wurde, obwohl mit dieser Priorität keines mehr vom Switch empfangen worden ist. Sind beide Listen nicht leer, wird das erste Paket auf den Round-Robin-Index gesetzt. Das nächste empfangene Paket muss nun der anderen Priorität entsprechen, sonst wurde eine Fehler gefunden.

Weighted-Fair-Queuing

Das Gewicht der Priorität sieben ist zwei und das Gewicht der Priorität null ist eins, was bedeutet, es müssen zwei mal so viele sieben Pakete wie null Pakete in der Zeit, in der beide Listen gefüllt sind, ankommen. Es gibt zwei Zustandsvariablen, welche die Priorität des letzten und vorletzten Paketes abspeichern. Falls eine der Listen leer ist, kann es nur zu einem Fehler kommen, falls ein Paket vom Switch versendet wurde, obwohl mit dieser Priorität keines mehr vom Switch empfangen worden ist. Sollte nun ein Muster auftreten, dass nicht (7,7,0), (0,7,7) oder (7,0,7) entspricht, handelt es sich um einen Fehlerfall und es wird der Fehlerindex erhöht.

Strict-Priority-Queuing

Die Warteschlange von Paketen muss solange geleert werden bis keine Pakete mehr übrig sind. Fehler können nur auftreten, falls sich noch Pakete in der Warteschlange von Priorität sieben befinden, aber ein Paket mit der Priorität null beim Empfänger angekommen ist. Falls die Warteschlange mit der Priorität sieben leer ist, aber noch ein Paket mit der Priorität sieben empfangen wird, handelt es sich ebenfalls um einen Fehlerfall.

Scheduled Traffic

Der Zyklus des Experiments waren zehn Sekunden, eine Sekunde für den echtzeitfähigen Netzwerkverkehr mit der Priorität sieben und neun Sekunden für den Best-Effort-Verkehr mit der Priorität null. Es gibt eine Startvariable, welche die erste Zeit von dem jeweiligen Zyklus abspeichert. Ist die Startzeit plus eine Sekunde beim echtzeitfähigen Netzwerkverkehr bzw. bei Best-Effort-Verkehr plus neun Sekunden kleiner als das aktuelle empfangene Paket, wurde der Zyklus überschritten und das Paket hätte nicht mehr versendet werden dürfen. Um die Startvariable nach einem Zyklus wieder auf die aktuelle Zeit zu setzten, gibt es eine Variable, welche die vorherige Zeit abspeichert. Falls die vorherige Zeit plus neun Sekunden bei echtzeitfähigen Verkehr und plus eine Sekunde bei Best-Effort-Verkehr kleiner der aktuellen Zeit ist, ist ein Zyklus vergangen. Die Startvariable wird dann auf die aktuelle Zeit neu gesetzt.

7.2.2 Resultate

Abbildung 7.10 stellt die Anzahl an Fehler dar, welche mittels der automatischen Auswertung aus Abschnitt 7.2.1 bestimmt wurden. Die vier Scheduling-Algorithmen FQ (Fair-Queuing), WFQ (Weighted-Fair-Queuing) SPQ (Strict-Priority-Queuing) und Scheduled Traffic (ST) werden einmal mit Stop and Wait und mit Batching dargestellt. Auffällig bei Abbildung 7.10 ist, dass es beim Stop and Wait entweder keine Fehler oder kaum Fehler gab.

Beim FQ gibt es, wie in Abbildung 7.10 zu sehen, beim Stop and Wait keine Fehler. Beim Batching hingegen wurden 1525 Fehler gefunden. Zu Fehlern kommt es sobald die eine Warteschlange bereits gefüllt ist und eine weitere Warteschlange mit einer anderen Priorität dazu kommt.

Es wurden beim WFQ beim Stop and Wait keine Fehler gefunden. Das Batching verursachte insgesamt 1953 Fehler. Hier wird das Gewicht von der priorisierten Warteschlange sieben teilweise nicht beachtet.

Das SPQ mit Stop and Wait verursachte auch hier keine Fehler. Pakete werden teilweise beim SPQ ab und zu mit Abwechselnder Priorität versendet oder die Warteschlange mit Priorität sieben ist kurzfristig leer und dann werden zu viele nuller Pakete auf einmal versendet, währenddessen die andere Warteschlange wieder gefüllt ist.

Die 25 Fehler bei ST in Abbildung 7.10 mit Stop and Wait lässt sich damit erklären, dass es beim echtzeitfähigen Netzwerkverkehr kein “Protection-Window“ gibt und Pakete noch kurz vor dem Ende des Zyklus versendet werden können. Ein weiterer Grund ist, dass die automatische Auswertung sehr streng ist und schon kleinste Mikrosekunden Unterschiede zwischen der Startzeit und der aktuellen Zeit zu einem Fehlerfall führen können. Die 44196 Fehler beim Batching lassen sich damit erklären, dass die Zyklen durch das unkontrollierte Senden ständig versetzt werden.

Wie schon in Abschnitt 7.1 zu sehen war, sieht man in Abbildung 7.10, dass es beim Stop and Wait zu keinen Paketverlusten kam. Da es bei jedem Scheduling-Algorithmus beim Batching zu massiven Fehlern kommt, sind hier nicht die Scheduling-Algorithmen das Problem, sondern die Technik mit der die Pakete versendet werden. Beide Diagramme zeigen gute Ergebnisse bei allen Scheduling-Algorithmen mit Stop and Wait, deshalb kann davon ausgegangen werden, dass der

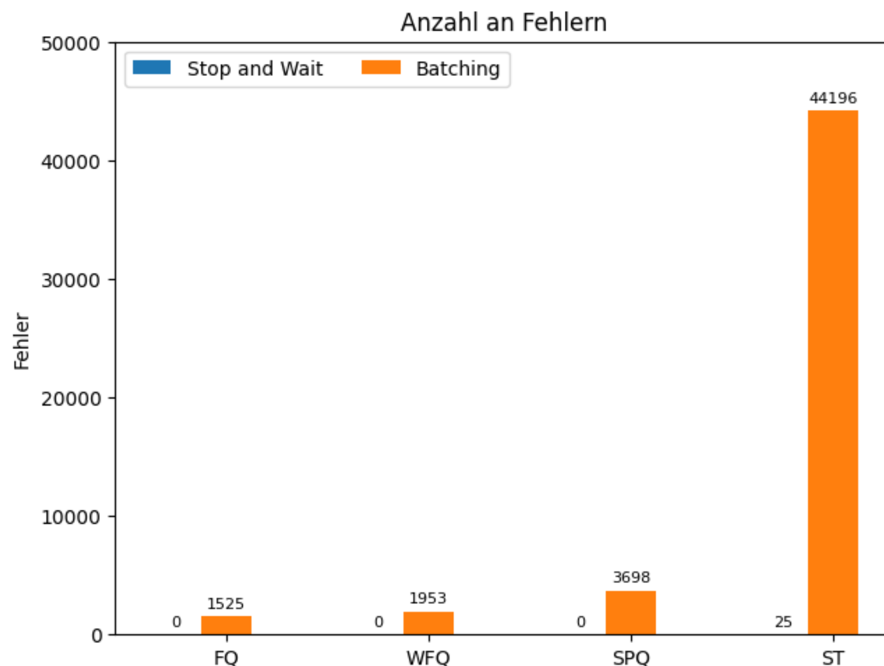

Abbildung 7.9: Fehleranzahl beim Scheduling

Switch die Pakete in die korrekt richtige Reihenfolge bringen kann. Das Experiment zeigt auch, sobald der Kernel die Pakete hat, die Reihenfolge der Pakete nicht mehr gewährleistet werden kann.

7.2.3 Zusammenfassung der Evaluierung

Stop and Wait führt zu besseren Scheduling-Entscheidungen im Vergleich zum Batching-Ansatz durch spätes Treffen der Scheduling-Entscheidung kurz vor dem Senden.

Der Stop Wait-Ansatz verringert den Durchsatz des Switches signifikant gegenüber dem Batching-Ansatz. Der erzielte Durchsatz von ca. 4,704 Mbps ist weit vom Durchsatz modernen Gigabit- oder gar Multi-Gigabit-Ethernet-Technolgien entfernt.

Die verwendete Technologie XDP bietet zwar die Möglichkeit, die Leistung eines Software-Switches zu verbessern, indem sie den Netzwerk-Stack auf eine höhere Schicht des Betriebssystems verlagert und so die Verarbeitungsleistung erhöht. Dennoch ist die Leistung von XDP immer noch begrenzt und kann nicht mit modernen Ethernet-Technologien mithalten, die speziell für die schnelle Übertragung großer Datenmengen konzipiert wurden. Der verwendete Ansatz - sei es Stop-and-Wait oder Batching - spielt keine Rolle.

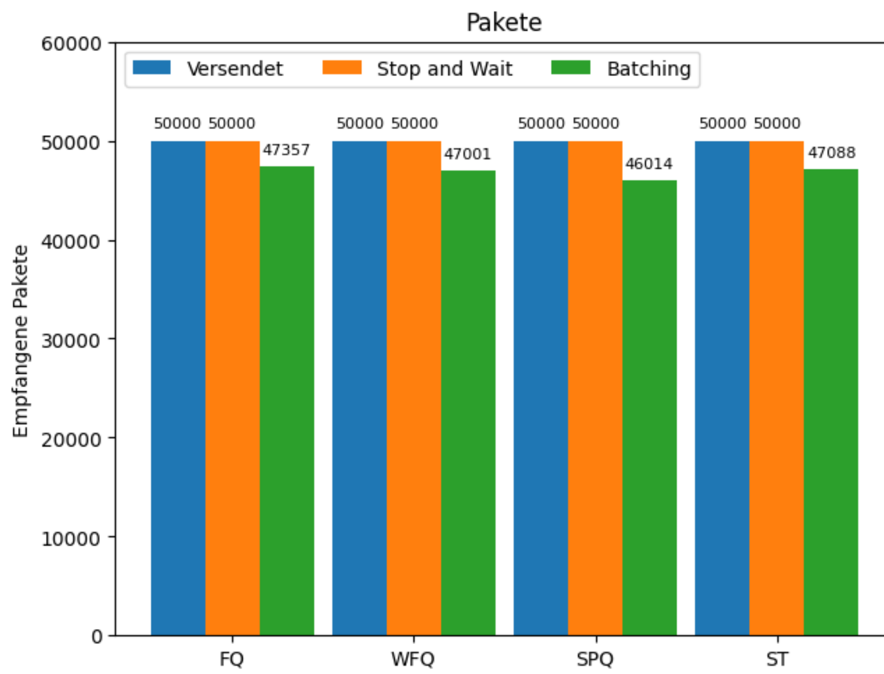

Abbildung 7.10: Empfangene Pakete beim Scheduling

8 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, ein generisches Framework zur Implementierung von einem Software-Switch zu entwickeln, das dazu genutzt werden kann, verschiedene Paket-Scheduling-Verfahren auf Anwendungsebene zu evaluieren. Die Anforderungen an das Framework waren, dass es leicht erweiterbar sein sollte, um verschiedene zeit- und prioritätengesteuerte Scheduling-Verfahren zu unterstützen. Weiterhin sollte das Framework das Scheduling-Verhalten akkurat abbilden und dabei möglichst spät vor dem Senden des Pakets über die physische Netzwerkschnittstelle die Scheduling-Entscheidung treffen. Dabei ist ein hoher Durchsatz von großer Bedeutung.

XDP ist eine von Linux im Kernel verankerte Technologie um Netzwerkpakete direkt nach der Netzwerkkarte abzufangen. Es ist eine Technologie zur Verbesserung der Netzwerk-Performance, welche flexible und leistungsstark ist. Die eBPF ermöglicht die Verwendung von XDP in der User-Space-Programmierung. Es ist einfach und erfordert keinen großen Aufwand, Pakete vor dem Kernel abzufangen. Diese Netzwerkpakete können an eine Map weitergeleitet werden. Maps können die Pakete vom Kernel mittels AF_XDP an Userspace weiterleiten.

Diese Arbeit hat erstmals ein generisches Framework mit AF_XDP erstellt, welches die Sockets nutzt, um damit einen Software-Switch zu implementieren. Die wesentliche Idee des Ansatzes besteht darin, das XDP-Framework zu nutzen, um einen direkten Zugriff auf Netzwerkpakete zu ermöglichen und dabei den Kernel zu umgehen. Dadurch soll die Netzwerkleistung verbessert werden. Das Framework ist generisch aufgebaut und es können eigene Forwarding, Klassifizierung und Scheduling-Klassen implementiert und genutzt werden. Im Rahmen der Demonstration wurde ein Beispiel-Setup für einen Switch präsentiert. Dabei wurden die verschiedenen Schritte erläutert, die beim Erstellen eines Switches zu beachten sind. Der Stop and Wait-Ansatz wird verwendet um sicherzustellen, dass jedes Paket erfolgreich übertragen wird, bevor das nächste gesendet wird. Hierbei wird auf den XDP-Completion-Ring zurückgegriffen, der anzeigt, wann ein Paket erfolgreich versendet wurde. Auf dieser Grundlage kann eine optimale Scheduling-Entscheidung getroffen werden.

Diese Arbeit lässt eine weite Auswahl potenzieller zukünftiger Arbeiten offen, welche zur Verbesserung des Softwareswitches beitragen können. Beispielsweise konzentriert sich diese Arbeit auf die grundlegende Implementierung und vergleicht XDP mit CL und ohne. Es wäre jedoch interessant zu untersuchen, wie effektiv XDP im Vergleich zu anderen Softwareswitches ist.

Auch wenn der Softwareswitch die Grundfunktionalität eines Layer zwei Software-Switches enthält, gibt es noch weitere potenzielle Funktionen, die für künftige Arbeiten interessant sein könnten. Dazu gehören:

- Zur Steigerung des Durchsatzes kann eine erneute Betrachtung der Synchronisierung von Threads mittels alternativer Locking-Methoden in Erwägung gezogen werden
- Weitere Scheduling-Algorithmen besonders weiter echtzeitfähige Algorithmen können implementiert werden

- Der bisher Layer zwei Software-Switches könnte, um die Funktionalität eines Layer drei Switches bzw. eines Routers erweitert werden
- Die Performance des Switches könnte noch zusätzlich mittels anderen Technologie, wie andere Kernel-Bypassing-Technologien wie DPDK, getestet werden

Bei dieser Arbeit wurden das Framework auf seine Korrektheit und seinen Durchsatz geprüft. Die Korrektheit wurde anhand von vier vorimplementierten Scheduling Algorithmen überprüft. Es wurde dabei gezeigt, dass das Scheduling nur korrekt sein kann mit dem Stop and Wait-Ansatz, bei dem auf das Versenden gewartet wird. Der erzielte Durchsatz von ca. 4,704 Mbps ist jedoch weit entfernt von den Durchsatzraten moderner Gigabit- oder Multi-Gigabit-Ethernet-Technologien. Die übertragenen Pakete erreichen den Empfänger in der richtigen Reihenfolge und es gehen alle Pakete tatsächlich beim Empfänger ein. Dank des Batching-Ansatzes, bei dem die Pakete an den Kernel weitergegeben werden, ohne auf Versandbestätigungen zu warten, erreichen sie den Empfänger schneller als beim Stop-and-Wait-Ansatz. Ein Nachteil des Batching-Verfahrens ist jedoch eine geringere Zuverlässigkeit beim Versenden, was dazu führen kann, dass nicht alle Pakete beim Empfänger ankommen, und auch eine inkorrekte Reihenfolge der Pakete. Sobald der Kernel ein Paket erhält, kann die Reihenfolge der Pakete nicht mehr beeinflusst werden.

Diese Arbeit hat gezeigt, dass beide Technologien XDP und eBPF miteinander vereinbar sind und es möglich, ist mittels dieser Technologie ein generisches Software-Framework zu definieren und zu implementieren. Die Korrektheit und der Durchsatz wurden ebenfalls evaluiert.
Kritisch bei der Implementierung ist, dass die einzelnen Threads zunächst mit Mutex und einmal mit einer threadsichern Klasse abgesichert werden, hier wäre eine einheitliche Lösung effizienter und technisch besser gewesen.
Zum aktuellen Zeitpunkt werden VLAN-Pakete, welche keinem Ausgangsport zugeordnet werden können, dem Empfänger zurückgeschickt. Dies ist bisher die einzige Methode, dass weder die Ringe noch der Puffer überläuft. Hier wäre es besser die Pakete aus dem Ring und dem Puffer zu entfernen.
In der Implementierung hat sich die Entscheidung des Multithreading als sehr vorteilhaft erwiesen, um die Performance deutlich zu erhöhen.
Falls der Switch keinen Puffer mehr frei hat, kommt es nicht zu einem Ausfall des Programmes und der Switch verwirft die neu eingetroffenen Pakete.
Die aktuelle Version des generischen Frameworks kann genutzt werden um neue Echtzeit-Scheduling-Algorithmen zu implementieren und auf ihre Korrektheit zu testen.

Literaturverzeichnis

[A L16] A Linux Foundation Collaborative Project. *Production Quality, Multilayer Open Virtual Switch*. 2016. URL: https://www.openvswitch.org/ (besucht am 02. 04. 2023) (zitiert auf S. 19).

[BN22] Bob Lantz, Nikhil Handigol, Brandon Heller, Vimal Jeyakumar. *Mininet*. 7.05.2022. URL: https://github.com/mininet/mininet/wiki (besucht am 16. 11. 2022) (zitiert auf S. 19, 20).

[Cil21] Cilium Authors. *BPF and XDP Reference Guide*. 7.12.2021. URL: https://docs.cilium.io/en/latest/bpf/ (besucht am 21. 10. 2022) (zitiert auf S. 12, 13).

[die] die.net. *phc2sys(8) - Linux man page*. URL: https://linux.die.net/man/8/phc2sys (besucht am 27. 03. 2023) (zitiert auf S. 43).

[DN16] F. Dürr, N. G. Nayak. „No-Wait Packet Scheduling for IEEE Time-Sensitive Networks (TSN)“. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS ’16. New York, NY, USA: Association for Computing Machinery, 2016, S. 203–212. ISBN: 9781450347877. DOI: 10.1145/2997465.2997494 (zitiert auf S. 7, 8).

[DPD] DPDK Project. *DPDK*. URL: https://www.dpdk.org/ (besucht am 12. 11. 2022) (zitiert auf S. 18).

[ele] elektronik-kompendium.de. *Ethernet-Frame (Rahmenformat)*. URL: https://www.elektronik-kompendium.de/sites/net/1406191.htm (besucht am 16. 03. 2023) (zitiert auf S. 10, 11).

[FOC+22] E. Freitas, A. T. de Oliveira Filho, Carmo, Pedro R. X. do, D. F. H. Sadok, J. Kelner. „Takeaways from an experimental evaluation of eXpress Data Path (XDP) and Data Plane Development Kit (DPDK) under a Cloud Computing environment“. In: *Research, Society and Development* 11.12 (2022), S. 1–21. DOI: 10.33448/rsd-v11i12.34200. URL: https://rsdjournal.org/index.php/rsd/article/view/34200 (zitiert auf S. 18).

[Gre19] B. Gregg. *BPF Performance Tools: Linux System and Application Observability*. 1st. Addison-Wesley Professional, 2019, S. 1–180. ISBN: 0136554822 (zitiert auf S. 12).

[Her20] D. Hercog. „Protocols RTP/RTCP“. In: *Communication Protocols: Principles, Methods and Specifications*. Cham: Springer International Publishing, 2020, S. 343–344. ISBN: 978-3-030-50405-2. DOI: 10.1007/978-3-030-50405-2{\textunderscore}24 (zitiert auf S. 10).

[iee11] ieee.org. „IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks“. In: *IEEE Std 802.1Q-2011 (Revision of IEEE Std 802.1Q-2005)* (2011), S. 1–1365. DOI: 10.1109/IEEESTD.2011.6009146 (zitiert auf S. 2, 6, 7, 9, 30).

[iee20] ieee.org. „IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems“. In: *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)* (2020), S. 1–499. DOI: 10.1109/IEEESTD.2020.9120376. URL: https://ieeexplore.ieee.org/document/9120376 (besucht am 02. 04. 2023) (zitiert auf S. 55).

[Int22] Intel Corporation. *bpf-examples*. 16.12.2022. URL: https://github.com/xdp-project/bpf-examples/tree/master/AF_XDP-forwarding (besucht am 02. 04. 2023) (zitiert auf S. 25, 38).

[IO 23] IO Visor Project. *bcc*. 2.04.2023. URL: https://github.com/iovisor/bcc (besucht am 02. 04. 2023) (zitiert auf S. 12).

[KT18] M. Karlsson, B. Töpel. „The path to DPDK speeds for AF XDP“. In: *Linux Plumbers Conference*. 2018, S. 1–9 (zitiert auf S. 14, 15).

[LG] Luigi Rizzo, Giuseppe Lettieri, Università di Pisa. *VALE, a Virtual Local Ethernet*. URL: http://info.iet.unipi.it/~luigi/vale/ (besucht am 02. 04. 2023) (zitiert auf S. 19).

[lui23] luigirizzo. *netmap*. 22.03.2023. URL: https://github.com/luigirizzo/netmap (besucht am 02. 04. 2023) (zitiert auf S. 17).

[MS99] Mahadevan I., Sivalingam K.M. „Quality of Service architectures for wireless networks: IntServ and DiffServ models“. In: *Proceedings Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN’99)*. 1999, S. 420–425. DOI: 10.1109/ISPAN.1999.778974 (zitiert auf S. 9).

[OTCK20] Osiński Tomasz, Tarasiuk Halina, Chaignon Paul, Kossakowski Mateusz. „P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4“. In: *2020 IFIP Networking Conference (Networking)*. 2020, S. 413–421 (zitiert auf S. 19).

[PH16] D. A. Patterson, J. L. Hennessy. *Computer Organization and Design ARM, Interactive Edition: The Hardware Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2016, S. 254–260. ISBN: 9780128018354. URL: https://books.google.de/books?id=Pz-XCgAAQBAJ (zitiert auf S. 6).

[PRO23] PROFIBUS Nutzerorganisation e.V. *PROFINET - the leading Industrial Ethernet Standard*. 2023. URL: https://www.profibus.com/technology/profinet (besucht am 26. 03. 2023) (zitiert auf S. 7).

[RCC12] L. Rizzo, M. Carbone, G. Catalli. „Transparent acceleration of software packet forwarding using netmap“. In: *2012 Proceedings IEEE INFOCOM*. 2012, S. 2471–2479 (zitiert auf S. 17, 18).

[Red20] Red Hat. *Was ist ein Hypervisor?* 10.01.2020. URL: https://www.redhat.com/de/topics/virtualization/what-is-a-hypervisor (besucht am 15. 11. 2022) (zitiert auf S. 19).

[Riz12] L. Rizzo. „netmap: a novel framework for fast packet I/O“. In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, S. 101–112 (zitiert auf S. 19).

[RL12] L. Rizzo, G. Lettieri. „Vale, a switched ethernet for virtual machines“. In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 2012, S. 61–72 (zitiert auf S. 17).

[SCFv03] H. Schulzrinne, S. Casner, R. Frederick, van Jacobson. *RTP: A transport protocol for real-time applications*. 2003. URL: https://www.rfc-editor.org/rfc/rfc3550 (besucht am 02. 04. 2023) (zitiert auf S. 10).

[Sch14] D. Scholz. „A look at Intel's dataplane development kit". In: *Network* 115 (2014), S. 1–8. URL: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2014-08-1/NET-2014-08-1_15.pdf (zitiert auf S. 18).

[SE08] R. Seifert, J. Edwards. *The All-New Switch Book: The Complete Guide to LAN Switching Technology*. John Wiley & Sons, 2008, S. 609–698 (zitiert auf S. 3–5).

[Sta15] W. Stallings. *Foundations of modern networking: SDN, NFV, QoE, IoT, and Cloud*. Addison-Wesley Professional, 2015, S. 1–91. (Besucht am 02. 04. 2023) (zitiert auf S. 1).

[TB] The kernel development community, Björn Töpel, Magnus Karlsson, Alexander Duyck, Alexei Starovoitov, Daniel Borkmann, Jesper Dangaard Brouer, John Fastabend, Jonathan Corbet, Michael S. Tsirkin, Qi Z Zhang, Willem de Bruijn. *AF_XDP*. URL: https://www.kernel.org/doc/html/latest/networking/af_xdp.html (besucht am 04. 01. 2022) (zitiert auf S. 15, 27, 28, 44).

[The20] The Linux Programming Interface. *network_namespaces(7) — Linux manual page*. 9.06.2020. URL: https://man7.org/linux/man-pages/man7/network_namespaces.7.html (besucht am 27. 03. 2023) (zitiert auf S. 44).

[Tok] E. C. Toke Høiland-Jørgensen. *xdp-tools - Library and utilities for use with XDP*. URL: https://github.com/xdp-project/xdp-tools (besucht am 27. 03. 2023) (zitiert auf S. 23).

[TW13] A. Tanenbaum, D. Wetherall. *Computer Networks : Pearson New International Edition*. Harlow, UNITED KINGDOM: Pearson Education, Limited, 2013, S. 193–496. ISBN: 9781292037189. URL: http://ebookcentral.proquest.com/lib/uni-stuttgart/detail.action?docID=5832033 (zitiert auf S. 3, 5, 6).

[VCP+20] M. A. M. Vieira, M. S. Castanho, R. D. G. Pac\'ıfico, E. R. S. Santos, Júnior, Eduardo P. M. Câmara, L. F. M. Vieira. „Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications". In: *ACM Comput. Surv.* 53.1 (2020), S. 1–36. ISSN: 0360-0300. DOI: 10.1145/3371038 (zitiert auf S. 1, 13, 14).

[Zöb20] D. Zöbel. „Grundlegende Planungsverfahren". In: *Echtzeitsysteme: Grundlagen der Planung*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, S. 181–253. ISBN: 978-3-662-60421-2. DOI: 10.1007/978-3-662-60421-2{\textunderscore}3 (zitiert auf S. 1).

Alle URLs wurden zuletzt am 02. 04. 2023 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift