



Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Aufbau einer Lernplattform zur Programmiersprache P4

Bachelorarbeit

von

Marcel Beausencourt

Matrikelnummer: 573019

Fachbereich 1 – Energie und Information –
der Hochschule für Technik und Wirtschaft Berlin

zur Erlangung des akademischen Grades
Bachelor of Engineering (B. Eng.)
im Studiengang
Informations- und Kommunikationstechnik

Tag der Abgabe: 09.02.2023

Erstgutachten: Prof. Dr. Thomas Scheffler
Zweitgutachten: Prof. Dr. Markus Nölle

Inhaltsverzeichnis

1 Einleitung	1
1.1 Hintergrund und Motivation	1
1.2 Aufgabenbeschreibung	1
1.3 Herangehensweise	2
2 Grundlagen von P4	3
2.1 Geschichtlicher Hintergrund	3
2.2 Zielsetzungen von P4	4
2.3 Allgemeines Funktionsprinzip	5
2.3.1 Genereller Aufbau der beiden Architekturen	6
2.3.2 wichtige Datentypen und Keywords	8
2.3.3 Parser	8
2.3.4 Match-Action und Control Flow	12
2.3.5 Deparser	16
2.3.6 weitere Blöcke	16
2.3.7 Paket-Replikation	17
2.4 Möglichkeiten von P4	18
3 Virtuelle Umgebung	23
3.1 Installation der VM	23
3.2 Konfiguration der VM	25
3.3 Übungen	27
3.3.1 Hello World	27
3.3.2 Simple Clone	29
3.3.3 Double Clone	30
3.3.4 Resubmit	31
3.3.5 Resubmit2-IP	33
3.3.6 Resubmit3-ARP	37
3.3.7 weitere Übungen	40
3.4 Konfiguration Mininet	40
3.4.1 Konfiguration der Topologie	40
3.4.2 Konfiguration Switches	41
3.4.3 Anpassen restlicher Dateien	44
3.5 Benutzung der Übungen	45
3.5.1 Start der Übung	45

3.5.2	Log-Files	46
3.5.3	Bedienung der Hosts	47
3.5.4	Beenden der Übung	47
4	Portierung auf reale Hardware	49
4.1	Laborhardware	49
4.2	Konfiguration der Hardware	51
4.2.1	Wedge 100BF-32X	51
4.2.2	IP-Konfiguration Server und PC	55
4.2.3	Konfiguration AS4610	56
4.3	Portierung des Codes von BMv2 auf TNA	56
4.3.1	Allgemeines	57
4.3.2	Änderungen Parser	57
4.3.3	Änderungen Deparser	57
4.3.4	Änderungen bei der Paketvervielfältigung	57
4.3.5	Anpassung der Ports	58
5	Tests	61
5.1	Tests in der virtuellen Umgebung	61
5.1.1	Testen mit Ping	61
5.1.2	Python Skripte	62
5.2	Tests auf der Laborhardware	63
5.2.1	Pingen auf der Hardware	63
5.2.2	iPerf-Tests	63
5.2.3	Scapy-Tests	65
5.2.4	Verifikation mit Wireshark	66
5.3	Fazit der Tests	67
6	Fazit und Ausblick	69
A	Aliasses auf Ubuntu-PC und/oder Server	73
B	/etc/hosts auf Ubuntu-Server	75
C	Konfiguration Switchports auf Wedge 100BF-32X	77
D	vollständige IP-Netns-Konfiguration des Servers	79
E	Konfiguration AS4610	81
F	Checksum berechnen	83
G	Python-Skript für Tabelleneinträge aus Simple-Clone	85
H	Scapy File	87

I IPv4-Adressen im Labornetz	89
J Programmdateien als .zip	91
Abkürzungsverzeichnis	93
Abbildungsverzeichnis	95
Tabellenverzeichnis	97
Quelltextverzeichnis	99
Literaturverzeichnis	101
Eigenständigkeitserklärung	103

Kurzfassung

Diese Bachelorarbeit soll als Grundlage für einen erleichterten Einstieg in die Programmiersprache P4 dienen. Aufbereitete Erkenntnissgewinne und Besonderheiten der Sprache, welche im bisherigen Verlauf des dazugehörigen Forschungsprojektes gesammelt wurden, werden dargestellt, erläutert und mittels selbst erstellten, praktischen Übungen vertieft.

Diese Übungen können sowohl auf realer Hardware als auch innerhalb einer *Virtuellen Maschine* (VM) bearbeitet werden. Die Hardware und weiteres Equipment stehen im Labor *Übertragungstechnik* (HTW Berlin Campus Wilhelminenhof G520) zur Verfügung. Die VM kann im Labor, aber auch auf einem eigenen Rechner genutzt werden. Der Zugang zur Laborhardware ist auch über das *Virtual Private Network* (VPN) der Hochschule möglich.

Die Einrichtung und Konfiguration der VM sollen in dieser Arbeit ebenfalls dargestellt werden. Außerdem soll ein Leitfaden für das Arbeiten mit der Hardware im Labor erstellt werden.

Die sechs selbst erstellten Übungen funktionieren vollständig für das *Behavioral Model Version 2* (BMv2)¹. Drei ausgewählte Übungen wurden im Laufe der Arbeit für den Einsatz auf einem realen P4-Switch portiert und getestet; die restlichen Übungen sollen folgen.

¹P4-Architektur des P4-Language-Konsortiums

Kapitel 1: Einleitung

1.1 Hintergrund und Motivation

Im Rahmen des Forschungsprojektes¹ beschäftige ich mich seit März 2022 intensiv mit der Programmiersprache „P4“ (Programming Protocol-independent Packet Processors). Sie wird seit 2013 durch das P4-Language-Consortium² entwickelt und ging aus Überlegungen zum Netzwerkprotokoll *OpenFlow* hervor. Mit Hilfe dieser imperativen Programmiersprache kann die *Data-Plane* von Netzwerkgeräten wie Switchen und Routern bzw. die in diesen verbaute *Network Processing Unit* (NPU) konfiguriert werden. Die *Data-Plane* ist die Hardwareschicht, welche für das Senden und Empfangen von Netzverkehr zuständig ist. Das Pendant zur *Data-Plane* ist die *Control-Plane*. In dieser laufen Routing-Protokolle und sie ist verantwortlich für das Anlegen von Tabelleneinträgen, die von den Routing-Protokollen genutzt werden. Die angelegten Tabelleneinträge werden von der *Data-Plane* genutzt, um Pakete weiterzuleiten.

P4 ist auch auf FPGAs und programmierbaren ASICs anwendbar. Das Ziel des Forschungsprojektes ist es einen Netzwerk-Lastgenerator zu realisieren und – damit verbunden – auch die Möglichkeiten und Chancen, welche P4 bietet, kennen zu lernen.

Obwohl ich bereits ein sehr gutes Vorwissen bezüglich Netzwerktechnik besaß, fiel mir in den ersten Monaten des Forschungsprojektes auf, wie schwierig es ist sich in die Thematik einzuarbeiten. Aufgrund der Tatsache, dass P4 eine relativ neue Programmiersprache ist, stellte ich fest, dass auch die Recherche im Internet sich schwieriger und aufwendiger gestaltete als erwartet. Aufgrund dessen kam mir die Idee meine bisherigen Erfahrungen und Erkenntnisse für interessierte Personen so aufzubereiten, dass diese sich schneller und ergebnisorientierter in die Sprache einarbeiten können.

1.2 Aufgabenbeschreibung

In dieser Arbeit sollen alle Schritte erläutert werden, die nötig sind, um die Übungen sowohl auf virtueller als auch auf realer Hardware absolvieren zu können. Dafür ist es wichtig auch eine grundlegende, inhaltliche Einführung in P4 und die bereits erstellten Übungen zu geben. Hierbei sollen Netzwerktopologien, dazugehörige Protokolle und der geschriebene Code

¹<https://www.ifaf-berlin.de/projekte/nettraffic-p4/>

²<https://p4.org/>

genauer erläutert werden. Auch automatisierte Abläufe sollen erwähnt und erklärt werden. Anschließend wird die Portierung des Codes von der virtuellen Umgebung auf die Hardware thematisiert. Dabei sollen vor allem strukturelle Unterschiede zwischen *Behavioral Model Version 2* (BMv2) und *Tofino Native Architecture* (TNA) dargestellt werden. Abschließend wird die Funktionalität des Netzes (inkl. Hosts) mittels *Internet Control Message Protocol* (ICMP) [s. [RFC792]], Scapy³ und anderen Tools verifiziert. Teilweise handelt es sich um englische Begriffe, weil diese auch in der deutschen Literatur gebräuchlicher sind.

1.3 Herangehensweise

In Kapitel 2 werden zunächst essentielle Bestandteile der P4-Architektur und - damit einhergehend - im Code erklärt. Hierbei wird bereits auf die Unterschiede zwischen virtueller (BMv2) und realer Hardware (TNA) eingegangen, bevor diese anhand der Übungen verdeutlicht werden. Am Ende des Kapitels sollen die Möglichkeiten der Sprache anhand eines selbst entworfenen Protokoll-Headers veranschaulicht werden.

In Kapitel 3 werden die sechs bereits erstellten Übungen erklärt. Hierzu zählen u.a.:

- Kriterien zur Weiterleitung der Netzwerkpakete
- Implementierte Netzwerkprotokolle
- Netzwerktopologien

Außerdem wird ausgeführt, wie die VM installiert und welche Schritte in dieser nötig sind, um diese gemäß des Designs der Übungen zu konfigurieren (z.B. Netzwerkkonfiguration in Mininet⁴, Initialisierung der Hosts).

Die Portierung des Codes von BMv2 auf TNA wird im Kapitel 4 betrachtet. Außerdem wird der Aufbau des P4-Labor-Netzes vorgestellt. Anschließend werden die Auswirkungen (der Portierung) auf den Programmcode und die Konfiguration der restlichen Hardware ausgearbeitet.

Das Testen des Codes und der Netze wird in Kapitel 5 erklärt. Hierzu zählt auch die richtige Benutzung der Hosts. Rudimentäres Prüfen auf Funktionalität kann mittels *ICMP* vollzogen werden. Mit Hilfe des Packet-Manipulations-Programms Scapy oder iPerf⁵ können extrem genau angepasste Testszenarien erstellt und geprüft werden. In Kapitel 6 gebe ich ein Fazit zur Arbeit und einen Ausblick in die Zukunft des Projektes.

³<https://scapy.net/>

⁴<http://mininet.org/>

⁵<https://iperf.fr/>

Kapitel 2: Grundlagen von P4

Dieses Kapitel gibt Auskunft über den geschichtlichen Hintergrund von P4 und welche Auswirkungen es auf unsere heutigen Netzwerke haben kann. Außerdem werden der grundlegende Aufbau eines P4-Programms und einige Hinweise zur Erstellung des Codes gegeben. Da die Vervielfältigung von Paketen für das Forschungsprojekt und die Übungen ein wichtiger Bestandteil ist, werden verschiedene Möglichkeit für die Replikation von Netzwerkpaketen kurz erklärt. Das Ende des Kapitels beschäftigt sich anhand eines selbst entworfenen Protokoll-Headers mit den Möglichkeiten, die durch P4 entstehen. Netzwerkpakete bestehen aus ihren Nutzdaten (engl. *payload*) und den Kontrolldaten, die vor (dann engl. *header*) oder nach (dann engl. *footer* oder *trailer*) den Nutzdaten in der jeweiligen Schicht des OSI-Schichtmodells¹ enkapsuliert werden. Die *Nutzdaten* sind die vom User gewünschten Daten wie z.B. der Inhalt einer Website oder ein TV-Stream. *Header*, *Footer* und *Trailer* beinhalten Daten von Netzwerkprotokollen. Im Data-Link-Header (Layer 2) sind bei Ethernet beispielsweise die *Media Access Control*(MAC)-Adresse und im Networking-Header (Layer 3) beim *Internet Protocol* (IP) die IP-Adressen von Sender und Empfänger enthalten [s. [RFC791]].

2.1 Geschichtlicher Hintergrund

Die Programmiersprache P4 wird seit 2013 entwickelt. Ursprünglich sollte sie in Verbindung mit *Software Defined Networking* (SDN) die Konfiguration von Netzwerken erleichtern und als Orientierung dafür dienen, wie sich *OpenFlow*² weiterentwickeln sollte. *OpenFlow* verfügte zu diesem Zeitpunkt nicht über die gewünschte Flexibilität zum Einbinden weiterer Protokoll-Header. Anstatt - wie bisher - weitere Protokoll-Header in *OpenFlow* zu importieren (innerhalb von vier Jahren wuchs die Anzahl der Header-Felder von 12 auf 41), verfolgten Sie bereits in der ersten Veröffentlichung zu P4 den Ansatz, dass Switches zukünftig über flexible Mechanismen zum Parsen von Netzwerkpaketen (kurz: Pakete) und deren Weiterleitung verfügen sollten. Schon bereits damals gab es Computerchips, welche diese Flexibilität und das Switchen in Terabit-Geschwindigkeiten beherrschten. [s. [Bos14]]

Um Daten möglichst schnell übertragen zu können, sind in industriellen Switchen und Routern sogenannte *Fixed-function ASICs* verbaut. Der Nachteil dieser ist, dass sie von Fabrik aus viele Netzwerkprotokolle verarbeiten müssen, auch wenn diese eventuell

¹<https://www.netzwerke.com/OSI-Schichten-Modell.htm>

²<https://opennetworking.org/>

gar nicht eingesetzt werden. Außerdem dauert es erfahrungsgemäß lange bis neue Protokolle in die ASICs eingearbeitet werden (~ 3 Jahre bei Cisco mit Release-Cycles) und infolge dessen noch länger bis sie auf den Geräten genutzt werden können, da neue Hardware erst gekauft werden muss.

Aus den eben genannten Gründen und Überlegungen wurde am 28. Juli 2014 das erste Paper zu P4 veröffentlicht. Aus dessen Titel “**P4**: programming protocol-independent packet processors“ leitet sich auch die Abkürzung P4 ab. Die Sprache wurde zwischenzeitlich neu überarbeitet, wodurch die aktuelle Version *P4₁₆* entstand. Zur eindeutigen Abgrenzung wird die ältere Version seitdem *P4₁₄* genannt. Im Zuge dessen wurde die Sprache stark vereinfacht (nur noch 40 statt 70 Keywords³) und viele Konstrukte in Bibliotheken ausgelagert. Diese Arbeit beschäftigt sich ausschließlich mit der aktuellen Version *P4₁₆*, welche nicht kompatibel zur älteren Version ist. [s. [P4 Lang. Spec. Abschnitt 3.2]]

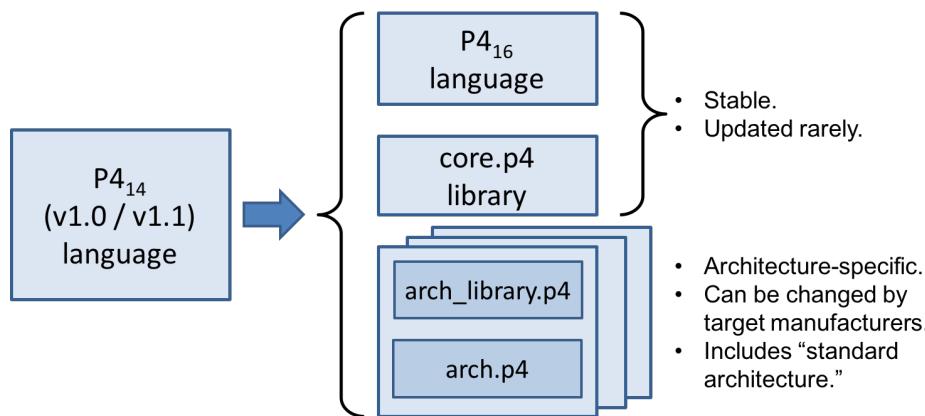


Abbildung 2.1: Änderungen von *P4₁₄* zu *P4₁₆* [P4 Lang. Spec. Figure 3]

P4 wurde ursprünglich für den Einsatz auf Switchen entwickelt, wurde aber so erweitert, dass es verschiedene Geräte wie auch Router, FPGAs und programmierbare ASICs unterstützt. Alle eben genannten Geräte werden in dieser Arbeit unter dem Begriff *Target* zusammengefasst. Im praktischen Teil dieser Arbeit sind damit entweder der virtuelle BMv2-Switch oder der Intel Tofino⁴ Chip im Edge-Core Wedge 100BF-32X gemeint.

2.2 Zielsetzungen von P4

P4 soll drei Ziele erfüllen:

1. Rekonfigurierbarkeit
2. Protokollunabhängigkeit
3. Unabhängigkeit von der Zielplattform

³vgl. bei C: int, double, struct, ...

⁴<https://www.intel.de/content/www/de/de/products/network-io/programmable-ethernet-switch/tofino-series.html>

Diese bedeuten folgendes:

1. Das Verhalten des Gerätes, wie Pakete weitergeleitet werden, kann selbst nach dem Deployen⁵ des Programms verändert werden.
2. Die ausführende Instanz soll nicht an spezifische Paketformate gebunden sein, sondern anhand der vom Parser extrahierten Headerdaten und definierten Match-Action-Tabellen (s. Abschnitt 2.3.4 Tabellen) arbeiten.
3. Eine zielarchitekturunabhängige Beschreibung (P4-Code) soll vom Compiler in ein zielarchitekturabhängiges Programm übersetzt werden [vgl. [Bos14]]. Ähnlich wie bei der Programmiersprache C soll sich eine P4-Programmiererin oder ein P4-Programmierer nicht mit den hardwarespezifischen Details des Zielgeräts befassen müssen.

2.3 Allgemeines Funktionsprinzip

Die P4-Architektur arbeitet mit *Programmierbaren Blöcken* oder *Programmable Blocks*. Hierzu gehören *Parser*, *Deparser*, *Ingress Control Flow* und *Egress Control Flow*. Sobald ein Paket in das Target eintritt, wird es mit den sogenannten *Intrinsic Metadata* initialisiert. Diese sind hardwareabhängige Informationen wie bspw. der Port⁶, an dem das Paket empfangen wurde oder der *Timestamp*⁷. Außerdem besteht zusätzlich die Möglichkeit dem Paket, während es vom Target verarbeitet wird, *benutzerdefinierte Metadaten* hinzuzufügen. Diese können sich bei der Entscheidung „wie das Paket weiterzuleiten sei“ als nützlich erweisen. Die nachfolgende Abbildung 2.2 veranschaulicht diese Teile der Architektur.

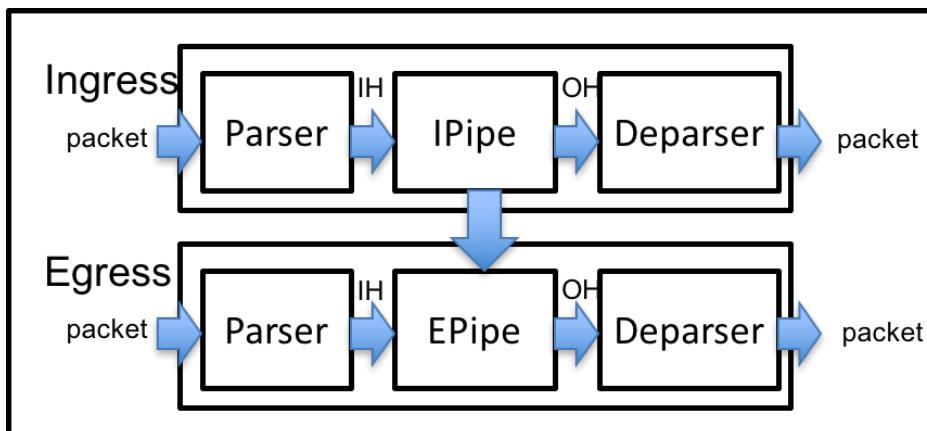


Abbildung 2.2: Schnittstellen eines P4-Programms[P4 Lang. Spec. Figure 12]

P4 hat eine C-ähnliche Syntax und .p4 als Dateiendung. Kommentare werden mit // oder /* KOMMENTAR */ geschrieben.

⁵von engl. = anwenden; Übertragen und Ausführen des kompilierten Programms auf das Ziel

⁶physische Netzwerkschnittstelle des Targets

⁷die Zeit, zu der das Paket empfangen wurde

2.3.1 Genereller Aufbau der beiden Architekturen

Die beiden Architekturen TNA (s. Abschnitt TNA) und BMv2 (s. Abschnitt BMv2) beinhalten grundlegend die selben Kontrollblöcke. Dennoch gibt es Unterschiede im Aufbau. Deshalb werden diese hier dargestellt und in den darauf folgenden Punkten erläutert. Der allgemeine Weg bei der Verarbeitung der Daten eines Pakets ist in Abbildung 2.3 zu sehen. Interessant ist hierbei, dass die *Nutzdaten (Payload bzw. Full Packet Data* in Abbildung 2.3) nicht in der *Match-Action* verarbeitet werden. Nur die *Header* spielen für diesen *Kontrollblock* eine Rolle bei der Verarbeitung und Weiterleitung. In der Abbildung sind auch die eigenen *Metadaten* und *Intrinsic Metadata* ersichtlich. Diese werden getrennt vom eigentlichen Paket im *Meta-Data-Bus* zwischen den verschiedenen *Kontrollblöcken* ausgetauscht.

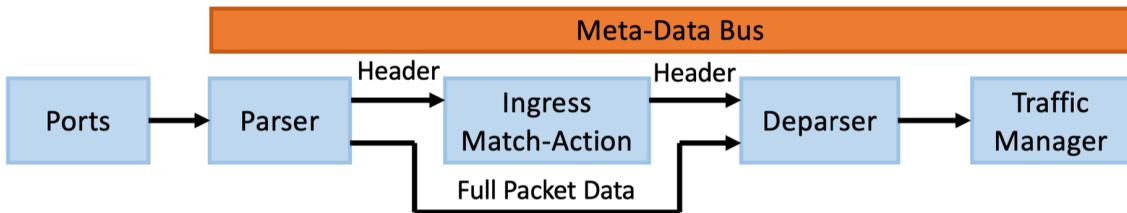


Abbildung 2.3: Packet-Flow in P4 (Quelle: Prof. Dr. Thomas Scheffler, HTW Berlin)

BMv2

Das *Behavioral Model Version 2* (BMv2) ist ein softwarebasierter Switch des P4-Language-Consortiums. Das Verhalten der Paket-Verarbeitung wird in diesem durch das P4-Programm bestimmt. BMv2 ist nicht dafür ausgelegt in einem produktiven Umfeld eingesetzt zu werden. Es soll als Tool für Entwicklung, Tests und Debugging genutzt werden. [s. [BMv2]] BMv2 verfügt über folgende sechs Kontrollblöcke. Diese werden im Programm wie in Codeauszug in diesem Abschnitt auf dem Target genutzt.

1. Parser
2. Checksum Verifikation
3. Ingress Processing
4. Egress Processing
5. Checksum Berechnung
6. Deparser

Codeauszug 2.1: Aufbau Kontrollblöcke in BMv2

```

1 /***** S W I T C H *****/
2 *****
3 *****
4
5 V1Switch(
6 MyParser(),
7 MyVerifyChecksum(),
8 MyIngress(),
9 MyEgress(),
10 MyComputeChecksum(),
11 MyDeparser()
12 ) main;

```

TNA

Die *Tofino Native Architecture* (TNA) ist eine P4-Architektur, welche von Intel für deren P4-fähige Hardware entwickelt wurde. Sie wird z.B. auf dem Tofino-Chip genutzt. TNA verfügt über ebenfalls sechs Kontrollblöcke. Diese können in zwei logische Blöcke unterteilt werden: Ingress und Egress-Block. Dabei sind die beiden Blöcke nach folgendem Schema aufgebaut:

1. Parser
2. Match-Action
3. Deparser

Die Blöcke werden in einer *Pipeline* zusammengefasst und über den Befehl in Zeile 11 in die jeweilige Pipe des Targets geladen. Das Target kann über 1-4 verschiedene oder gleiche Pipelines verfügen. Diese müssten in Zeile 11 nach `pipe` mit Komma getrennt eingefügt werden (s. folgender Codeauszug).

Codeauszug 2.2: Aufbau Kontrollblöcke in TNA

```

1 *****
2 Pipeline(
3     IngressParser(),
4     Ingress(),
5     IngressDeparser(),
6     EgressParser(),
7     Egress(),
8     EgressDeparser()
9 ) pipe;
10
11 Switch(pipe) main;

```

Wichtige Bestandteile der TNA sind auch der *Traffic Manager* (TM) und die darin enthaltene *Packet Replication Engine* (PRE). Der *TM* speichert Pakete im Buffer bzw. der Queue bis diese gesendet werden und setzt die dafür nötigen Befehle aus dem Code um wie z.B. das Setzen des Egress-Ports. Die *PRE* ist für die Vervielfältigung von Paketen (beispielweise bei Multicast) verantwortlich. [s. [TNA]]

TM und *PRE* sind nicht aus dem Programm konfigurierbar. Die Daten, welche beide für das Weiterleiten nutzen, beziehen sie aus den *Intrinsic Metadata* sowie Einträgen der *Tabellen* (s. Abschnitt 2.3.4).

2.3.2 wichtige Datentypen und Keywords

Nachfolgende Tabelle 2.1 listet wichtige Datentypen und Keywords von P4 auf. Zusätzlich zu diesen Datentypen gibt es sogenannte *Externs*. Dies sind vordefinierte Funktionen, welche z.B. einen *Mirror* erzeugen können (s. Abschnitt 4.3.3).

Name	P4-Code	Erklärung
Boolean	<code>bool</code>	wie in anderen Programmiersprachen: Werte: true oder false
Integer	<code>int</code>	32-bit große Ganzzahl
Bit	<code>bit<x></code>	Datentyp, der genau X Bit lang ist X ist eine positive Ganzzahl
Error	<code>error</code>	Datentyp, der verschiedene Fehler signalisiert
Struct	<code>struct</code>	Erstellen eigener Datentypen
Definition	<code>#define</code>	ähnlich wie bei C: prozedurale Anweisung auch vergleichbar mit Alias bei Linux
Typdefinition	<code>typedef</code>	Erstellen von wichtigen/viel genutzten Typen wie z.B. IPv4-Adresse
Header	<code>header</code>	Datentyp, um Protokollheader zu definieren
Konstante	<code>const</code>	Konstante gefolgt von Datentyp, Name und Wert
Kontrollblock	<code>control</code>	alle P4-Kontrollblöcke außer der Parser
Parser	<code>parser</code>	Deklaration Parser
Tabelle	<code>table</code>	Deklaration Tabelle
Funktion	<code>action</code>	vergleichbar mit Funktionen aus anderen Programmiersprachen

Tabelle 2.1: P4-Datentypen [s. [P4 Lang. Spec. Abschnitt 7]]

2.3.3 Parser

Der erste Kontrollblock eines P4-Programms - der *Parser* - schreibt zunächst hardwarespezifische Daten (z.B. den eingehenden Port) des eingehenden Pakets in die *Intrinsic Metadata*. Der *Parser* ist als State-Machine zu benutzen. Hierbei müssen alle möglichen Header, die das Target verstehen und bearbeiten kann, zu Beginn definiert, nach festgelegten Kriterien

verarbeitet und gespeichert werden (solange das Paket noch nicht weitergeleitet wurde). Wird ein Header geparsst, wird dieser Header validiert. Dies ist wichtig für die *Kontrollblöcke* in den Abschnitten 2.3.4 (Match-Action und Control Flow) und 2.3.5 (Deparser), damit diese wissen, ob ein bestimmter Header beim aktuellen Paket existiert oder nicht.

Als Veranschaulichung dient nachfolgender Parser-Code aus der Übung 3.3.6 (Resubmit3-ARP) in der virtuellen Umgebung. Bei diesem wurden Header für Ethernet und IPv4 angelegt und im `struct headers` hinzugefügt. Daraufhin folgt der Code für die State-Machine beginnend mit dem obligatorischen Zustand `start`, welcher die *Intrinsic Metadata* initialisiert. Durch Benutzung des Keywords `transition` weist man den Parser an bei welchem `state` er fortfahren soll. Diese `transition` kann durch ein `select` ergänzt werden, wodurch es einem `switch case` in C gleicht. Mit dem Befehl `pkt.extract(hdr.X)`⁸ werden erneut Headerdaten des Pakets extrahiert. Im `state parse_ethernet` ist zu erkennen, dass die `select`-Bedingung anhand des *Ethertypes* im Ethernet-Header (Layer3) verendet wird. Der *Ethertype* signalisiert, welches Protokoll im nächsten Header (Layer3) verwendet wird. Die zwei Möglichkeiten wurden zu Beginn des Beispielcodes definiert. Sollte keine der definierten Bedingungen erfüllt sein, tritt der `default`-Zustand ein.

Codeauszug 2.3: Parsercode aus Resubmit3-ARP

```

1 [...]  

2 // Ethertype ;)  

3 const bit<16> TYPE_IPV4 = 0x800;  

4 const bit<16> TYPE_ARP = 0x806;  

5 [...]  

6 header ethernet_t {  

7     macAddr_t dstAddr;  

8     macAddr_t srcAddr;  

9     bit<16> etherType;  

10 }  

11  

12 header ipv4_t {  

13     bit<4> version;  

14     bit<4> ihl;  

15     bit<8> diffserv;  

16     bit<16> totalLen;  

17     bit<16> identification;  

18     bit<3> flags;  

19     bit<13> fragOffset;  

20     bit<8> ttl;  

21     bit<8> protocol;  

22     bit<16> hdrChecksum;  

23     ip4Addr_t srcAddr;  

24     ip4Addr_t dstAddr;  

25 }  

26 [...]

```

⁸X: Platzhalter für den Headertype z.B. „IPv4“

```

27 struct headers {
28     ethernet_t ethernet;
29     ipv4_t ipv4;
30 }
31 [...]
32     state start {
33         transition parse_ethernet;
34     }
35
36     state parse_ethernet {
37         packet.extract(hdr.ethernet);
38         transition select(hdr.ethernet.etherType){
39             TYPE_IPV4: parse_ipv4;
40             TYPE_ARP: arp_fwd;
41             default: accept;
42         }
43     }
44
45     state parse_ipv4 {
46         packet.extract(hdr.ipv4);
47         transition accept;
48     }
49
50     state arp_fwd {
51         transition accept;
52 }
```

[...] signalisiert hier und im Laufe der Arbeit, dass Code aus dem Programm für die Darstellung ausgelassen wurde. Der vollständige Programmcode ist im angehängten zip-Ordner einsehbar (s. Anhang J).

Das Keyword `accept` signalisiert dem Parser das Beenden seiner erfolgreichen Ausführung. Das Gegenstück dazu bildet das Keyword `reject`, welches einen Parser-Fehler andeutet. Das jeweilige Verhalten des Programms in diesen beiden Zuständen ist architekturabhängig. Daraufhin wird das Programm mit dem nächsten *Kontrollblock* beginnen. Im Parser können Metadaten bereits initialisiert werden. `if..else`-Bedingungen und *Actions* (s. Abschnitt 2.3.4) können im *Parser* nicht verwendet werden.

Der grundlegende Aufbau des Parsers in der virtuellen Umgebung sieht wie folgt aus:

Codeauszug 2.4: Einfacher BMv2 Parser

```

1 parser MyParser(packet_in packet,
2                     out headers hdr,
3                     inout metadata meta,
4                     inout standard_metadata_t standard_metadata) {
5
6     state start {
```

```

7     transition accept;
8 }
9 }
```

Grundlegender Aufbau in TNA:

Codeauszug 2.5: Einfacher TNA Parser

```

1 parser MyIngressParser(packet_in pkt,
2         out headers hdr,
3         out metadata meta,
4         out ingress_intrinsic_metadata_t ig_intr_md) {
5
6     state start {
7         pkt.extract(ig_intr_md);
8         transition accept;}
9 }
```

Tabelle 2.2 listet die vier Parameter des Parsers auf und zeigt Unterschiede in der Nomenklatur der beiden Architekturen.

Parameter	Richtung	Architektur	
		BMv2	TNA
eingehendes Paket	ohne Richtung		packet_in
Headerdaten	Out		headers
Metadata	Out		metadata
Intrinsic Metadata	Out	standard_metadata_t	ingress_intrinsic_metadata_t

Tabelle 2.2: Auflistung Parameter im Parser

Die „Richtung“ dieser Daten bedeutet folgendes:

Richtung	Bedeutung
in	Daten können nur gelesen werden
inout	Daten können gelesen und geschrieben werden
out	Daten können nur geschrieben werden

Tabelle 2.3: Richtung der Parameter

Sowohl die Reihenfolge, in der diese Daten bei der Deklarierung gesetzt werden, als auch die Richtung sind unveränderbar im jeweiligen Block. Dies wird in den folgenden Blöcken noch einmal deutlich. Der Name der *Kontrollblöcke* und *Parameter* ist frei wählbar.

2.3.4 Match-Action und Control Flow

Generelles

Auf den *Parser* folgt der *Match-Action-*, *Control-Flow*- oder allgemein *Processing-Block*. Der letzte Begriff vereint die ersten beiden, wobei alle drei genutzt werden können. In diesem Block werden Entscheidungen zur weiteren Verarbeitung der Pakete getroffen (*Match-Action*). Dies kann ein einfaches Discarden⁹, Setzen des ausgehenden Ports oder auch eine Anwendung einer Tabelle wie z.B. der Routingtabelle sein. Die Routing-Tabelle ist eine Tabelle innerhalb von Routern, welche der Router nutzt, um Forwarding-Entscheidungen zu treffen. Sie enthält Netzadressen samt Subnetzmaske und Outgoing-Port. *Control-Flow* bezieht sich auf die Reihenfolge, in der *Actions*, Bedingungen und *Tabellen*-Referenzen bearbeitet werden. [s. [Bos14, Abschnitt 4.6]] Nachfolgend sind aus der Übung 3.3.1 (Hello World) der *Match-Action*-Code für beide Architekturen zu sehen. In den Programmen wird die Entscheidung für das Weiterleiten der Pakete anhand des eingehenden Ports (if-Bedingung: `ig_intr_md.ingress_port`) getroffen. Daraufhin wird der *Egress-Port egress_spec* (ausgehender Port) gesetzt. Port bezeichnet hier eine physikalische Schnittstelle - auch *Interface* genannt - eines *Targets*.

BMv2:

Codeauszug 2.6: Einfache BMv2 Match-Action

```
1 control MyIngress(inout headers hdr,
2                     inout metadata meta,
3                     inout standard_metadata_t standard_metadata) {
4
5     apply {
6         if (standard_metadata.ingress_port == 1) {
7             standard_metadata.egress_spec = 2;
8         }
9         else if (standard_metadata.ingress_port == 2) {
10            standard_metadata.egress_spec = 1;
11        }
12    }
13 }
```

TNA:

Codeauszug 2.7: Einfache TNA Match-Action

```
1 control MyIngress(inout headers hdr,
2                     inout metadata meta,
3                     in ingress_intrinsic_metadata_t ig_intr_md,
4                     in ingress_intrinsic_metadata_from_parser_t ig_prsr_md,
5                     inout ingress_intrinsic_metadata_for_deparser_t ig_dprsr_md,
```

⁹engl. to discard something = etwas verwerfen; Paket wird nicht weitergeleitet

```

6           inout ingress_intrinsic_metadata_for_tm_t ig_tm_md)
7 {
8 // 134 = QSFP1-3; 135 = QSFP1-4
9 apply {
10     if (ig_intr_md.ingress_port == 134) {
11         ig_tm_md.unicast_egress_port = 135;
12     }
13     else if (ig_intr_md.ingress_port == 135) {
14         ig_tm_md.unicast_egress_port = 134;
15     }
16 }
17 }

```

Auffällig ist beim Vergleich des Codes, dass in TNA sechs Parameter an den *Match-Action-Kontrollblock* übergeben werden. Die vier letzten Parameter werden in BMv2 in einem Parameter vereint. Außerdem haben Parameter (im Vergleich zum Parser) ihre Richtungen geändert. Der Aufbau der einzelnen Blöcke kann für die TNA im File `tofino1_arch.p4`¹⁰ nachgelesen werden. Der Inhalt jeglicher intrinsischen Daten wie etwa Variablenname oder Datentyp ist in der Datei `tofino1_base.p4`¹¹ zu finden. Die Dateien sind im Github von Barefootnetworks¹² abrufbar.

Actions

Funktionen werden in P4 *Actions* genannt und über das selbige Keyword deklariert. Dabei ist es möglich, dass *Actions* ohne und mit Parametern erzeugt werden können. Dies ist an den beiden nachfolgenden *Actions* zu sehen. Der/die Übergabeparameter werden dabei direkt in den Code geschrieben oder aus einer Tabelle ausgelesen (siehe folgender Abschnitt Tabellen). Die `action drop` veranlasst das Target das Paket zu verwerfen und nicht weiterzuleiten. Die `action send` setzt den gewünschten Egress-Port für das Paket.

Codeauszug 2.8: Zwei Actions

```

1 action drop() {
2     ig_dprsr_md.drop_ctl = 1;
3 }
4
5 action send(PortId_t port) {
6     ig_tm_md.unicast_egress_port = port;
7 }

```

Actions können kaskadiert, lokale Variablen initialisiert und *if-Bedingungen* genutzt werden. P4 verfügt über einige vordefinierte *Actions* [s. [Bos14, Abschnitt 4.5]], welche in Tabelle

¹⁰https://github.com/barefootnetworks/Open-Tofino/blob/master/share/p4c/p4include/tofino1_arch.p4

¹¹https://github.com/barefootnetworks/Open-Tofino/blob/master/share/p4c/p4include/tofino1_base.p4

¹²Barefoot Networks entwickelte den Tofino Chip und wurde später von Intel aufgekauft

2.4 kurz erklärt werden:

Name	Erklärung
<code>set_field</code>	Wert für spezifisches Feld im Header setzen; Masken werden unterstützt
<code>copy_field</code>	Kopieren eines Feldes in ein anderes
<code>add_header</code>	Header inkl. Felder gültig setzen
<code>remove_header</code>	Löschen eines Headers inkl. Felder aus dem Paket
<code>increment</code>	In- oder Dekrementieren eines Wertes im Feld
<code>checksum</code>	Checksum von einem Header berechnen

Tabelle 2.4: Vordefinierte Actions

Tabellen

Tabellen sind in P4 der bevorzugte Weg, um Forwarding-Entscheidungen zu treffen. Diese werden mit dem Keyword `table` und einem frei wählbaren Namen deklariert. Hierbei muss mindestens ein `Key` mit *Match-Kriterium*, `Actions`, `Default-Action` und `size` (= Größe) angegeben werden. Sie werden mit `TABELLENNAME.apply()` aufgerufen. Tabellen müssen via Control-Plane - also außerhalb des P4-Codes - befüllt werden. Der `Key`¹³ ist dabei das zu überprüfende Kriterium. Er wird mit den Key-Einträgen der jeweiligen Tabelle verglichen. Als `Key` können alle in der Match-Action vorkommenden Daten verwendet werden, die gelesen werden können.¹⁴ Die drei Match-Kriterien sind in Tabelle 2.5 erklärt.

Match-Kriterium	Erklärung
<code>exact</code>	alle Bits müssen exakt übereinstimmen
<code>lpm</code>	Longest Prefix Match: der Key mit den meisten übereinstimmenden und aufeinanderfolgenden Bits wird ausgewählt ¹⁵
<code>ternary</code>	mit Angabe einer Maske ¹⁶ : die Maske bestimmt, welche aufeinanderfolgenden Bits zwischen Wert und Key verglichen werden

Tabelle 2.5: Match-Action Kriterien

Codeauszug 2.9: Tabelle `ipv4_host` und Ausführen der Tabelle

```

1  table ipv4_host {
2      key = { hdr.ipv4.dst_addr : exact; }
3      actions = {
4          set_nexthop;
5          @defaultonly NoAction;

```

¹³der Daten des aktuell zu bearbeitenden Pakets

¹⁴Daten mit Richtung `in` oder `inout`

¹⁵vgl. LPM bei Routing

¹⁶vgl. Subnetzmaske: die Anzahl der mindestens übereinstimmenden, aufeinanderfolgenden Bits

```

6     }
7     const default_action = NoAction();
8     size = IPV4_HOST_TABLE_SIZE;
9 }
10 [...]
11 ipv4_host.apply();

```

Die zuvor deklarierten Actions werden im entsprechenden Abschnitt der Tabelle hinzugefügt; analog dazu die gewählte Default-Action. Die Default-Action tritt ein, wenn kein passender Eintrag in der Tabelle gefunden wird. Die size gibt hierbei an, wie viele Einträge die jeweilige Tabelle speichern kann. Im Beispiel ist erkennbar, dass dort nicht zwingend eine Ganzzahl, sondern auch zuvor definierte Werte (`#define`) angegeben werden können. In den ersten 12 Zeilen des nachfolgenden Codeauszugs ist der Output der Tabelle `mirror.cfg` dargestellt. Capacity entspricht hierbei der zuvor konfigurierten size. Usage gibt die Anzahl bereits erstellter Einträge in der Tabelle an. Die Spezifikation der Tabelle erlaubt dem Kompiler zu entscheiden, wie viel Speicher er für diese in *Ternary Content-Addressable Memory* (TCAM) oder *Static Random Access Memory* (SRAM) bereitstellen muss. [s. [Bos14, Abschnitt 4.4]] Diese Tabelle wurde in der *Barefoot Runtime* (BFRT) ausgegeben (s. Abschnitt 4.2.1).

Informationen über den Key sind in Zeilen 9-12 zu finden. Anschließend wird der erste Eintrag in der Tabelle samt den Funktionsparametern (inkl. der gesetzten Werte) ausgegeben. In Zeile 20 ist ersichtlich, dass bei diesem Key die Action `normal` ausgeführt wird. Weitere Informationen zu diesem Abschnitt können in der [P4 Runtime Spec. Abschnitt 9] gefunden werden.

Codeauszug 2.10: Ausgabe Info und Ausgabe eines Eintrags mittels `dump()`

```

1 bfrt.mirror.cfg> info
2 -----> info()
3 Table Name: cfg
4 Full Name: mirror.cfg
5 Type: MIRROR_CFG
6 Usage: 1
7 Capacity: 1024
8
9 Key Fields:
10 Name Type Size Required Read Only
11 ----- -----
12 $sid EXACT 16 True False
13
14 bfrt.mirror.cfg> dump
15 -----> dump()
16 ---- cfg Dump Start ----
17 Entry 0:
18 Entry key:
19   $sid : 0x0001

```

```
20 Entry data (action : $normal):
21   $session_enable : True
22   $direction : BOTH
23   $uicast_egress_port : 0x00000028
24   $uicast_egress_port_valid : True
25   $egress_port_queue : 0x00000000
26   $ingress_cos : 0x00000000
27   $packet_color : GREEN
28   $level1_mcast_hash : 0x00000000
29   $level2_mcast_hash : 0x00000000
30   $mcast_grp_a : 0x0000
31   $mcast_grp_a_valid : False
32   $mcast_grp_b : 0x0000
33   $mcast_grp_b_valid : False
34   $mcast_l1_xid : 0x0000
35   $mcast_l2_xid : 0x0000
36   $mcast_rid : 0x0000
37   $icos_for_copy_to_cpu : 0x00000000
38   $copy_to_cpu : False
39   $max_pkt_len : 0x0000
40
41 ----- cfg Dump End -----
```

2.3.5 Deparser

Der *Deparser* stellt das Gegenstück zum *Parser* dar und wird nach der *Match-Action* angewandt. Er bereitet die zuvor extrahierten und veränderten Werte des Pakets für das Senden vor indem er sie wieder zusammensetzt. Der *Deparser* hat in BMv2 lediglich diese Rolle. In TNA wird er auch dazu benutzt, um Pakete vervielfältigen zu lassen oder bspw. Prüfsummen¹⁷ neu zu berechnen. Verifizieren und Neuberechnen der Checksum sind in BMv2 in eigenen Kontrollblöcken angelegt (s. Abschnitt 2.3.6).

2.3.6 weitere Blöcke

Dieser Abschnitt bezieht sich auf die beiden Checksum-Kontrollblöcke der BMv2. In TNA sind diese als externe Funktionen deklariert und sowohl in Ingress-Parser als auch Egress-Deparser anwendbar. Die Checksum ist ein 16-bit langes Feld im IPv4-Header (s. Abschnitt 2.4). Sie dient dazu Überprüfen zu können, ob ein Paket ohne Fehler übertragen oder auch manipuliert wurde.

¹⁷engl.: checksum

Checksum Berechnung

Vor dem Senden eines IP-Pakets wird aus den restlichen Feldern im IPv4-Header eine Checksum berechnet. Dieser Wert wird dann in das dafür vorgesehene Feld des IPv4-Headers eingetragen. Werden Daten im IPv4-Header verändert, muss die Checksum neu berechnet werden.

Checksum Verifikation

Sie wird beim Empfangen eines Pakets durchgeführt. Sollte dabei ein Fehler festgestellt werden, wird das Paket verworfen. [s. [RFC791]]

2.3.7 Paket-Replikation

Um Pakete vervielfältigen zu können, gibt es in P4 verschiedene Möglichkeiten. Sie werden unter dem Begriff *Mirror* bzw. *Mirroring* zusammengefasst. Diese sind:

- Resubmit
- Recirculate
- Clone Ingress to Egress (CI2E)
- Clone Egress to Egress (CE2E)

Diese vier Mechanismen stellen alle unterschiedliche Wege dar, welche das vervielfältigte Paket noch einmal durchlaufen muss, bevor es gesendet wird. Diese sind an nachfolgender Abbildung 2.4 erkennbar:

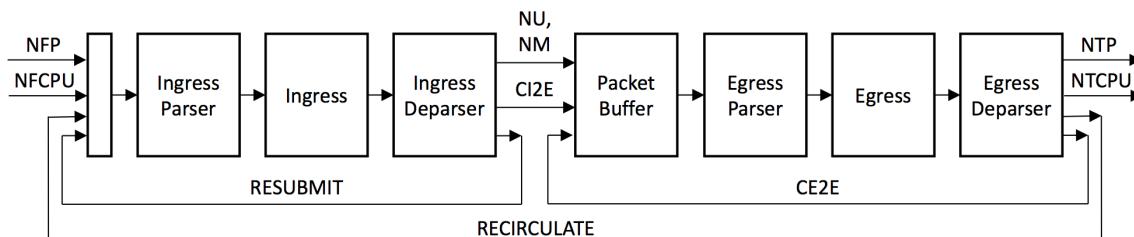


Abbildung 2.4: Paketfluss bei verschiedenen Arten der Vervielfältigung
[s. [PSA, Figure 2]]

2.4 Möglichkeiten von P4

Aufgrund des Ziels *2. Protokollunabhängigkeit* (s. Abschnitt 2.2) sind in P4 standardmäßig keine Netzwerkprotokolle implementiert. Diese müssen anhand der entsprechenden *Request for Comments* (RFC) oder anderer Standards im Code angelegt werden. Als Beispiel dient hierfür der IPv4-Header nach [RFC791]. Dieser ist wie folgt aufgebaut:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31																						
Version	IHL	Type of Service	Total Length																																																		
Identification										Flags	Fragment Offset																																										
Time to Live	Protocol										Header Checksum																																										
Source Address																																																					
Destination Address																																																					
Options																Padding																																					

Der IPv4-Header wird im Programm folgendermaßen angelegt:

Codeauszug 2.11: IPv4 Header in P4

```
1 header ipv4_t {  
2     bit<4> version;  
3     bit<4> ihl;  
4     bit<8> diffserv;  
5     bit<16> totalLen;  
6     bit<16> identification;  
7     bit<3> flags;  
8     bit<13> fragOffset;  
9     bit<8> ttl;  
10    bit<8> protocol;  
11    bit<16> hdrChecksum;  
12    ip4Addr_t srcAddr;  
13    ip4Addr_t dstAddr;  
14 }
```

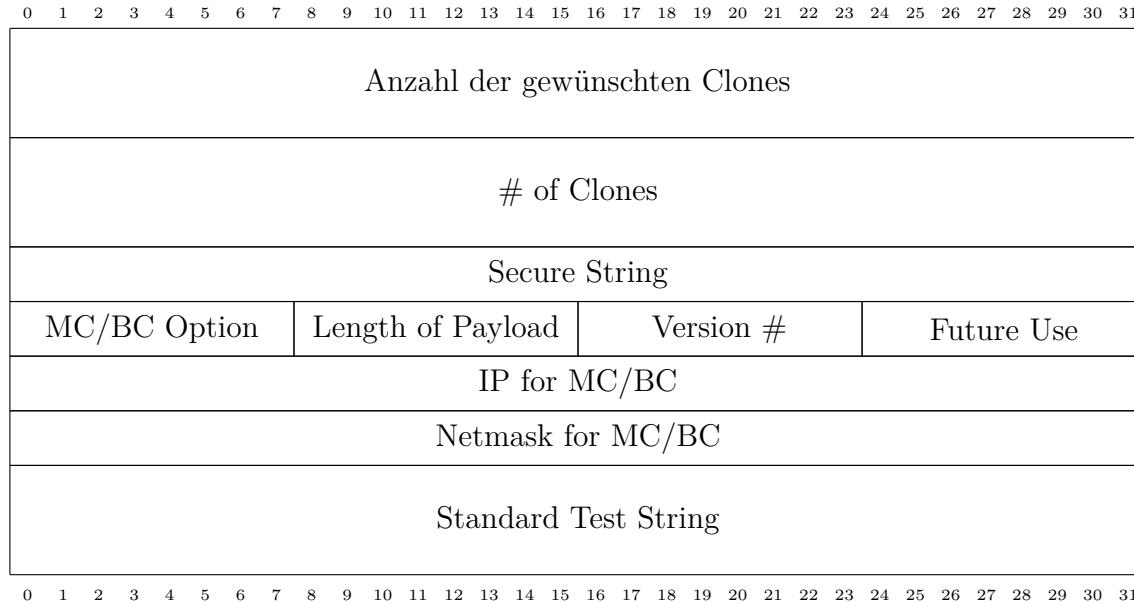
Der Typ `ip4Addr_t` wurde mittels `typedef` (vgl. Abschnitt 2.3.2) angelegt:

Codeauszug 2.12: Erstellung eines eigenen Typs für die IPv4-Adresse

```
1 typedef bit<32> ip4Addr_t;
```

Padding aus dem IPv4-Header muss hierbei nicht, kann aber implementiert werden. Es wird verwendet, um sicherzustellen, dass der Header auch die richtige Länge (→ Vielfaches von 32-bit) hat. Ist der Header kein ganzzahliges Vielfaches von 32-bit, werden fehlende Bits mit Wert=0 (Null) an den Header hinzugefügt bis dieser die richtige Länge erreicht hat. [s. [RFC791]].

Da man also jegliche Standards noch einmal selbst ins Programm einbinden muss, besteht so auch die Möglichkeit Header/Protokolle selbst zu entwerfen und zu nutzen. Diese müssen analog zum IPv4-Header geschrieben werden. Anschließend muss man das Programm anweisen diesen neuen Header in den *Kontrollblöcken* richtig zu verarbeiten. Als Beispiel hierfür dient der selbst erstellte *Clone-Header* aus dem Forschungsprojekt:



Codeauszug 2.13: Clone-Header Implementation im Code

```

1 header clone_t {
2     bit<64> number0fClones;
3     // short: NOC --> shows how many times the packet should be cloned
4     bit<64> cloneCounter;
5     // for initial packet = 0; count +1 for every cloned packet
6     bit<32> secureString;
7     //used for checking whether the packet has permission to be cloned or ←
8         not --> Hannes :-*←
9     bit<8> receiverOption;
10    //identifier whether a packet should be just transmitted to the IP in ←
11        IP-Header(0), to 1 extra host(1), broadcasted(2) or multicasted (3)
12    bit<8> payloadLength;
13    //identifier how long the payload is --> payloadLength * 255 Byte (1st ←
14        idea); payloadLength = 0 --> only 64bit payload field is used
15    bit<8> version;
16    //version identifier
17    bit<8> free;
18    // future use --> maybe identifier for type of retransmission (clone, ←
19        resubmit, recicle, ... )
20    ip4Addr_t receiverIP;
21    //shows Unicast, Multicast or Broadcast address
22    bit<32> receiverMask;
23    //Netmask for receiverIP
24    bit<64> standardString;
```

```
21 //preconfigured Test String aka "Hallo ihr"
22 }
```

Dieser Header wurde für Layer 4 im OSI-Schichtmodell konzipiert und baut auf Ethernet (Layer2) und IPv4 (Layer3) auf. Im Header wird eingetragen wie oft ein Paket geklont werden soll (`Anzahl der gewünschten Clones`). Das folgende Feld `# of Clones` dient zur Überprüfung, wie oft das Paket bereits vervielfältigt wurde. Stimmen die Zahlen in diesen beiden Feldern überein, so wurde das Paket oft genug geklont. Danach wird kein Klon mehr erstellt und gesendet. Die restlichen Felder sind vorerst nur geplant und sind in den Kommentaren im Code erklärt.

Als Identifier¹⁸ dient hierbei das Feld *Protocol* des IPv4-Headers. Der Wert `253` (vgl. Codeauszug 2.14, Zeile 3) aus der Liste der Protokolle in [IP-Proto-Nr.] wurde verwendet, um dem Programm zu signalisieren, dass auf Schicht 4 ein selbst entworfener Header - der *Clone-Header* - folgt. Die Werte `253` und `254` wurden speziell für experimentelle und Testzwecke angelegt [s. [RFC3692]].

Codeauszug 2.14: Code für Clone-Pakete

```
1 [...]
2 // IP-Protocol Field
3 const bit<8> PROTOCOL_CLONE = 0xFD; // = 253 binary -> indicates a Clone ←
4     L4 datagram
5 [...]
6 state parse_ipv4 {
7     packet.extract(hdr.ipv4);
8     meta.mymeta.counter = 0;
9     transition select(hdr.ipv4.protocol){
10         PROTOCOL_CLONE: parse_clone;
11         default: accept;
12     }
13         //transition accept;
14
15 state parse_clone {
16     packet.extract(hdr.clone);
17     transition select(hdr.clone.secureString)
18     {
19         SECURE_STRING: accept;
20         default: reject;
21     }
22 }
23 [...]
```

Wie aus der bisherigen Arbeit ersichtlich ist, bietet P4 die Möglichkeit ein komplett selbst definiertes Verarbeiten und Weiterleiten von Paketen zu entwerfen und zu implementieren.

¹⁸identifiziert im Layer3-Header welches Protokoll in Layer4 vorhanden ist

Hardwareressourcen können effizienter genutzt werden, da nur benötigte Protokolle, Actions oder ähnliches eingebunden und Tabellen sehr genau konfiguriert werden (z.B. Größe der Tabelle). Dadurch dass das P4-Programm größtenteils unabhängig von der Hardware geschrieben werden kann und der Compiler für die Verknüpfung von Code und Hardware sorgt, ist auch eine unabhängige Weiterentwicklung von Programmiersprache und Hardware möglich. [s. [P4 Lang. Spec. Abschnitt 3.1.]]

Kapitel 3: Virtuelle Umgebung

Dieses Kapitel beschäftigt sich mit den Übungen, die in der virtuellen Umgebung absolviert werden können. Mit virtueller Umgebung ist hierbei eine *Virtuelle Maschine* (VM) gemeint. Auf dieser befinden sich Programme, welche Netzwerke inkl. Switchen, Hosts (Endgeräte) und anderen Geräten emulieren können (s. Abschnitt 3.4). In 3.1 wird die Installation und in 3.2 die Konfiguration der VM erklärt. In Abschnitt 3.3 werden die sechs Übungen inkl. Netzwerktopologien und Forwarding-Regeln erläutert. Am Ende des Kapitels wird auf die darin enthaltenen Tools und deren Einsatz (3.5) sowie Konfiguration (3.4) eingegangen.

3.1 Installation der VM

Die VM wird vom P4-Language-Consortium bereitgestellt und basiert auf dem Betriebssystem (OS) Ubuntu¹ 20.04. Die aktuelle Version der VM kann hier heruntergeladen werden: [Download Link](#)

Hier sollte eine der Dateien aus dem Reiter *Release VM Image link* und keine der *Development VMs* heruntergeladen werden. Um diese zu installieren, wird eine Virtualisierungssoftware benötigt. Hierfür können bspw. Oracles² *VirtualBox VM Manager*³, VMwares⁴ *VMware Workstation Player*⁵, Microsofts⁶ *Hyper-V*⁷ oder auch die freie Software *QEMU*⁸ verwendet werden. Tabelle 3.1 gibt Auskunft auf welchem Betriebssystem die eben genannten Programme laufen:

	Windows	MacOS	Linux
VirtualBox	✓	✓	✓
Workstation Player	✓	X	✓
Hyper-V	✓	X	X
QEMU	✓	✓	✓

Tabelle 3.1: Verfügbarkeit Virtualisierungssoftware auf OS

¹<https://ubuntu.com/>

²<https://www.oracle.com/>

³<https://www.virtualbox.org/>

⁴<https://www.vmware.com/de.html>

⁵<https://www.vmware.com/de/products/workstation-player/workstation-player-evaluation.html>

⁶<https://www.microsoft.com/de-de/>

⁷<https://learn.microsoft.com/de-de/virtualization/hyper-v-on-windows/>

⁸<https://www.qemu.org/>

Die Installation wird anhand VirtualBox in Windows 7 erklärt. Sie wurde auf die selbe Weise auf Ubuntu durchgeführt. In VirtualBox kann die VM mittels **Datei\Appliance importieren**... oder dem Tastatursymbol **Strg+I**⁹ importiert werden. Die Software nimmt automatisch Einstellungen vor, welche in folgender Abbildung zu sehen sind:

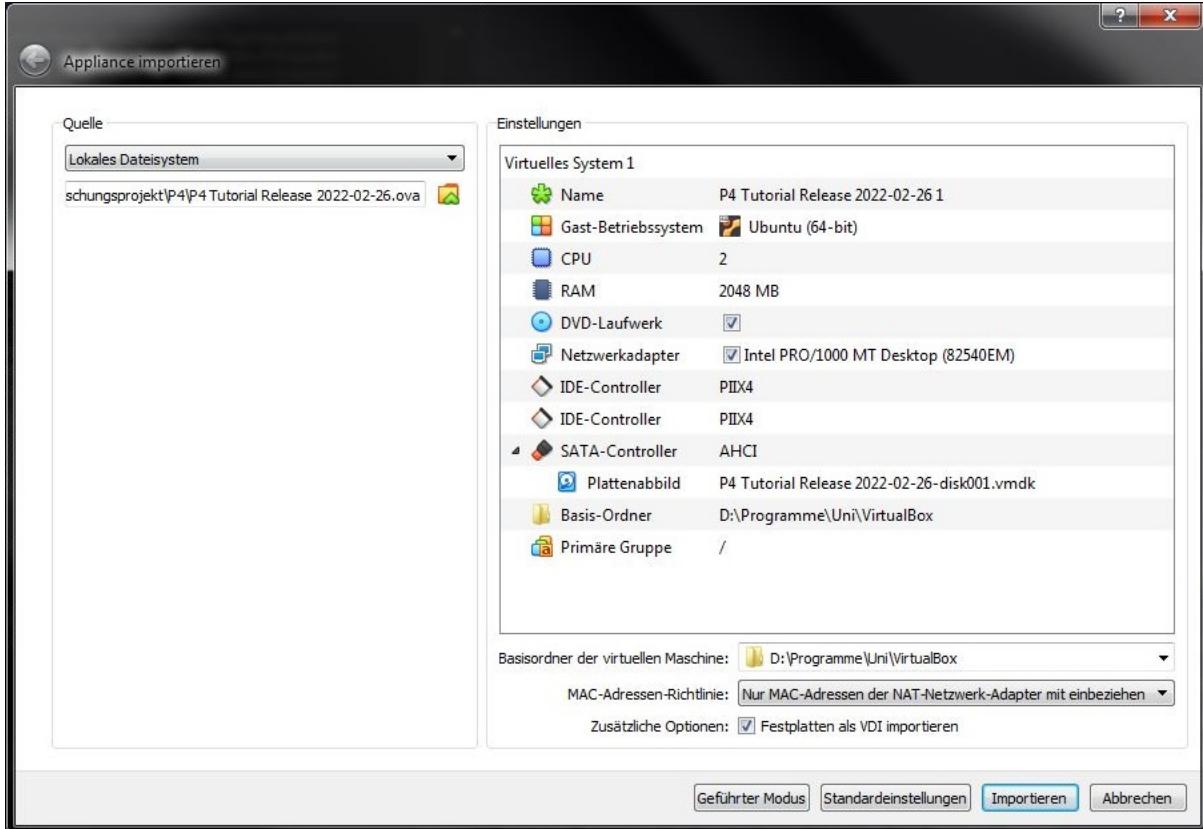


Abbildung 3.1: VM-Settings beim Import in VirtualBox

Diese Einstellungen können direkt übernommen werden. Der **Name** ist dabei frei wählbar. CPU und RAM sollten nicht niedriger, können aber höher eingestellt werden. Der **Netzwerkadapter** ist hierbei von der Hardware des Gerätes abhängig, auf dem VirtualBox genutzt wird. Mit **Basis-Ordner** ist hierbei der Speicherort der VM gemeint. Dieser kann ebenfalls angepasst werden. Diese Einstellungen können nach dem Importieren im Hauptmenü von VirtualBox noch einmal verändert werden (**Hauptmenü → Ändern**). Dafür muss die VM ausgeschaltet sein.

Daraufhin kann die VM mittels **Importieren**-Button importiert werden. Dies kann je nach Hardware der Hostmachine einige Minuten dauern. Bei erfolgreichem Import ist die VM nun im Hauptmenü von VirtualBox zu sehen (vgl. blaues Highlight in Abbildung 3.2). Hier sind auch die getätigten Einstellungen aus Abbildung 3.1 und die restlichen Optionen der VM zu sehen. Anschließend kann die VM mittels **Starten**-Button gestartet werden.

⁹Ctrl+I bei englischsprachigem OS



Abbildung 3.2: VM nach Import in VirtualBox

3.2 Konfiguration der VM

Nach Start der VM loggt man sich mit dem Account P4 und dem Passwort p4 ein. Die Konfiguration und auch das spätere Arbeiten in der VM wird mittels *Terminal* gemacht. Dieses ist auf dem Desktop verlinkt. Im Ordner `~/tutorials` finden sich Übungen, Anleitungen und Lösungen zu den Übungen des P4-Language-Consortiums. Da es sich hierbei um ein Git¹⁰-Repository handelt, können die aktuellen Dateien mittels `git pull` heruntergeladen werden, nachdem man via `cd ~/tutorials` in den Ordner gewechselt ist. Git ist auf der VM vorinstalliert. Zu Beginn des Forschungsprojektes versuchten wir uns anhand dieser Übungen in P4 einzuarbeiten, mussten jedoch schnell feststellen, dass diese für den Einstieg in die Programmiersprache ungeeignet sind, sofern man nicht bereits ein gutes Verständnis über P4 hat. Deshalb werden diese Übungen nicht weiter thematisiert und sind aus Gründen der Vollständigkeit genannt. Sie können zusätzlich zu den später in Abschnitt 3.3 aufgelisteten Übungen absolviert werden.

Um die Übungen, die für diese Arbeit erstellt wurden, zu importieren, empfiehlt es sich einen Ordner via `mkdir ORDNERNAME` zu erstellen und mit `cd ORDNERNAME` in diesen zu wechseln. Im Beispiel heißt der Ordner `p4-htw`. In diesem Ordner wird das Git-Repository (kurz: Repo) des Forschungsprojektes geklont. Das Repo kann an jede Stelle im Dateisystem geklont werden, da dafür noch einmal ein eigener Ordner erstellt wird (hier: `p4-projekt`). Nach Eingabe der Accountdaten (Username und Passwort) wird das Repo geklont. Dafür muss zuvor der Zugriff durch das Forschungsteam erlaubt werden. Anschließend wird in den Repo-Ordner gewechselt und mittels `ls -lisa` die Ordnerstruktur ausgegeben. Das

¹⁰<https://git-scm.com/>

Repo wurde hier mittels *Hypertext Transfer Protocol Secure* (HTTPS) geklont. Das Klonen ist auch via *Secure-Shell* (SSH) möglich.

```
p4@p4:~$ mkdir p4-htw
p4@p4:~$ cd p4-htw/
p4@p4:~/p4-htw$ git clone https://gitlab.rz.htw-berlin.de/s0573019/p4-projekt.git
Cloning into 'p4-projekt'...
Username for 'https://gitlab.rz.htw-berlin.de': s0573019
Password for 'https://s0573019@gitlab.rz.htw-berlin.de':
remote: Enumerating objects: 505, done.
remote: Total 505 (delta 0), reused 0 (delta 0), pack-reused 505
Receiving objects: 100% (505/505), 7.09 MiB | 8.84 MiB/s, done.
Resolving deltas: 100% (263/263), done.
p4@p4:~/p4-htw$ cd p4-projekt/
```

Abbildung 3.3: Einrichten des Git-Repository in der VM

Codeauszug 3.1: CLI-Befehle für die Einrichtung von Git in der VM

```
1 mkdir ORDNERNAME
2 cd ORDNERNAME
3 git clone https://gitlab.rz.htw-berlin.de/s0573019/p4-projekt.git
4 USERNAME
5 PASSWORD
6 cd p4-projekt
7 ls -lisa
```

Um die Accountdaten nicht bei jeder Operation mit Git wie bspw. `git pull` (Download der Daten) oder `git push` (Upload der Daten) eingeben zu müssen, empfiehlt es sich diese mittels `git config credential.helper store` zu speichern. Hierbei ist zu beachten, dass diese Daten als Klartext gespeichert werden. Diese Daten werden bei der nächsten Eingabe gespeichert. Um Push-Operationen authentifizieren zu können, sollten auch der eigene Name und die Mail-Adresse mittels folgender Befehle angegeben werden.

```
1 git config --global user.name "VORNAME NACHNAME"
2 git config --global user.email p4istcool@example.yeah
```

Folgende Übungen aus dem Repo sollen in diesem Kapitel genauer ausgeführt werden:

1. Hello_World
2. Simple-Clone
3. Double-Clone
4. Resubmit
5. Resubmit2-IP
6. Resubmit3-ARP

```
p4@p4:~/p4-htw/p4-projekt$ ls -lisa
total 4124
940403 4 drwxrwxr-x 17 p4 p4 4096 Jan 12 13:56 .
940104 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:55 ..
1311067 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:56 100-clones
1311077 4 drwxrwxr-x 2 p4 p4 4096 Jan 12 13:56 Calculator
1311080 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:56 Clone-Draft
1311093 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:56 Double-Clone
940404 4 drwxrwxr-x 8 p4 p4 4096 Jan 12 13:56 .git
940412 4 -rw-rw-r-- 1 p4 p4 54 Jan 12 13:56 .gitignore
1311105 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:56 Hello_World
1311112 4 drwxrwxr-x 2 p4 p4 4096 Jan 12 13:56 Intel_Code
940416 4 -rw-rw-r-- 1 p4 p4 238 Jan 12 13:56 marcel-neu.yaml
940417 4028 -rw-rw-r-- 1 p4 p4 4122906 Jan 12 13:56 p4-plakat-htw-white.pdf
940415 4 -rw-rw-r-- 1 p4 p4 1471 Jan 12 13:56 README.md
940418 4 -rwxrwxr-x 1 p4 p4 1537 Jan 12 13:56 receive-srv.py
1311114 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:56 Resubmit
1311124 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:56 Resubmit2-IP
1311134 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:56 Resubmit3-ARP
1311144 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:56 Simple-Clone
940419 4 -rw-rw-r-- 1 p4 p4 46 Jan 12 13:56 tcpdump-for-ethertype
1311154 4 drwxrwxr-x 3 p4 p4 4096 Jan 12 13:56 TestMarcel
940420 4 -rw-rw-r-- 1 p4 p4 59 Jan 12 13:56 test.txt
940421 4 drwxrwxr-x 6 p4 p4 4096 Jan 12 13:56 tofino-ports
1311161 4 drwxrwxr-x 2 p4 p4 4096 Jan 12 13:56 Topo-Drawings
940426 4 -rw-rw-r-- 1 p4 p4 1550 Jan 12 13:56 vim-p4-syntax-highlight-anleitung.txt
1311177 4 drwxrwxr-x 2 p4 p4 4096 Jan 12 13:56 wedge-files
```

Abbildung 3.4: Ordnerstruktur des Projekt-Repos

3.3 Übungen

Dieser Abschnitt erklärt die sechs Übungen, die in der VM gemacht werden können. Es werden Netzwerktopologien, implementierte Protokolle und Forwarding-Regeln erklärt und Code aus dem jeweiligen P4-Programm gezeigt. Die restliche Konfiguration der VM wird in Abschnitt 3.4 erläutert.

3.3.1 Hello World

Wie in anderen Programmiersprachen gibt es auch für P4 ein „Hello_World“-Programm. Eine Bildschirmausgabe mit diesem Text ist hierbei nicht zu erwarten. Stattdessen ist eine einfache Regel zum Weiterleiten der Pakete implementiert (s. Tabelle 3.2). Dafür wird der ausgehende Port (*Egress Port*) mit dem Befehl `standard_metadata.egress_spec = 2;` gesetzt. Zuvor wird überprüft, auf welchem eingehenden Port (*Ingress Port*) das Paket empfangen wurde (`if (standard_metadata.ingress_port == 1)`). Header und eigene Metadaten werden im Programm nicht verwendet. Das Netzwerk besteht hierbei aus zwei PCs (H_1 und H_2) und einem Switch (S_1), auf dem das P4-Programm läuft. Die Hosts sind dabei über deren Interface Eth0 mit den korrespondierenden Switch-Ports ($H_1 \rightarrow P_1^{11}$; $H_2 \rightarrow P_2$) verbunden (s. Abbildung 3.5).

¹¹= Port 1

Incoming Port	Outgoing Port
1	2
2	1

Tabelle 3.2: Forwarding-Regel für Hello_World

Hello-World Pod-Topology

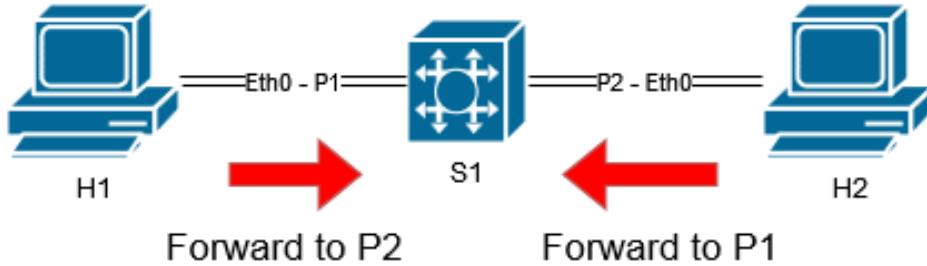


Abbildung 3.5: Netzwerktopologie zu Hello_World

Das Forwarding im *Match-Action-Kontrollblock* (`control MyIngress`) kann wie in Codeauszug 3.2 umgesetzt werden.

Codeauszug 3.2: Forwarding-Code für Hello_World

```

1 control MyIngress(inout headers hdr,
2                     inout metadata meta,
3                     inout standard_metadata_t standard_metadata) {
4
5     apply {
6         if (standard_metadata.ingress_port == 1) {
7             standard_metadata.egress_spec = 2;
8         }
9         else if (standard_metadata.ingress_port == 2) {
10            standard_metadata.egress_spec = 1;
11        }
12    }
13 }
```

Die restlichen, unbenutzten *Kontrollblöcke* müssen deklariert und mit einer leeren `apply`-Anweisung initialisiert werden. Um dies zu veranschaulichen, dient der nachfolgende Quelltextauszug aus dem *Egress-Processing*:

Codeauszug 3.3: Leerer Egress-Kontrollblock

```

1 control MyEgress(inout headers hdr,
```

```

2           inout metadata meta,
3           inout standard_metadata_t standard_metadata) {
4     apply { }
5 }
```

3.3.2 Simple Clone

Das Programm *Simple-Clone* nutzt als erste der Übungen einen Mechanismus zum Vervielfältigen bzw. Klonen von Paketen. Das Netz ist wie in Übung 3.3.1 aufgebaut. Zusätzlich wurde ein dritter Host *H3* analog zu den ersten beiden hinzugefügt. Sobald die Pakete, welche von *H1* gesendet werden, im Switch eintreffen, werden diese 1:1 kopiert und via *P3* gesendet; also an *H3* geklont. Das Originalpaket wird an *H2* gesendet. Es kann sowohl von *H1* als auch *H2* gesendet werden. Empfänger ist dann der nicht-sendende Host.

Simple-Clone Pod-Topology

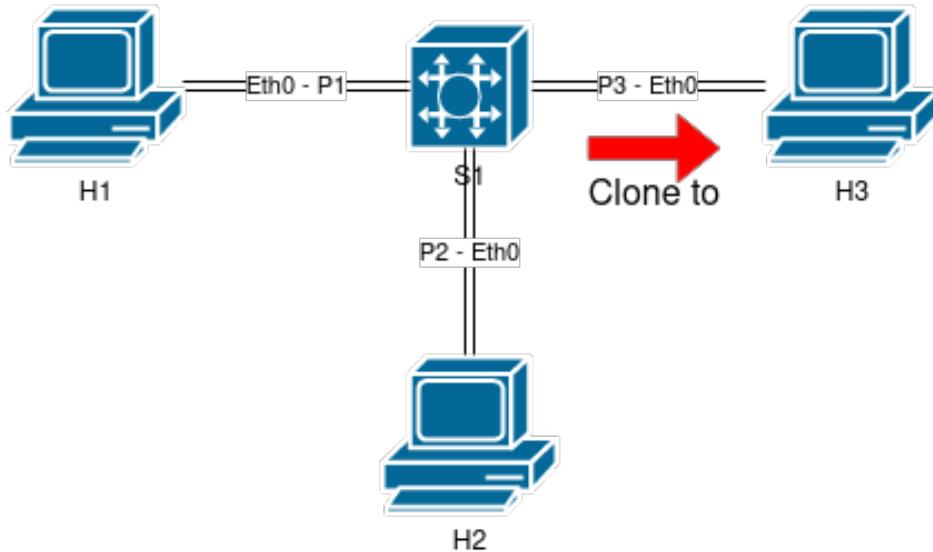


Abbildung 3.6: Netzwerktopologie zu Simple-Clone

Um das P4-Programm für die Vervielfältigung vorzubereiten, müssen 32-bit lange Identifier (ID) nach folgendem Schema angelegt werden:

Codeauszug 3.4: Identifier für Vervielfältigung

```
1 const bit<32> I2E_CLONE_SESSION_ID = 100;
```

Diese ID wird beim später folgendem Funktionsaufruf als Parameter übergeben und muss in Abschnitt 3.4 (Konfiguration Mininet) auf dem Switch ebenfalls angelegt werden. Die

ID gibt dem Switch die nötigen Informationen, wohin die geklonten Pakete zu senden sind. In dieser Übung wird *Clone Ingress to Egress* (CI2E) (vgl. Abschnitt 2.3.7) benutzt. Der Funktionsaufruf findet also im *Ingress-Processing* statt. In *Behavioral Model Version 2* (BMv2) gibt es dafür die Funktion `clone_preserving_field_list(CLONETYPE, SESSION_ID, ownMETADATA_Field)`. Im ersten Parameter wird angegeben, welche Art des Klonens genutzt wird (hier: *CI2E*). Der zweite ist die eben angesprochene *Clone-Session-ID*. Mit dem dritten Parameter ist es möglich eigene Metadaten zu verarbeiten. Die angegebene Zahl signalisiert, welches Feld der eigenen Metadaten genutzt werden soll. Das Forwarding für diese Übung sieht folgendermaßen aus:

Codeauszug 3.5: Forwarding-Code für Simple-Clone

```

1  action do_clone_i2e() {
2      clone_preserving_field_list(CloneType.I2E, I2E_CLONE_SESSION_ID, 0);
3  }
4
5  apply {
6      if (standard_metadata.ingress_port == 1) {
7          standard_metadata.egress_spec = 2;
8          do_clone_i2e();
9      }
10     else if (standard_metadata.ingress_port == 2) {
11         standard_metadata.egress_spec = 1;
12     }
13 }
14 }
```

3.3.3 Double Clone

In dieser Übung werden bestimmte Netzwerkpakete zweimal geklont. Dafür wurde dem Netzwerk aus Abschnitt 3.3.2 ein weiterer P4-Switch hinzugefügt. Dieser klont Pakete von *H1* bzw. *P1* nach *P4*¹². Pakete, die an Port 4 empfangen werden, werden nach *P2* geklont. Header sind nicht definiert. Eigene Metadaten sind angelegt, werden hier aber nicht benutzt. *Clone-Session-IDs* sind wie in Übung 3.3.2 (Simple Clone) deklariert.

Das Forwarding wurde folgendermaßen implementiert:

Codeauszug 3.6: Forwarding-Code für Double-Clone

```

1  action do_clone_i2e() {
2      clone_preserving_field_list(CloneType.I2E, I2E_CLONE_SESSION_ID, 0);
3  }
4  apply {
5      if (standard_metadata.ingress_port == 1) {
6          standard_metadata.egress_spec = 4;
```

¹²hier: Port 4

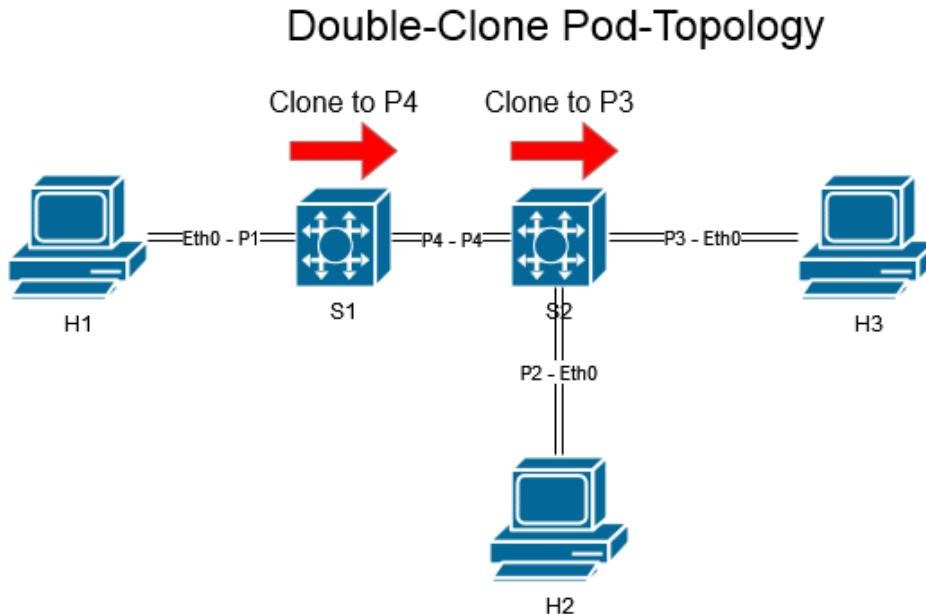


Abbildung 3.7: Netzwerktopologie zu Double-Clone

```

7     do_clone_i2e();
8 }
9 else if (standard_metadata.ingress_port == 4) {
10     standard_metadata.egress_spec = 2;
11     do_clone_i2e(); }
12 }
13 }
```

3.3.4 Resubmit

Die nächsten drei Übungen nutzen *Resubmit* als Weg der Vervielfältigung. Das Netz ist dabei wie in Übung 3.3.1 aufgebaut.

Header werden nicht extrahiert. Das Paket wird unverändert weitergeleitet. Für das Forwarding ist es wichtig, vervielfältigte Pakete von normalen zu unterscheiden. Dies wird mittels folgender Konstanten erreicht:

Codeauszug 3.7: Typen der Vervielfältigung

```

1 const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_NORMAL = 0;
2 const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_INGRESS_CLONE = 1;
3 const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_EGRESS_CLONE = 2;
4 const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_COALESCED = 3;
5 const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_RECIRC = 4;
6 const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_REPLICATION = 5;
7 const bit<32> BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT = 6;
```

Resubmit Pod-Topology

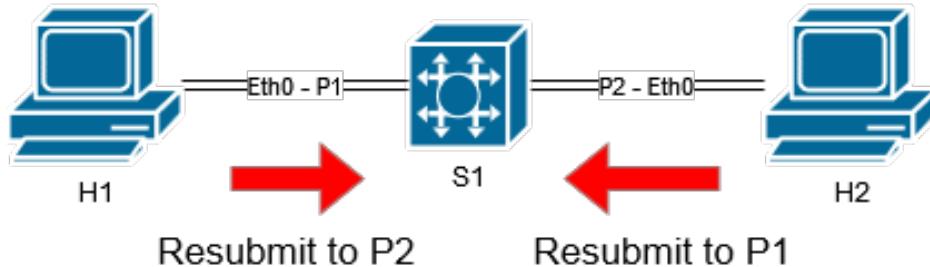


Abbildung 3.8: Netzwerktopologie zu Resubmit

Das Forwarding nutzt erstmals auch die *eigenen Metadaten*. In diesen wird der *Ingress-Port* gespeichert.

```

1 struct mymeta_t {
2     @field_list(1) // this number must be called in ←
3         resubmit_preserving_field_list()
4     bit<9> port;
5 }
```

Das *Ingress-Processing* ist bei dieser Übung deutlich ausgeprägter als in den vorherigen Übungen. Zu Beginn wird anhand der oben deklarierten Vervielfältigungstypen überprüft, ob ein Paket bereits vervielfältigt wurde (`if (standard_metadata.instance_type == BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT)`). Ist dies der Fall, so wird anhand des *Ingress-Ports* in den Metadaten (`if (meta.mymeta.port == 1)`) der *Egress-Port* gesetzt. Falls von den ersten beiden Bedingungen keine zutrifft, wird anhand des *Ingress-Ports* (`standard_metadata.ingress_port == 0`) des Resubmit-Pakets der *Egress-Port* auf *P2* gesetzt. Wenn das Originalpaket (`else if (standard_metadata.instance_type == BMV2_V1MODEL_INSTANCE_TYPE_NORMAL)`) verarbeitet wird, wird zunächst der *Ingress-Port* im *Port-Feld* der Metadaten gespeichert (`meta.mymeta.port = standard_metadata.ingress_port;`). Danach wird der *Resubmit*-Befehl gesetzt (`(resubmit_preserving_field_list(1);`). Die 1 signalisiert, in welchem Feld der eigenen Metadaten gespeichert werden soll, solange das Paket noch nicht gesendet wurde. Anschließend wird anhand des *Ingress-Ports* der *Egress-Port* wie in Übung 3.3.1 gesetzt.

Codeauszug 3.8: Forwarding-Code für Resubmit

```

1 apply {
2     // Check whether packet was resubmitted
3     if (standard_metadata.instance_type == ←
4         BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT){
5         // packet is already resubmitted --> change outgoing port :)
```

```

6     if (meta.mymeta.port == 1) {
7         standard_metadata.egress_spec = 2;
8     }
9     else if (meta.mymeta.port == 2) {
10        standard_metadata.egress_spec = 1;
11    }
12 //all Resubmitted Packets: ingress_port = 0
13 else if (standard_metadata.ingress_port == 0) {
14     standard_metadata.egress_spec = 2;
15 }
16 }
17 else if (standard_metadata.instance_type == ←
18     BMV2_V1MODEL_INSTANCE_TYPE_NORMAL){
19 // standard, incoming packet --> resubmit
20 // --> speichert den Ingress Port in "MYMETA.PORT" --> spaeter wichtig :)
21 meta.mymeta.port = standard_metadata.ingress_port;
22 resubmit_preserving_field_list(1);
23 // set outgoing port -> does nothing because "original" packet isn't ←
24 // transmitted, only resubmitted packet
25 if (standard_metadata.ingress_port == 1) {
26     standard_metadata.egress_spec = 2;
27 }
28 else if (standard_metadata.ingress_port == 2) {
29     standard_metadata.egress_spec = 1;
30 }
31 }
```

3.3.5 Resubmit2-IP

Das Netzwerk von Übung „Resubmit2-IP“ ist wie in Übung 3.3.2 aufgebaut. Da das Forwarding mittels einer *Tabelle* erfolgt, sind Header für Ethernet und IPv4 im Programm angelegt. Diese Header werden im *Parser* extrahiert, um im Programm damit arbeiten zu können. Der IPv4-Header ist wie in Abschnitt 2.4 angelegt. Der Typ `macAddr_t` wurde mittels `typedef` deklariert. Der Ethernet-Header wurde folgendermaßen implementiert:

Codeauszug 3.9: Ethernet-Header in P4

```

1 header ethernet_t {
2     macAddr_t dstAddr;
3     macAddr_t srcAddr;
4     bit<16> etherType;
5 }
```

Diese beiden Header müssen im `struct headers` eingetragen werden:

Resubmit2-IP Pod-Topology

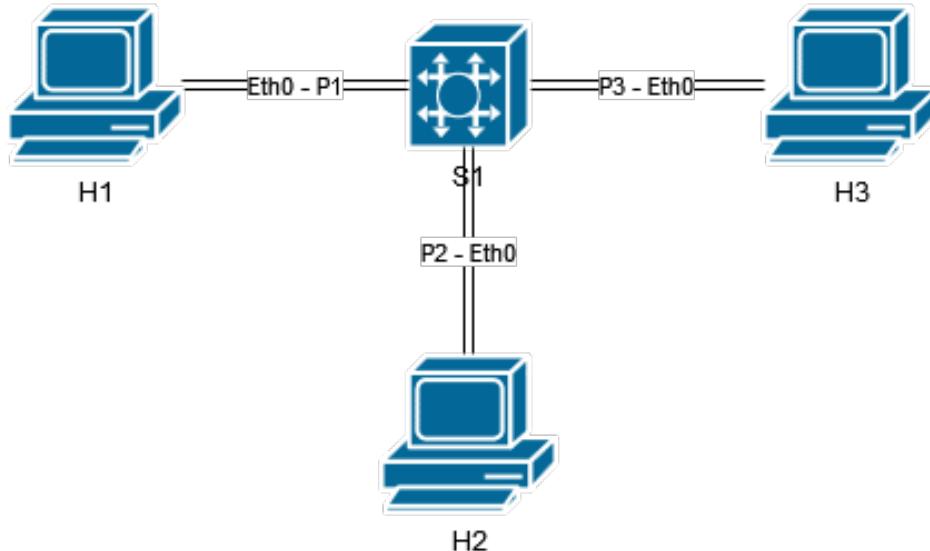


Abbildung 3.9: Netzwerktopologie zu Resubmit2-IP

Codeauszug 3.10: Header-Struct

```

1 struct headers {
2     ethernet_t ethernet;
3     ipv4_t ipv4;
4 }
```

Der *Parser* muss nun angewiesen werden diese beiden Header auszulesen und anhand der darin enthaltenen Daten entscheiden, welcher `state` als nächstes bearbeitet wird. In dieser Übung wird diese Entscheidung nur anhand des *Ethertypes* im Ethernet-Header getroffen. Der *Ethertype* dient dazu das Protokoll auf Layer3 zu identifizieren (ähnlich wie das *Protocol-Field* im IPv4-Header, vgl. Abschnitt 2.4 Möglichkeiten von P4). Hierfür wurden zwei IDs des *Ethertypes* nach [EtherType-Numbers] angelegt:

Codeauszug 3.11: Etheratypes in P4

```

1 const bit<16> TYPE_IPV4 = 0x800;
2 const bit<16> TYPE_ARP = 0x806;
```

Der *Parser* wird angewiesen direkt von `state start` zu `state parse_ethernet` zu gehen. Dort wird der Ethernet-Header extrahiert und anhand des *Ethertypes* entschieden (`transition select(hdr.ethernet.etherType)`), welcher `state` als nächstes durchgeführt wird.

Codeauszug 3.12: State-Machine des Parsers für Resubmit2-IP

```

1 state start {
2     transition parse_ethernet;
3 }
4
5 state parse_ethernet {
6     packet.extract(hdr.ethernet);
7     transition select(hdr.ethernet.etherType){
8         TYPE_IPV4: parse_ipv4;
9         TYPE_ARP: arp_fwd;
10        default: accept;
11    }
12 }
13
14 state parse_ipv4 {
15     packet.extract(hdr.ipv4);
16     transition accept;
17 }
18
19 state arp_fwd {
20     transition accept;
21 }

```

Es wurden zwei *Actions* definiert. Die `action drop` markiert das Feld in den `standard_metadata`, welches dem *Traffic-Manager* (TM) signalisiert, dass das Paket verworfen und nicht gesendet wird. Die `action ipv4_forward` setzt den Outgoing-Port und die Ziel-Media-Access-Control-(MAC)-Adresse anhand der Daten aus der `table ipv4_lpm`. Außerdem wird die *Time to Live* (TTL) dekrementiert und die Source-MAC-Adresse neu gesetzt. Die `table ipv4_lpm` nutzt als `key` die Ziel-IP-Adresse und das *Match-Kriterium* `lpm` (vgl. Abschnitt 2.3.4). Wird beim Durchsuchen der *Tabelle* ein dem `Key` entsprechender Eintrag gefunden, so kann eine der zwei zuvor erklärten *Actions* oder `NoAction` ausgeführt werden. `NoAction` ist hierbei die *Default-Action*. Diese *Tabelle* kann bis zu 1024 Einträge haben (`size = 1024`). Die Einträge müssen über die *Control-Plane* in die *Tabelle* eingefügt werden. Dies wird im Abschnitt 3.4 thematisiert. In diesem Beispiel wird die Interaktion von *Control- und Data-Plane* nochmal deutlich. Die zuvor aus der *Control-Plane* in eine *Tabelle* eingefügten Einträge, werden von der *Data-Plane* genutzt, um für jedes Paket einzeln zu entscheiden, ob und auf welchem Port dieses Paket gesendet wird.

Der `apply`-Block ist wie in Übung 3.3.4 konfiguriert. Das statische Forwarding anhand des *Ingress-Ports* wurde hierbei durch das Anwenden der *Tabelle* ersetzt (`ipv4_lpm.apply();`). Anhand der Bedingung `if(hdr.ipv4.isValid())` kann überprüft werden, ob ein IPv4-Header beim aktuellen Paket vorhanden ist. Für ein einfacheres Testen wurde diese Bedingung auskommentiert. Sollte diese Zeile wieder genutzt werden wollen, muss auch die nächste *else-Bedingung* wieder genutzt werden. Ansonsten wird für alle Pakete, die keinen IPv4-Header haben (z.B. *Address Resolution Protocol* (ARP)), kein *Outgoing-Port* gesetzt

und die Pakete nicht weitergeleitet.

Codeauszug 3.13: Ingress-Processing in Resubmit2_IP

```

1  action drop() {
2      mark_to_drop(standard_metadata);
3  }
4
5  action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
6      standard_metadata.egress_spec = port;
7      hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
8      hdr.ethernet.dstAddr = dstAddr;
9      hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
10 }
11
12 // Table for Eth/IP routing
13 table ipv4_lpm {
14     key = {
15         hdr.ipv4.dstAddr: lpm;
16     }
17     actions = {
18         ipv4_forward;
19         drop;
20         NoAction;
21     }
22     size = 1024;
23     default_action = NoAction();
24 }
25
26 apply {
27     // Check whether packet was resubmitted
28     if (standard_metadata.instance_type == ↵
29         BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT){
30         // packet is already resubmitted --> change outgoing port :)
31         // if(hdr.ipv4.isValid()){
32             ipv4_lpm.apply();
33         // }
34         // else{
35             // standard_metadata.egress_spec = 2;
36         // }
37     }
38     else if (standard_metadata.instance_type == ↵
39         BMV2_V1MODEL_INSTANCE_TYPE_NORMAL){
40         // standard, incoming packet --> resubmit
41         meta.mymeta.port = standard_metadata.ingress_port;
42         resubmit_preserving_field_list(1);
43     }

```

43 }

Da im *Ingress-Processing* die Daten im IPv4-Header verändert werden, muss auch die *Checksum* neu berechnet werden. Im *Checksum-Computation-Kontrollblock* müssen alle Felder des IPv4-Header dazu genutzt werden:

Codeauszug 3.14: Checksum Computation in Resubmit2_IP

```

1 control MyComputeChecksum(inout headers hdr, inout metadata meta) {
2     apply {
3         update_checksum(
4             hdr.ipv4.isValid(),
5             { hdr.ipv4.version,
6                 hdr.ipv4.ihl,
7                 hdr.ipv4.diffserv,
8                 hdr.ipv4.totalLen,
9                 hdr.ipv4.identification,
10                hdr.ipv4.flags,
11                hdr.ipv4.fragOffset,
12                hdr.ipv4.ttl,
13                hdr.ipv4.protocol,
14                hdr.ipv4.srcAddr,
15                hdr.ipv4.dstAddr },
16                hdr.ipv4.hdrChecksum,
17                HashAlgorithm.csum16);
18    }
19 }
```

Da sowohl IPv4- als auch Ethernet-Header (MAC-Adressen) verändert wurden, müssen diese beiden Header im *Deparser* explizit gesendet werden (`packet.emit(hdr.X13)`):

Codeauszug 3.15: Deparser in Resubmit2_IP

```

1 control MyDeparser(packet_out packet, in headers hdr) {
2     apply {
3         packet.emit(hdr.ethernet);
4         packet.emit(hdr.ipv4);
5     }
6 }
```

3.3.6 Resubmit3-ARP

Das Netzwerk dieser Übung ist wie in Übung 3.3.5 Resubmit2-IP aufgebaut (s. Abbildung 3.10). Will ein Host Pakete an einen anderen senden, muss der Sender die MAC-Adresse des Ziels wissen. Um diese herauszufinden, gibt es das *ARP*-Protokoll. Es ordnet einer

¹³X = Platzhalter für den gewünschten Header

Ziel-IP-Adresse eine Ziel-MAC-Adresse zu. Dafür wird vom Sender ein Broadcast¹⁴-Paket gesendet. Es wird mittels Broadcast-MAC-Adresse (ff:ff:ff:ff:ff:ff) an die Ziel-IP-Adresse gesendet. Sollte ein Host diese IP-Adresse haben, antwortet er an den ursprünglichen Sender mittels der eigenen MAC-Adresse. Diese Daten werden in der *ARP-Tabelle* gespeichert. [s. [RFC826]]

Dieser Mechanismus wurde auch in dieser Übung implementiert, um das Netz dynamischer zu machen und weniger manuelle Konfiguration nötig ist. Außerdem bedient sich das *Internet Control Message Protocol* (ICMP) nach [RFC792] dieser Technik. *Header, Clone-Session-IDs, Clone-Types, Ethertypes, IPv4 Protocol Numbers, eigene Metadaten, Checksum Computation, Deparser, Tabellen und Actions* wurden aus Übung Resubmit2-IP übernommen. Um Broadcasts weiterleiten zu können, muss im *Ingress-Processing* eine

Resubmit3-ARP Pod-Topology

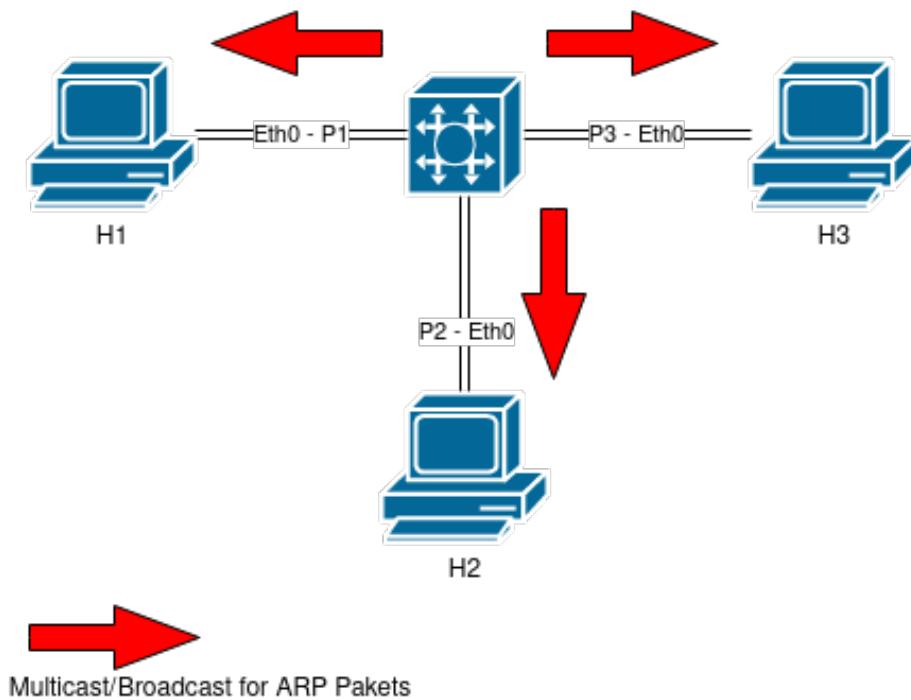


Abbildung 3.10: Netzwerktopologie zu Resubmit3-ARP

Action deklariert werden. Hierfür wird der Mechanismus *Multicast* verwendet, da es in P4 keinen Broadcast-Mechanismus gibt. Multicast ermöglicht die Kommunikation zwischen einem Sender und mehreren Empfängern [s. [RFC1112]].

Hierfür muss Multicast eine Gruppe zugewiesen werden (`standard_metadata.mcast_grp = 1;`) (hier: 1) und die TTL verändert werden. Diese Gruppe signalisiert beim Senden, an welche Ports das ursprüngliche Paket gesendet werden soll (s. Abschnitt 3.4 Konfiguration Mininet).

¹⁴Senden an das gesamte Subnetz

Codeauszug 3.16: Actions für Multicast/Broadcast/ARP

```

1  action broadcast() {
2      //basically Multicast to Group 1
3      standard_metadata.mcast_grp = 1;
4      //meta.ingress_metadata_nhop_ipv4 = hdr.ipv4.dstAddr;
5      hdr.ipv4.ttl = hdr.ipv4.ttl + 8w255;
6  }
7
8  action arp_forward(){
9      broadcast();
10 }

```

Im *Ingress-Processing* wird anhand des *Ethertypes* überprüft, ob ein ARP- oder IPv4-Paket gerade verarbeitet wird. Sollte es ein ARP-Paket sein (`if (hdr.ethernet.etherType == TYPE_ARP)`), wird dieses mittels `action arp_forward()` als Broadcast gesendet. Wenn es ein IPv4-Paket ist (`else if (hdr.ethernet.etherType == TYPE_IPV4)`), wird dieses wie in Übung Resubmit2-IP behandelt. Die genaue Zuweisung von Literalen an Variablen wie im vorigen Codeauszug in Zeile 5 (+ 8w255) kann in [P4 Lang. Spec. Abschnitt 6.3.3.2] nachgelesen werden.

Codeauszug 3.17: Ingress-Processing in Resubmit3_ARP

```

1 apply {
2     if (hdr.ethernet.etherType == TYPE_ARP){
3         arp_forward();
4     }
5     else if (hdr.ethernet.etherType == TYPE_IPV4){
6         // Check whether packet was resubmitted
7         if (standard_metadata.instance_type == ←
8             BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT){
9             // packet is already resubmitted --> change outgoing port :)
10            // if(hdr.ipv4.isValid()){
11                ipv4_lpm.apply();
12            //}
13        }
14        else if (standard_metadata.instance_type == ←
15            BMV2_V1MODEL_INSTANCE_TYPE_NORMAL){
16            // standard, incoming packet --> resubmit
17            // speichert den Ingress Port in meta.mymeta.port -> spaeter wichtig :)
18            meta.mymeta.port = standard_metadata.ingress_port;
19            resubmit_preserving_field_list(1);
20        }

```

3.3.7 weitere Übungen

Es wurden noch weitere Übungen/Programme erstellt, welche hier kurz erläutert werden.

Name	Beschreibung
100-clones	von einem eingehenden Paket werden 100 Klonen angefertigt und gesendet
Calculator	Beispielprogramm aus einer Intel-Schulung: der Switch wird zu einem Taschenrechner umfunktioniert und sendet das Ergebnis der Berechnung an den ursprünglichen Host zurück
Clone-Draft	Implementierung des Clone-Headers aus Abschnitt 2.4

Tabelle 3.3: Weitere Übungen aus dem Git-Repo

3.4 Konfiguration Mininet

Auf der VM ist *Mininet* installiert. Mininet kann ein virtuelles Netzwerk samt Netzwerkgeräten und Hosts initialisieren. Switches inkl. Control-Plane und Einträgen in Tabellen sowie Host-PCs werden hier konfiguriert. In jedem Ordner der Übungen gibt es den Ordner `pod-topo`. In diesem befinden sich die drei benötigten Files für die Konfiguration:

Dateiname	Nutzen
s1-commands.txt	Konfiguration Mirror-Session Konfiguration Multicast
s1-runtime.json	Konfiguration Tabellen inkl. Match-Action Anlegen Tabelleneinträge
topology.json	Erstellung Hosts inkl. IP- und MAC-Adressen Erstellung Switch virtuelle Verkabelung der Geräte

Tabelle 3.4: Files für Mininet Konfiguration und deren Nutzen

Die Dateitypen und Namen dürfen hierbei nicht geändert werden. Sofern mehrere Switches wie in Übung 3.3.3 (Double Clone) genutzt werden, gibt es auch für diese die spezifischen Command- und Runtime-Dateien (vgl. Codeauszug in Abschnitt 3.4.1, Zeile 14f).

3.4.1 Konfiguration der Topologie

Die Konfiguration des Netzes wird anhand des Netzes aus der Übung Double Clone erklärt, da in dieser zwei Switches genutzt werden. So kann auch die Interconnection von zwei Netzwerkgeräten erklärt werden. Im ersten Abschnitt (Z. 2-8) des folgenden Codes werden die `hosts` mit Namen, IP- und MAC-Adresse konfiguriert. Im zweiten Teil (Z. 10-13) werden die `switches` mit Namen und Input-Files initialisiert (vgl. Tabelle 3.4). Am Ende

(Z. 18ff) werden Switche und Hosts miteinander verbunden. Für Hosts wird deren Name (z.B. h3) und der zugehörige Switchport (z.B. s2-p3 für Switch 2 Port 3) angegeben; bei Switch-Interconnections jeweils die Portbeschreibung (["s1-p4", "s2-p4"]).

Codeauszug 3.18: topology.json aus Double-Clone

```

1  {
2      "hosts": {
3          "h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
4              "commands": []},
5          "h2": {"ip": "10.0.1.2/24", "mac": "08:00:00:00:02:22",
6              "commands": []},
7          "h3": {"ip": "10.0.1.3/24", "mac": "08:00:00:00:03:33",
8              "commands": []}
9      },
10     "switches": {
11         "s1": { "runtime_json" : "pod-topo/s1-runtime.json",
12             "cli_input": "pod-topo/s1-commands.txt"
13         },
14         "s2": { "runtime_json" : "pod-topo/s2-runtime.json",
15             "cli_input": "pod-topo/s2-commands.txt"
16         }
17     },
18     "links": [
19         ["h1", "s1-p1"], ["h2", "s2-p2"], ["h3", "s2-p3"], ["s1-p4", "s2-p4"]
20     ]
21 }
```

3.4.2 Konfiguration Switches

Für jeden Switch müssen Command- und Runtime-Dateien erstellt werden (vgl. Tabelle 3.4). Diese Dateien werden in den beiden nächsten Unterabschnitten erklärt.

Mirror und Multicast Einstellungen

Die in den Übungen 3.3.2 und 3.3.3 genutzte *Clone-Session-IDs* müssen auf dem Switch in der Datei **s1-commands.txt** angelegt werden. Mittels **mirroring_add 100 3** wird das *Target* angewiesen, alle Klone mit **Clone-Session-ID = 100** auf Port3 zu senden. Via **mc_mgrp_create 1** wird die Multicast-Gruppe 1 erstellt (vgl. Resubmit3-ARP).

mc_node_create X Y weiß dabei einer *Replication-ID* (rid) (**intrinsic_metadata.egress_rid**) X einen *Egress-Port* Y zu. Die *Replication-ID* kann als zusätzlicher *Key* für *Tabellen* und *Actions* genutzt werden. So kann z.B. ein Paket zweimal über denselben Port gesendet werden, aber verschiedene Destination-IP-Adressen besitzen [s. [MC-Replication-ID]]. Da dieser Mechanismus in den Übungen nicht genutzt wird und

somit egal ist, was dort eingetragen wird, wurde `rid = 0` gesetzt. `mc_node_associate X Y` weiß dabei Node Y der Multicast-Gruppe X zu. Hierbei ist zu beachten, dass dem ersten Host `H1` aus Code 3.4.1 der Node-Index 0 (Null) zugewiesen wurde. Analog dazu die restlichen Hosts.

Codeauszug 3.19: s1-commands.txt aus Resubmit3_ARP

```
1 mirroring_add 100 3
2
3 mc_mgrp_create 1
4 mc_node_create 0 1
5 mc_node_create 0 2
6 mc_node_create 0 3
7
8 mc_node_associate 1 0
9 mc_node_associate 1 1
10 mc_node_associate 1 2
```

Erstellung Tabellen und Hinzufügen von Einträgen

Tabellen, welche im P4-Programm angelegt wurden, müssen in der Control-Plane des Targets konfiguriert werden. Im ersten Abschnitt des Runtime-Files (`s1-runtime.json`) wird `bmv2` als Target-Architektur und die Info- sowie JSON-Konfigurations-Datei angegeben. Daraufhin wird die Tabelle `ipv4_1pm` aus dem Kontrollblock `MyIngress` samt Default-Action inkl. Parametern angegeben. Danach werden dieser Tabelle drei Einträge hinzugefügt. Hierbei wird der Key aus dem Programm als „match“ bezeichnet. Anschließend folgt die auszuführende Action („action_name“) inkl. Parameter und Übergabewerten. Sollte für den ersten Eintrag (Zeile 12-22) der Key zutreffen, so wird die Action `MyIngress.ipv4_forward` ausgeführt. Die MAC-Adresse `08:00:00:00:01:11` und der Egress-Port `1` ("port": 1) werden dann in der Action im Programm gesetzt.

Codeauszug 3.20: s1-runtime.json aus Resubmit3_ARP

```
1 {
2   "target": "bmv2",
3   "p4info": "build/resubmit3-ip.p4.p4info.txt",
4   "bmv2_json": "build/resubmit3-ip.json",
5   "table_entries": [
6     {
7       "table": "MyIngress.ipv4_1pm",
8       "default_action": true,
9       "action_name": "MyIngress.drop",
10      "action_params": {}
11    },
12    {
13      "table": "MyIngress.ipv4_1pm",
```

```

14     "match": {
15         "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
16     },
17     "action_name": "MyIngress.ipv4_forward",
18     "action_params": {
19         "dstAddr": "08:00:00:00:01:11",
20         "port": 1
21     }
22 },
23 {
24     "table": "MyIngress.ipv4_lpm",
25     "match": {
26         "hdr.ipv4.dstAddr": ["10.0.1.2", 32]
27     },
28     "action_name": "MyIngress.ipv4_forward",
29     "action_params": {
30         "dstAddr": "08:00:00:00:02:22", "port": 2
31     }
32 },
33 {
34     "table": "MyIngress.ipv4_lpm",
35     "match": {
36         "hdr.ipv4.dstAddr": ["10.0.1.3", 32]
37     },
38     "action_name": "MyIngress.ipv4_forward",
39     "action_params": {
40         "dstAddr": "08:00:00:00:03:33",
41         "port": 3
42     }
43 },
44 ]
45 }

```

Der Import von Tabelleneinträgen mit der Datei `s1-runtime.json` hat anfangs nicht funktioniert. Deshalb werden hier die nötigen Befehle aufgelistet, die für eine manuelle Konfiguration genutzt werden. Dafür loggt man sich auf dem Switch (S1) ein (`simple_switch_CLI -thrift-ip 10.0.2.15`). Dieser Befehl muss aus dem Projektordner und nicht in Mininet eingegeben werden. In der CLI des Switches können Einträge nach folgendem Beispiel hinzugefügt werden:

Codeauszug 3.21: Anlegen von Tabelleneinträgen auf einem Switch in Mininet

```

1 // Allgemeiner Befehl
2 table_add KONTROLLBLOCKNAME.TABELLENNAME KONTROLLBLOCKNAME.ACTIONNAME KEY ↵
    => PARAMETER
3
4 // Hinzufuegen von Eintrag mit Key="10.0.1.2" in Tabelle "ipv4_lpm" in ↵

```

```
1 Kontrollblock "MyIngress" mit Action "ipv4_forward" mit Parametern ←  
2 "08:00:00:00:02:22" und "2" (=Port)  
3 table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.1.2 => ←  
4 08:00:00:00:02:22 2
```

Der automatisierte Import dieser Einträge funktionierte zum Ende dieser Arbeit.

3.4.3 Anpassen restlicher Dateien

Als letzter Schritt der Konfiguration müssen noch die *Makefiles* angepasst werden. Diese werden für das Kompilieren der Programme und Topologien benötigt. In jedem Ordner einer Übung gibt es ein übungsspezifisches Makefile. Außerdem gibt es noch ein allgemeines Makefile, das nicht im Git-Repo, sondern in `~/tutorials/utils/`¹⁵ gespeichert ist.

Übungsspezifisches Makefile

Die übungsspezifischen Makefiles geben als erstes an, welches Switch-Modell benutzt werden soll. Diese Zeile sollte nicht geändert werden. Danach wird die gewünschte Topologie (vgl. Abschnitt 3.4.1) referenziert. Das übergeordnete Makefile wird als letztes eingebunden. Für diese beiden Dateien ist es wichtig den richtigen Ordnerpfad und Dateinamen anzugeben. Je nachdem in welchem Ordner sich das Git-Repo befindet, muss dieser Pfad angepasst werden, falls es sich um eine relative Pfadangabe handelt. Um dies zu vermeiden, wurde der absolute Pfad angegeben. Dieser funktioniert unabhängig vom Installationsort des Git-Repos.

Codeauszug 3.22: Übungsspezifisches Makefile

```
1 BMV2_SWITCH_EXE = simple_switch_grpc  
2 TOPO = pod-topo/topology.json  
3  
4 include ~/tutorials/utils/Makefile
```

Übergeordnetes Makefile

Hier ist es wichtig zu überprüfen, dass folgender Pfad- und Dateiname richtig angegeben wird:

`RUN_SCRIPT = ~/tutorials/utils/run_exercise.py`

Alle Pfade bzw. Dateien können absolut oder relativ angegeben werden.

¹⁵Ordner vom P4-Language-Consortium

3.5 Benutzung der Übungen

3.5.1 Start der Übung

Das P4-Programm sowie Mininet können via `make` gestartet werden. Dafür muss man im Terminal per `cd ORDNERPFAD` in den Ordner der Übung wechseln. Sollte in einem der beiden Makefiles Fehler sein, werden diese im Terminal angezeigt. Bei erfolgreichem Kompilieren der Files, werden die konfigurierten Hosts inkl. Port, IP- und MAC-Adressen und Files, welche für die Konfiguration des/der Target/s genutzt wurden, ausgegeben. Außerdem wird hier angegeben, wie man die Konfiguration des laufenden Targets ändern, dessen Logs betrachten und Netzwerkverkehr mitschneiden kann.

```
p4@p4:~/p4-htw/p4-projekt/Resubmit3-ARP$ make
mkdir -p build pcaps logs
p4c-bm2-ss --p4v 16 --p4runtime-files build/resubmit3-arp.p4.p4info.txt -o build/resubmit3-arp.json resubmit3-arp.p4
sudo python3 ~/tutorials/utils/run_exercise.py -t pod-topo/topology.json -j build/resubmit3-arp.json -b simple_switch_grpc
Reading topology file.
Building mininet topology.
Configuring switch s1 with file pod-topo/s1-commands.txt
Configuring switch s1 using P4Runtime with file pod-topo/s1-runtime.json
    ERROR! While parsing input runtime configuration: file does not exist /home/p4/p4-htw/p4-projekt/Resubmit3-ARP/build/resubmit3-ip.p4.p4info.txt
s1 -> gRPC port: 50051
*****
h1
default interface: eth0 10.0.1.1      08:00:00:00:01:11
*****
h2
default interface: eth0 10.0.1.2      08:00:00:00:02:22
*****
h3
default interface: eth0 10.0.1.3      08:00:00:00:03:33
*****
Starting mininet CLI
=====
Welcome to the BMV2 Mininet CLI!
=====
Your P4 program is installed into the BMV2 software switch
and your initial runtime configuration is loaded. You can interact
with the network using the mininet CLI below.

To inspect or change the switch configuration, connect to
its CLI from your host operating system using this command:
  simple_switch_CLI --thrift-port <switch thrift port>

To view a switch log, run this command from your host OS:
  tail -f /home/p4/p4-htw/p4-projekt/Resubmit3-ARP/logs/<switchname>.log

To view the switch output pcap, check the pcap files in /home/p4/p4-htw/p4-projekt/Resubmit3-ARP/pcaps:
  for example run: sudo tcpdump -XXX -r s1-eth1.pcap

To view the P4Runtime requests sent to the switch, check the
corresponding txt file in /home/p4/p4-htw/p4-projekt/Resubmit3-ARP/logs:
  for example run: cat /home/p4/p4-htw/p4-projekt/Resubmit3-ARP/logs/s1-p4runtime-requests.txt
mininet>
```

Abbildung 3.11: Starten der Übung

Nützliche *Command Line Interface* (CLI)-Befehle in Mininet sind in Tabelle 3.5 zu sehen.

Befehl	Beschreibung
net	zeigt Geräte an und welches Interface womit verbunden sind
nodes	zeigt alle Geräte an
links	zeigt Verbindungen zwischen Geräten und deren Status an
exit	Beenden von Mininet

Tabelle 3.5: Nützliche Befehle in Mininet

Der Output dieser Befehle sieht für Übung Double Clone folgendermaßen aus:

```

mininet> net
h1 eth0:s1-eth1
h2 eth0:s2-eth2
h3 eth0:s2-eth3
s1 lo: s1-eth1:eth0 s1-eth4:s2-eth4
s2 lo: s2-eth2:eth0 s2-eth3:eth0 s2-eth4:s1-eth4
mininet> nodes
available nodes are:
h1 h2 h3 s1 s2
mininet> links
E0119 14:41:24.020191297 2300 fork_posix.cc:70] Fork support is only compatible with the epoll1 and poll polling strategies
E0119 14:41:24.029566969 2300 fork_posix.cc:70] Fork support is only compatible with the epoll1 and poll polling strategies
eth0<->s1-eth1 (OK OK)
E0119 14:41:24.039231950 2300 fork_posix.cc:70] Fork support is only compatible with the epoll1 and poll polling strategies
E0119 14:41:24.048147779 2300 fork_posix.cc:70] Fork support is only compatible with the epoll1 and poll polling strategies
eth0<->s2-eth2 (OK OK)
E0119 14:41:24.057805943 2300 fork_posix.cc:70] Fork support is only compatible with the epoll1 and poll polling strategies
E0119 14:41:24.069741363 2300 fork_posix.cc:70] Fork support is only compatible with the epoll1 and poll polling strategies
eth0<->s2-eth3 (OK OK)
E0119 14:41:24.075179274 2300 fork_posix.cc:70] Fork support is only compatible with the epoll1 and poll polling strategies
E0119 14:41:24.079421087 2300 fork_posix.cc:70] Fork support is only compatible with the epoll1 and poll polling strategies
s1-eth4<->s2-eth4 (OK OK)

```

Abbildung 3.12: Bildschirmausgabe der nützlichen Mininet-Befehle

3.5.2 Log-Files

Nach Starten der Übung wird im jeweiligen Übungsverzeichnis auch der Ordner `logs` angelegt. Dort werden für jeden Switch drei Files erstellt. Das wichtigste ist hierbei `SWITCHNAME.log`. In diesem wird die Verarbeitung von jedem eingehenden Paket inkl. aller *Kontrollblöcke* und den darin enthaltenen *Actions*, *Bedingungen* und anderen Abläufen festgehalten. Sollte ein Programm nicht wie gewünscht funktionieren, ist diese Datei ein guter Anhaltspunkt, um mögliche Fehlerquellen zu erschließen. Ein kurzer Ausschnitt des Logs vom ersten Switch (S1) aus Übung 3.3.3 ist in Abbildung 3.13 zu sehen.

```

[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [ctx 0] Table 'tbl_do_clone_i2e': miss
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [ctx 0] Action entry is MyIngress.do_clone_i2e -
[10:10:35.063] [bmv2] [T] [thread 1534] [34.0] [ctx 0] Action MyIngress.do_clone_i2e
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [ctx 0] double-clone.p4(58) Primitive clone_preserving_field_list(CloneType.I2E, I2E_CLONE_SESSION_ID, 0)
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [ctx 0] Pipeline 'ingress': end
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [ctx 0] Cloning packet at ingress
[10:10:35.063] [bmv2] [D] [thread 1534] [34.1] [ctx 0] Parser 'parser': start
[10:10:35.063] [bmv2] [D] [thread 1534] [34.1] [ctx 0] Parser 'parser' entering state 'start'
[10:10:35.063] [bmv2] [D] [thread 1534] [34.1] [ctx 0] Parser state 'start' has no switch, going to default next state
[10:10:35.063] [bmv2] [T] [thread 1534] [34.1] [ctx 0] Bytes parsed: 0
[10:10:35.063] [bmv2] [D] [thread 1534] [34.1] [ctx 0] Parser 'parser': end
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [ctx 0] Cloning packet to egress port 4
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [ctx 0] Egress port is 4
[10:10:35.063] [bmv2] [D] [thread 1535] [34.1] [ctx 0] Pipeline 'egress': start
[10:10:35.063] [bmv2] [D] [thread 1535] [34.1] [ctx 0] Pipeline 'egress': end
[10:10:35.063] [bmv2] [D] [thread 1535] [34.1] [ctx 0] Deparser 'deparser': start
[10:10:35.063] [bmv2] [D] [thread 1535] [34.1] [ctx 0] Deparser 'deparser': end
[10:10:35.063] [bmv2] [D] [thread 1535] [34.0] [ctx 0] Pipeline 'egress': start
[10:10:35.063] [bmv2] [D] [thread 1535] [34.0] [ctx 0] Pipeline 'egress': end
[10:10:35.063] [bmv2] [D] [thread 1535] [34.0] [ctx 0] Deparser 'deparser': start
[10:10:35.063] [bmv2] [D] [thread 1535] [34.0] [ctx 0] Deparser 'deparser': end
[10:10:35.063] [bmv2] [D] [thread 1539] [34.1] [ctx 0] Transmitting packet of size 70 out of port 4
[10:10:35.064] [bmv2] [D] [thread 1539] [34.0] [ctx 0] Transmitting packet of size 70 out of port 4
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [ctx 0] Processing packet received on port 4
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [ctx 0] Parser 'parser': start
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [ctx 0] Parser 'parser' entering state 'start'
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [ctx 0] Parser state 'start' has no switch, going to default next state
[10:12:13.368] [bmv2] [T] [thread 1534] [35.0] [ctx 0] Bytes parsed: 0
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [ctx 0] Parser 'parser': end
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [ctx 0] Pipeline 'ingress': start
[10:12:13.368] [bmv2] [T] [thread 1534] [35.0] [ctx 0] double-clone.p4(63) Condition "standard_metadata.ingress_port == 1" (node_2) is false
[10:12:13.368] [bmv2] [T] [thread 1534] [35.0] [ctx 0] double-clone.p4(67) Condition "standard_metadata.ingress_port == 4" (node_5) is true
[10:12:13.368] [bmv2] [T] [thread 1534] [35.0] [ctx 0] Applying table 'tbl_doubleclone68'
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [ctx 0] Looking up key:

```

Abbildung 3.13: Output `s1.log` aus Double-Clone

Im Ordner `pcap` wird für jeden In- und Outgoing-Switchport der Netzverkehr mitgeschnitten und kann mit Programmen wie *Wireshark*¹⁶ oder *tcpdump*¹⁷ inspiziert werden.

¹⁶<https://www.wireshark.org/>

¹⁷<https://www.tcpdump.org/>

3.5.3 Bedienung der Hosts

Die Konsole der Hosts kann in der CLI via `xterm HOSTNAME` gestartet werden. Hier können auch mehrere Hosts angegeben werden (Trennung durch *Leerzeichen*). In der Konsole können ICMP-Pakete (`ping`) und mit den beiden Python-Dateien `send.py` und `receive.py` Netzwerkpakete gesendet und empfangen werden. Gängige Linux-Netzwerkbefehle wie `ifconfig` können verwendet werden. Von hier wird auch Scapy gestartet und konfiguriert. Die Nutzung dieser Tools wird in Kapitel 5 Tests erklärt.

3.5.4 Beenden der Übung

Mit `exit` wird Mininet beendet. Danach müssen noch Log- und Build-Dateien gelöscht werden. Dies geschieht mittels der CLI-Befehle `make stop` und `make clean`. Diese Ordner sind prinzipiell vom Upload in das Git-Repo ausgeschlossen. Dies wurde in der Datei `.gitignore` hinterlegt.

Kapitel 4: Portierung auf reale Hardware

Dieses Kapitel beschäftigt sich mit der Hardware, welche im Labor genutzt wird. Dazu zählt auch die Netzwerktopologie im Labor, die Konfiguration und Benutzung der Geräte. Außerdem werden einige der Übungen aus Kapitel 3 Virtuelle Umgebung für den Einsatz auf der Hardware portiert. Die Programme sind im Git-Repo im Ordner `tofino-ports` gespeichert.

```
mbeausencourt@P4-PC: ~/Documents/p4-projekt/tofino-ports$ ls
total 24
11141141 4 drwxrwxr-x 6 mbeausencourt mbeausencourt 4096 Dez 22 15:58 .
11141591 4 drwxrwxr-x 17 mbeausencourt mbeausencourt 4096 Dez 22 10:46 ..
11142451 4 drwxrwxr-x 2 mbeausencourt mbeausencourt 4096 Dez 19 17:56 00_Testing
11141321 4 drwxrwxr-x 2 mbeausencourt mbeausencourt 4096 Dez 7 13:42 01_Hello_World
11141442 4 drwxrwxr-x 2 mbeausencourt mbeausencourt 4096 Dez 22 15:58 02_Simple_Clone
11403656 4 drwxrwxr-x 2 mbeausencourt mbeausencourt 4096 Jan 23 17:22 03_Double_Clone
```

Abbildung 4.1: Ordnerstruktur `tofino-ports`

4.1 Laborhardware

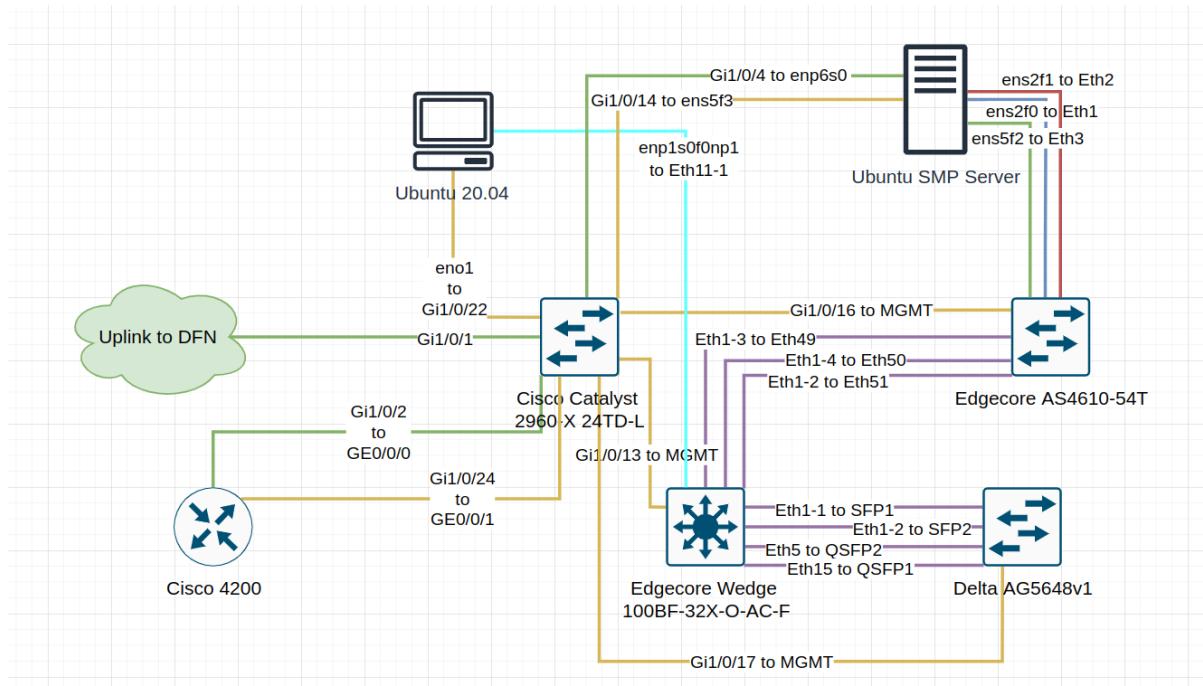
Im Labor sind drei Switche in einem Rack verbaut:

- Edge-Core DCS800 (Wedge 100BF-32X)(= P4-Switch)
- ⇒ Intel Tofino (Network Processing Unit) inside and P4-Programmable
- Edge-Core AS4610-54T
- Delta AG5648V1

Diese Switche sind sogenannte *Whitebox-Switche*. Auf diesen muss ein Betriebssystem installiert werden. Dafür wurde Ubuntu genutzt. Mit den Switchen sind ein Linux-PC und ein Linux-Server (beide Ubuntu 20.04.5 LTS) für *Secure Shell* (SSH)-Verbindungen und Tests angeschlossen. Auf dem Server ist eine virtuelle Instanz der *Tofino Native Architecture* (TNA) installiert. Sie beinhaltet einen Software-Switch mit TNA, welcher ähnlich wie der virtuelle Switch aus Kapitel 3 Virtuelle Umgebung funktioniert. Da die Übungen auf realer Hardware absolviert werden können, wird die virtuelle Instanz hier nicht weiter thematisiert. Diverse Kabel (RJ45¹, Glasfaser) und die dazugehörigen *Small Form-factor*

¹Registered Jack 45 = LAN-Kabel

Pluggables (SFPs)² liegen im Labor bereit und sind bereits in den Geräten installiert. Sowohl auf dem Server als auch auf dem P4-Switch ist das *Barefoot Software Development Environment* (*bf-sde*) installiert. Auf dem Server läuft Version 9.9.0; auf dem P4-Switch 9.9.1. Darin sind alle Dateien zum Kompilieren und Starten des P4-Programms enthalten. Wichtige Shell-Skripte sind hier und auch im Ordner `~/tools` enthalten. Da es auf dem Switch einen generischen Account (p4-user) gibt, muss die Installation von *bf-sde* sowie das Importieren des Git-Repos nicht erneut gemacht werden. Deswegen wird dies hier nicht erklärt.



Legende der Verkabelung/Links			
Farbe	Netname	Use	VLAN ID
Grün	DFN-Netz (Deutsches ForschungsNetz)	Internet Access	Vlan184
Gelb	172.16.1.0/24	P4-MGMT-Netz	Vlan172
Lila	High-Speed (>= 10Gbit/s)	P4-Data-Links	
Blau	NETNS Blue Context	P4-Test-Link	
Rot	NETNS Red Context	P4-Test-Link	
Grün	NETNS Green Context	P4-Test-Link	
Cyan	25Gbit/s Fiber Link	P4-Test-Link	

Abbildung 4.2: Netzwerktopologie im Labor

Abbildung 4.2 stellt das Netzwerk im Labor dar. *Cisco 4200* ist ein Router, welcher zwischen den verschiedenen Netzen und auch dem Internet routet. *Cisco Catalyst 2960-X* ist der zentrale Switch im Labor, auf welchem die verschiedenen Netze (Virtual Local Area Networks (VLANs)) konfiguriert sind. *Edgecore AS4610* verfügt auch über mehrere VLANs, welche die verschiedenen Testports des Servers auf Layer2 trennt, damit nicht irrtümlicherweise Pakete zwischen den drei verschiedenen Testports (blau, rot, grün) ausgetauscht werden

² optische Module, <https://community.fs.com/de/blog/sfp-module-what-is-it-and-how-to-choose-it.html>

können. Dies sorgt dafür, dass jeglicher P4-Netzverkehr³ des Servers immer via Ports Eth49, Eth50 oder Eth51 des AS4610 an den P4-Switch gesendet werden muss. Der cyane Link vom Ubuntu-PC zum P4-Switch stellt hier eine Besonderheit dar. Pakete können direkt vom P4-Switch an diesen Port des PCs gesendet werden und müssen nicht den normalen Weg via AS4610 nehmen. Außerdem ist die Geschwindigkeit des Ports (25Gbit/s) dafür geeignet Tests mit einer hohen Datenrate laufen zu lassen. Dafür wurde in den PC eine optische Netzwerkkarte mit SFPs verbaut. Die Geräte wurden mit Multimode-Glasfasern⁴ verbunden.

Der Switch *Delta AG5648V1* wird in den Übungen nicht genutzt und steht für zukünftige Anwendungen zur Verfügung.

Die drei Switche werden aufgrund von Energiesparmaßnamen nach jeder Benutzung ausgeschalten. Dafür ist im Rack ein Mehrfachstecker verbaut, welcher über das Netzwerk bedient werden kann. Hierfür wurden *Aliases* auf PC und Server erstellt (s. Anhang A). Ein **alias** ist unter Linux ein Kurzbefehl, welcher eigenständig festgelegt werden kann. Bevor der P4-Switch ausgeschalten wird, sollte sicher gestellt werden, dass das Git-Repo auf dem aktuellen Stand ist und Änderungen hochgeladen wurden. P4-Programme sollten zuvor auch beendet werden. Ansonsten kann es zum Datenverlust kommen, welcher eine komplette Neueinrichtung des Git-Repos mit sich bringen kann. Alternativ sind auf diesem Mehrfachstecker auch Knöpfe zum manuellen An-/Ausschalten vorhanden.

Für SSH-Verbindungen in diesem Netz wurden weitere Aliasses auf dem PC und/oder Server angelegt. Die Aliasses sind in Anhang A zu finden. Nützliche Einträge für die IP-Adressen in diesem Netz wurden in der Datei `/etc/hosts` angelegt und sind in Anhang B zu sehen. Die IPv4-Adressen der Interfaces des Servers und des PCs für die Tests sind in Anhang I zu sehen.

4.2 Konfiguration der Hardware

4.2.1 Wedge 100BF-32X

Der P4-Switch verfügt über 32x100Gbit/s Ports. Jeder dieser Ports kann auch via Breakout-Kabel⁵ und Konfiguration auf 4x10 oder 25Gbit/s bzw. 2x50Gbit/s geschalten werden. Um die Umgebungsvariable für *bf-sde* zu setzen und den dazugehörigen Treiber zu installieren, wurden folgende Befehle in die Datei `.profile` auf dem Switch eingetragen:

Codeauszug 4.1: Treiber und Umgebungsvariable in `.profile`

```
1 source ~/tools/set_sde.bash
2 sudo $SDE_INSTALL/bin/bf_kdrv_mod_load $SDE_INSTALL
```

³zur Unterscheidung von Internet-Netzverkehr

⁴<https://cleerlinefiber.com/2021/01/27/multimode-fiber-types/>

⁵Kabel, welches von 1 auf 4 Ports multiplext

Sie wurden dort eingetragen, da sie bei jeder neuen Shell-Session sonst manuell eingegeben werden müssten. Das Eintragen in die Datei sorgt dafür, dass diese beiden Befehle bei jeder neuen SSH-Session zum Switch ausgeführt werden. Wenn die Treiber bereits installiert wurden (nur einmal nach Reboot erforderlich), wird auf der Konsole des Switches eine zu vernachlässigende Fehlermeldung ausgegeben. Ohne diese Befehle ist ein Starten des Programms nicht möglich.

Kompilieren und Start des Programms

P4-Programme können über folgende Befehle kompiliert und gestartet werden:

Codeauszug 4.2: Vollständige Befehle für Start und Kompilieren des Programms

```
1 // Kompilieren
2 ~/tools/p4_build.sh PROGRAMMNAME.p4
3 // Starten des Programms
4 ./run_switchd.sh -p PROGRAMMNAME
```

Für das Starten des Programms ist es wichtig, zuvor in den SDE-Ordner zu wechseln (`sde`). Beim Kompilieren muss der `PROGRAMMNAME` mit Dateiendung (`.p4`) angegeben werden; beim Starten nur der Name ohne Dateiendung. Dafür wurden Aliasses auf dem P4-Switch erstellt:

Codeauszug 4.3: Aliasses für Kompilieren und Start

```
1 alias p4-compile='~/tools/p4_build.sh'
2 alias p4-start='cd sde; ./run_switchd.sh -p'
```

Diese werden wie die vollständigen Befehle unter Angabe des Programmnamens verwendet. Nach erfolgreichem Kompilieren und Start des Programms wechselt man auf die Shell (= `bfshell`) von `bf-sde`.

BFShell

In der `bfshell` müssen Switchports konfiguriert werden. Einträge können *Tabellen* inkl. *Actions* hinzugefügt werden. Außerdem beinhaltet diese Shell *Barefoot Runtime Python* (`bfprt_python`). Damit können Python⁶-Skripte ausgeführt werden, welche zur Konfiguration des Switches genutzt werden. Weitere Tools sind in der `bfshell` verfügbar.

⁶<https://www.python.org/>

UCLI

Die Konfiguration der Switchports wird aus der `bfshell` mittels `ucli` gestartet. Dafür sind pro Port drei Befehle nötig, um das Forwarding zu aktivieren:

Codeauszug 4.4: Port-Konfig in BFShell

```

1 // 1 - Port hinzufuegen
2 port-add <port_str> <speed (1G, 10G, 25G, 40G, 40G-R2, 50G(50G/50G-R2, ↵
   50G-R2-C, 50G-R1), 100G(100G/100G-R4, 100G-R2, 100G-R1), ↵
   200G(200G/200G-R4, 200G-R2, 200G-R8), 400G(400G/400G-R8, 400G-R4))> ↵
   <fec (NONE, FC, RS)>
3 // 2 - Autonegotiation einstellen
4 an-set <port_str> <autonegotiation (0 = automatisch, 2 = ausgeschalten)
5 // 3 - Port anschalten
6 port-enb <port_str>
```

Als Beispiel wird hier der zweite Breakout-Port des physikalisch ersten Ports (1/2) mit 10Gbit/s (10G) und keiner (NONE) *Forward Error Correction* (FEC)⁷ konfiguriert. Im zweiten Befehl wird *Auto Negotiation* (AN) ausgeschalten (2). Bei eingeschalteten Feature können zwei direkt miteinander verbundene Netzwerkgeräte die physikalischen Parameter der Verbindung (z.B. Geschwindigkeit, Duplex-Modus) eigenständig aushandeln. Da der Switch AS4610 dieses Feature nicht unterstützt, muss es auf den Ports des P4-Switches, welche mit dem AS4610 verbunden sind, ausgeschalten werden. Der dritte Befehl schaltet den gesamten ersten Port ein.

Codeauszug 4.5: Konfigurationsbeispiel eines Switchports

```

1 port-add 1/2 10G NONE
2 an-set 1/2 2
3 port-enb 1/-
```

Die vollständige Konfiguration der Switchports ist in Anhang C zu sehen. Anschließend kann mittels `pm show` die Portkonfiguration ausgegeben werden. Wichtige Spalten sind hierbei `D_P` (Portname für das P4-Programm und Tabellen) und `OPR` (= Operational Status: ist der Port aktiv mit einem anderen Gerät verbunden? UP \Rightarrow ja; DWN⁸ \Rightarrow nein). Die beiden `FRAMES`-Spalten zeigen an, wieviele Layer2-Frames auf dem jeweiligen Port empfangen (RX) bzw. gesendet wurden (TX). Mit `exit` wird `ucli` beendet und wieder in die `bfshell` gewechselt.

⁷<https://www.rfc-editor.org/rfc/rfc6363>

⁸DWN = Down

PORT	MAC	ID_P P PT SPEED	FEC	AN KR RDY ADM OPR LPBK	FRAMES RX	FRAMES TX	E
1/0	23/0 132 2/ 4 10G	NONE Au Au YES ENB DWN	NONE		0	0	
1/1	23/1 133 2/ 5 10G	NONE Ds Au YES ENB DWN	NONE		0	0	
1/2	23/2 134 2/ 6 10G	NONE Ds Au YES ENB DWN	NONE		0	0	
1/3	23/3 135 2/ 7 10G	NONE Ds Au YES ENB DWN	NONE		0	0	
11/0	13/0 40 0/40 25G	NONE Ds Au YES ENB DWN	NONE		0	0	
11/1	13/1 41 0/41 25G	NONE Ds Au YES ENB DWN	NONE		0	0	
11/2	13/2 42 0/42 25G	NONE Ds Au YES ENB UP	NONE		0	0	
11/3	13/3 43 0/43 25G	NONE Ds Au YES ENB UP	NONE		0	0	

Abbildung 4.3: Ausgabe Portkonfiguration

BFRT

Mit dem Befehl `bfprt` kann in die *Barefoot Runtime* gewechselt werden, um *Tabelleneinträge* anzulegen. Hier werden zunächst Standardnodes⁹ (z.B. `mirror` oder `port`) und die Node angezeigt, welche automatisch den Namen des P4-Programms übernimmt. In dieser sind die im Programm angelegten *Tabellen* zu finden. In `mirror` sind die Tabellen gespeichert, welche für die Paketvervielfältigung genutzt werden. Nach folgendem Schema kann in eine der Programm-Tabellen gewechselt werden:

```
1 | bfprt.P4-PROGRAMMNAME.pipe.KONTROLLBLOCKNAME.TABELLENNAME
```

Diese Befehle können auch einzeln, nacheinander eingegeben werden. Auto vervollständigung der Befehle via *Tab*-Taste ist wie in Linux möglich.

Mit dem Befehl `info()` können Größe (`capacity`), Anzahl der Einträge (`Usage`), *Key-Fields* mit deren *Match-Action-Types*, Parameter (`Data Fields`) und *Actions* der Tabelle angezeigt werden. Die Einträge einer *Tabelle* können mittels `dump()` ausgegeben werden. Folgende Befehle können genutzt werden, um Tabelleneinträge zu verändern:

Codeauszug 4.6: Befehle für Bearbeitung von Tabelleneinträgen

```
1 | // Eintrag mit Action hinzufuegen
2 | add_with_ACTIONNAME(KEY, PARAMETER)
3 | // Eintrag mit Action bearbeiten
4 | mod_with_ACTIONNAME(KEY, PARAMETER)
5 | // Default-Action einstellen
6 | set_default_with_ACTIONNAME(KEY, PARAMETER)
```

Key, *Action* und deren Parameter müssen immer angegeben werden. Bei Einträgen mit *Match-Kriterium* `ternary` muss zusätzlich eine *Maske* angegeben werden (s. Anhang G).

⁹diese werden automatisch angelegt

Beenden des Programms

Das laufende P4-Programm kann aus der *bfshell* mittels folgender Befehle beendet werden:

Betriebssystem	Tastaturbefehl
Windows	Ctrl + 4
Linux	Ctrl + AltGr + ß
macOS	Ctrl + Alt + Shift + 7

Tabelle 4.1: Tastaturbefehle zum Beenden des P4-Programms

4.2.2 IP-Konfiguration Server und PC

IP-Konfiguration Server

Die drei Test-Interfaces des Servers (vgl. Abbildung 4.2: blauer, roter, grüner Link) werden mit dem Tool `ip netns`¹⁰ konfiguriert. `netns` steht für *network namespaces*. Es ermöglicht die Erstellung von *Namenskontexten* (engl. *namespaces*). Diese Namenskontakte erlauben die Konfiguration von Interfaces mit eigener Routing-Tabelle und Firewall-Regeln. Dies unterbindet eine direkte Kommunikation der Interfaces untereinander [s. [IP-NetNS]]. Dies ist für die Tests wichtig, damit sicher gestellt wird, dass die gesendeten Pakete den P4-Switch durchlaufen. Jedem Namespace muss ein Interface, ein Name und eine IP-Adresse zugewiesen werden, bevor das Interface aktiviert wird. Als Beispiel dient der nachfolgende Codeauszug. Die gesamte Konfiguration ist in Anhang D zu sehen. Eine vollständige IPv4-Adressliste für die Testinterfaces ist in Anhang I einsehbar.

Codeauszug 4.7: Konfiguration Namespace blue auf Server

```

1 // Erstellung Namespace 'blue'
2 sudo ip netns add blue
3
4 // Aktivieren des physischen Interfaces ens2f0
5 sudo ifconfig ens2f0 up
6
7 // Hinzufuegen des Interfaces zu Namespace
8 sudo ip link set dev ens2f0 netns blue
9
10 // Konfiguration IP-Adresse + Aktivieren Interface (up) + Loopback (lo)
11 sudo ip netns exec blue ifconfig ens2f0 10.1.1.1/24 up
12 sudo ip netns exec blue ifconfig lo up
13
14 // Ausgabe der Namespace-Interface-Konfiguration

```

¹⁰<https://www.man7.org/linux/man-pages/man8/ip-netns.8.html>

```
15 | sudo ip netns exec blue ifconfig
```

IP-Konfiguration PC

Auf dem Ubuntu-PC wurde lediglich das optische Interface `enp1s0f1np1` mit der IP-Adresse `10.1.1.5/24` ohne Gateway im *Graphical User Interface* (GUI) konfiguriert.

4.2.3 Konfiguration AS4610

Der P4-Switch hat nur optische Interfaces (*Portgeschwindigkeit* $\geq 10\text{GBit/s}$). Da der PC ursprünglich nur 1GBit/s-RJ45-Anschlüsse besaß, mussten diese an den AS4610 angeschlossen werden. Um wie in Abschnitt 4.2.2 zu gewährleisten, dass alle Pakete durch den P4-Switch bearbeitet werden, wurde auch AS4610 dementsprechend konfiguriert. Die verbundenen Testports (Eth1-Eth3) des Servers wurden mittels *VLANs* getrennt. Um die Kommunikation mit dem P4-Switch zu ermöglichen, wurde für jeden dieser Ports und VLANs ein Pendant mit Verbindung zum P4-Switch angelegt (Eth49-Eth51).

AS4610 Ports	VLAN ID	Server-Port	Namespace	Ports P4-Switch
Eth1, Eth49	10	ens2f0	blue	Eth1/3
Eth2, Eth50	20	ens2f1	red	Eth1/4
Eth3, Eth51	30	ens5f2	green	Eth1/2

Tabelle 4.2: Portkonfig AS4610 und korrespondierende Ports

Die Dateien, welche diese Konfigurationen enthalten, sind im Verzeichnis `~/etc/systemd/network/` gespeichert. Der Inhalt dieser Dateien ist in Anhang E zu sehen. Die Dokumentation für die Konfiguration des Switches ist online zugänglich:
https://docs.bisdn.de/platform_configuration.html

4.3 Portierung des Codes von BMv2 auf TNA

Da *TNA* einen anderen Aufbau wie *BMv2* besitzt (vgl. Abschnitt 2.3.1) und *Actions* anders benutzt werden, muss ein für *BMv2* geschriebenes P4-Programm angepasst werden. Die Übungen 3.3.1, 3.3.2 und 3.3.3 wurden für den Einsatz auf dem P4-Switch portiert. Diese wurden im Git-Repo unter `tofino-ports/` gespeichert. In diesem Verzeichnis wurde auch ein Ordner für Tests angelegt. Tests bezieht sich hierbei auf Veränderungen im P4-Code. Python-Skripte - wie in Abschnitt 4.2.1 beschrieben - wurden in den beiden Clone-Übungen hinzugefügt, um deren *Tabellen* zu befüllen.

4.3.1 Allgemeines

Die *TNA* muss zu Beginn des Programms mittels `#include <tna.p4>` importiert werden. Die Kontrollblöcke müssen wie in Abschnitt 2.3.1 angelegt werden. Die Parameter der einzelnen *Kontrollblöcke* müssen wie in `tofino1_arch.p4`¹¹ übergeben werden. In der Datei `tofino1_base.p4`¹² ist auch ersichtlich, welche Variablen in den einzelnen Parametern enthalten sind.

4.3.2 Änderungen Parser

Im *Parser* ist es wichtig, dass im `state start` die *Ingress Intrinsic Metadata* mit dem Befehl `pkt.extract(ig_intr_md)`¹³ extrahiert werden. `pkt` und `ig_intr_md` sind die Namen der *Parameter* `packet_in` und `ingress_intrinsic_metadata`. Um eine *Checksum* mit `ipv4_checksum.verify()` zu überprüfen, muss diese mittels `Checksum() NAME` deklariert werden.

4.3.3 Änderungen Deparser

Sofern ein *Header* bei der Verarbeitung verändert wurde, muss für das Senden des Pakets `pkt.emit(hdr)` im `apply`-Block ausgeführt werden. Finden keine Änderungen der Header-Daten statt, muss weder in *TNA* noch in *Behavioral Model Version 2* (BMv2) ein `emit` benutzt werden (s. Übung Hello World).

Um die *Checksum* neu zu berechnen, muss diese via `Checksum() NAME` deklariert werden. Der Code dafür ist in Anhang F zu sehen. Um Pakete zu vervielfältigen muss mit `Mirror()` `MIRRORNAME` eine Mirror-Instanz angelegt und mit `MIRRORNAME.emit<mirror_hdr>(MIRROR_SESSION_ID, {MIRROR_DATA})` erzeugt werden. Dafür ist es zwingend notwendig, zuvor auch den *Mirror-Typ* mittels *if-Bedingung* zu überprüfen [s. [TNA]].

4.3.4 Änderungen bei der Paketvervielfältigung

Die *Clone-Session-IDs* aus Übung 3.3.2 (Simple Clone) müssen auch in der *TNA* für die Paketvervielfältigung genutzt werden. Diese werden in den Übungen *Simple-Clone* und *Double-Clone* in der *Tabelle port_acl* gesetzt. Nachdem das Paket mittels `emit` vervielfältigt wurde, wird die *Clone-Session-ID* in der *Tabelle mirror* in der *BFRT* ausgelesen und mit der konfigurierten *Action* gesendet. In den Projektordnern dieser beiden Übungen sind Python-Skripte angelegt, welche nach Start des Programms und Konfigurieren der Switchports mit folgendem Befehl auf dem P4-Switch angewendet werden können:

¹¹https://github.com/barefootnetworks/Open-Tofino/blob/master/share/p4c/p4include/tofino1_arch.p4

¹²https://github.com/barefootnetworks/Open-Tofino/blob/master/share/p4c/p4include/tofino1_base.p4

¹³im Egress Parser: `eg_intr_md`

Codeauszug 4.8: Ausführen eines Python-Skripts zum Anlegen von Tabelleneinträgen

```

1 // Allgemeine Syntax
2 ~/bf-sde-9.9.1/run_bfshell.sh -b ~/PROJEKTPFAD/SKRIPTNAME.py -i
3 // Beispiel fuer Simple-Clone
4 ~/bf-sde-9.9.1/run_bfshell.sh -b ←
    ~/p4-projekt/tofino-ports/02_Simple_Clone/basic_table_setup_simple_clone.py ←
    -i

```

Beim Ausführen des Skripts ist es wichtig, dass keine andere Instanz von *BFRT* läuft. Ansonsten wird eine Fehlermeldung ausgegeben. Sollte eine Instanz geöffnet sein, kann diese mittels `exit` beendet werden und das Ausführen des Skripts ist möglich.

Nachfolgender Codeauszug zeigt das Anlegen von zwei Tabelleneinträgen aus einem Python-Skript. Der erste Befehl setzt die *Clone-Session-ID* (`mirror_session=1`) anhand des *Ingress-Ports*. Die 9-bit lange `ingress_port_mask`¹⁴ mit dem Wert 510 ist das *Match-Kriterium* (= *ternary*) des Eintrags. Sie signalisiert in Verbindung mit dem *Key* (`ingress_port=134`), dass für auf den Ports 134 und 135 empfangene Pakete `mirror_session = 1` gesetzt werden soll. Der zweite Befehl setzt für alle vervielfältigten Pakete mit *Clone-Session-ID 1* (`sid=1`) den *Egress-Port* und validiert diesen. Außerdem wird diese *Clone-Session* sowohl für In- und Outgoing-Netzverkehr (`direction='BOTH'`) aktiviert (`session_enable=True`).

Mindestens diese fünf Angaben müssen für die Tabelle `mirror.cfg` gemacht werden. Ansonsten funktioniert die Vervielfältigung auf dem P4-Switch nicht. Es gibt noch viele, weitere Optionen, die in dieser Tabelle gesetzt werden können. Das gesamte Skript ist in Anhang G zu sehen.

Codeauszug 4.9: Auszug aus Python-Skript für Simple-Clone

```

1 # Set Mirror Session to 1 for Ingress Ports 134-135 (QSFP1-3 und 1-4)
2 p4.Ingress.port_acl.add_with_acl_mirror(ingress_port=134, ←
    ingress_port_mask=510, mirror_session=1)
3
4 # Set for clone-id=1 with action=normal: enable session, direction=BOTH ←
    (in+egress), egress_port=40, validate port
5 bfrt.mirror.cfg.add_with_normal(sid=1, session_enable=True, ←
    direction='BOTH', ucast_egress_port=40, ucast_egress_port_valid=True)

```

4.3.5 Anpassung der Ports

Eine der wichtigsten Anpassungen bei der Portierung ist die der physikalischen Schnittstellen (Ports). Der *Egress-Port* wird mit `ig_tm_md.udcast_egress_port` gesetzt (BMv2: `standard_metadata.egress_spec`). Die Portnummer kann wie in Abschnitt 4.2.1 (s. Abbildung 4.3, Spalte D_P) ermittelt und anschließend im Programm eingetragen werden.

¹⁴Ingress_Ports sind 9 Bit lang, daher muss auch die Maske so lang sein

Codeauszug 4.10: Vergleich Übung Hello_World: Setzen Egress-Port in BMv2 und TNA

```
1 // BMv2-Switch Port 2 als Egress-Port setzen
2     standard_metadata.egress_spec = 2
3
4 // P4-Switch Port Eth1-4 als Egress-Port setzen
5     ig_tm_md.unicast_egress_port = 135
```


Kapitel 5: Tests

Dieses Kapitel beschäftigt sich mit verschiedenen Möglichkeiten zum Testen der P4-Programme auf Funktionalität. Die Tools `ip-netns`, `ping`, `iperf` und `scapy` sind bereits auf dem Server installiert. `Wireshark` ist auf dem PC und den virtuellen Hosts installiert. In der virtuellen Umgebung werden außerdem `ping` und zwei Python-Skripte genutzt. Die Tests mit den verschiedenen Tools haben folgendes Ziel:

1. Generierung von Paketen auf einem Host
2. Senden dieser Pakete an einen anderen Host
3. Erfolgreiches Empfangen der Pakete auf dem zweiten Host
4. Darstellung und Verifikation des Netzverkehrs

5.1 Tests in der virtuellen Umgebung

Nach Starten des Programms müssen zunächst die Konsolen der Hosts geöffnet werden (`xterm HOSTNAME` s. Abschnitt 3.5.3).

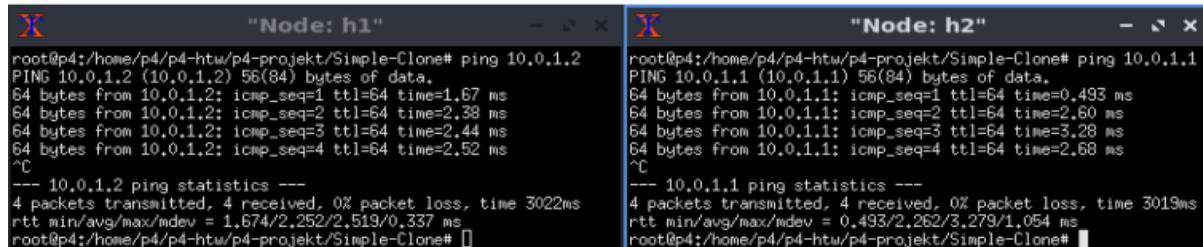
5.1.1 Testen mit Ping

Pings nach [RFC792] funktionieren nur für folgende Übungen:

- Hello World
- Simple-Clone: H1 \longleftrightarrow H2
- Resubmit
- Resubmit3-ARP

Dies resultiert aus der Tatsache, dass das *Internet Control Message Protocol* (ICMP) (= `ping`) das *Address Resolution Protocol* (ARP) nutzt, um zur gewünschten Ziel-IP-Adresse noch die MAC-Adresse des Ziel-Hosts zu erlangen. Dieser Mechanismus nutzt dafür Broadcast. Broadcasts und ARP wurden erst in Übung *Resubmit3-ARP* vollständig implementiert. Die anderen drei Übungen nutzen statisches Port-Forwarding. Deshalb ist `pingen` in diesen Übungen möglich. Der abgebildete Test wurde in der Übung *Simple Clone*

absolviert. Bei manchen Tests kann nach den einzelnen Zeilenausgaben in Abbildung 5.1 noch ein (DUP!) am Ende der Zeile stehen. Dies signalisiert, dass es sich bei dem Paket um ein Duplikat handelt und ist eine Folge des *Mirrorings*.



The image shows two terminal windows side-by-side. The left window is titled "Node: h1" and the right window is titled "Node: h2". Both windows are running a ping command between two hosts.

```
root@p4:/home/p4/p4-htw/p4-projekt/Simple-Clone# ping 10.0.1.2
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
64 bytes from 10.0.1.2: icmp_seq=1 ttl=64 time=1.67 ms
64 bytes from 10.0.1.2: icmp_seq=2 ttl=64 time=2.38 ms
64 bytes from 10.0.1.2: icmp_seq=3 ttl=64 time=2.44 ms
64 bytes from 10.0.1.2: icmp_seq=4 ttl=64 time=2.52 ms
^C
--- 10.0.1.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3022ms
rtt min/avg/max/mdev = 1.674/2.252/2.519/0.337 ms
root@p4:/home/p4/p4-htw/p4-projekt/Simple-Clone# 
```

```
root@p4:/home/p4/p4-htw/p4-projekt/Simple-Clone# ping 10.0.1.1
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=64 time=0.493 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=2.60 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=3.28 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=2.68 ms
^C
--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3019ms
rtt min/avg/max/mdev = 0.493/2.262/3.279/1.054 ms
root@p4:/home/p4/p4-htw/p4-projekt/Simple-Clone# 
```

Abbildung 5.1: Pings in VM

5.1.2 Python Skripte

Die Python-Skripte `send.py` und `receive.py` sind in jedem Übungsordner angelegt. Mit dem ersten können Testpakete gesendet und mit dem zweiten dargestellt werden, nachdem sie empfangen wurden. Diese müssen aus der Konsole der Hosts nach folgender Syntax gestartet werden:

```
1 // send.py
2 python3 send.py ZIEL-IP-ADRESSE "NACHRICHT"
3 // receive.py
4 python3 receive.py
```

Die beiden Skripte müssen auf unterschiedlichen Hosts gestartet werden. **ZIEL-IP-ADRESSE** ist die IP-Adresse des Hosts, auf welchem `receive.py` ausgeführt wird. Als Beispiel wurde in Übung 3.3.3 ein Paket vom ersten Host (H1) an den zweiten (H2) mit dem Inhalt „dies ist ein p4-test“ gesendet. Das Paket wird vom ersten Switch (S1) geklont. Original und Klon werden dann an den zweiten Switch (S2) gesendet und von diesem wiederum geklont. Jeweils zwei Pakete werden dann an H2 und den dritten Host (H3) gesendet und werden in der Konsole ausgegeben (s. Abbildung 5.2). *Pings* werden mit `receive.py` nicht ausgegeben.

```

[N] "Node: h3"
root@p4:/home/p4/tutorials/p4-projekt/Double-Clone# python3 send.py 10.0.1.2 "d
ies ist ein p4-test"
sending on interface eth0 to 10.0.1.2
###[ Ethernet ]###
dst      = ff:ff:ff:ff:ff:ff
src      = 08:00:00:00:01:11
proto   = IPv4
type    = IP
###[ IP ]###
version = 4
ihl     = 5
tos     = 0x0
len     = 60
id      = 1
flags   =
frag   = 0
ttl    = 64
proto  = TCP
checksum = 0x64b9
src     = 10.0.1.1
dst     = 10.0.1.2
options \
###[ TCP ]###
sport   = 59442
dport   = 1234
seq     = 0
ack     = 0
dataofs = 5
reserved = 0
flags   = S
window  = 8192
checksum = 0x68c2
urgptr  = 0
options = []
###[ Raw ]###
load   = 'dies ist ein p4-test'
root@p4:/home/p4/tutorials/p4-projekt/Double-Clone# []

[N] "Node: h2"
root@p4:/home/p4/tutorials/p4-projekt/Double-Clone# python3 receive.py
sniffing on eth0
got a packet
###[ Ethernet ]###
dst      = ff:ff:ff:ff:ff:ff
src      = 08:00:00:00:01:11
proto   = IPv4
type    = IP
###[ IP ]###
version = 4
ihl     = 5
tos     = 0x0
len     = 60
id      = 1
flags   =
frag   = 0
ttl    = 64
proto  = TCP
checksum = 0x64b9
src     = 10.0.1.1
dst     = 10.0.1.2
options \
###[ TCP ]###
sport   = 59442
dport   = 1234
seq     = 0
ack     = 0
dataofs = 5
reserved = 0
flags   = S
window  = 8192
checksum = 0x68c2
urgptr  = 0
options = []
###[ Raw ]###
load   = 'dies ist ein p4-test'

[N] "Node: h3"
root@p4:/home/p4/tutorials/p4-projekt/Double-Clone# python3 receive.py
sniffing on eth0
got a packet
###[ Ethernet ]###
dst      = ff:ff:ff:ff:ff:ff
src      = 08:00:00:00:01:11
proto   = IPv4
type    = IP
###[ IP ]###
version = 4
ihl     = 5
tos     = 0x0
len     = 60
id      = 1
flags   =
frag   = 0
ttl    = 64
proto  = TCP
checksum = 0x64b9
src     = 10.0.1.1
dst     = 10.0.1.2
options \
###[ TCP ]###
sport   = 59442
dport   = 1234
seq     = 0
ack     = 0
dataofs = 5
reserved = 0
flags   = S
window  = 8192
checksum = 0x68c2
urgptr  = 0
options = []
###[ Raw ]###
load   = 'dies ist ein p4-test'

```

Abbildung 5.2: Python-Tests in der VM

5.2 Tests auf der Laborhardware

Tests auf der Hardware können gestartet werden, sobald das P4-Programm auf dem Switch ausgeführt, die Switchports konfiguriert und benötigte Tabelleneinträge hinzugefügt wurden (s. Kapitel 4).

5.2.1 Pingen auf der Hardware

Ping funktioniert auf der Hardware wie in der Virtuellen Umgebung. Für diese Tests werden die drei Interfaces des Servers aus den *ip-netns-Namenskontexten* verwendet (s. Abschnitt 4.2.2). Für jedes Interface wurde ein eigener *Ping-Alias angelegt* (s. Anhang A Z.29-31). Mit dem Befehl `blueping 10.1.1.2` wird die IP-Adresse des Namenskontexts `red` vom Interface des Kontexts `blue` gepingt.

5.2.2 iPerf-Tests

iPerf ist ein Tool, um aktiv die Datengeschwindigkeit in IP-Netzen zu messen. Dabei können viele nützliche Optionen gesetzt werden, um das Tool für die entsprechenden Bedürfnisse und Anforderungen zu optimieren. Ein paar dieser Optionen sind im nachfolgenden erklärt. Weitere Optionen von iPerf können in der Online-Dokumentation des Tools nachgelesen

werden: <https://iperf.fr/iperf-doc.php>

iPerf wird im Abschnitt 5.2.4 noch einmal zur Generierung von Netzverkehr benutzt.

iPerf Server

Um Tests mit iPerf durchführen zu können, muss man auf einem der Test-Ports des Ubuntu-Servers (`blue`, `red`, `green`) einen iPerf-Server starten. Dies wird in iPerf mit der Option `-s` signalisiert. Im Beispiel wurde in der CLI das Interface aus dem ip-netns Kontext `blue` mit der IP-Adresse 10.1.1.1 verwendet (Zeile 1). Sobald ein Client eine Verbindung zu diesem Server herstellt, wird dies angezeigt (Zeile 6). Anschließend wird die Dauer, Menge und Geschwindigkeit der ausgetauschten Daten ausgegeben.

Codeauszug 5.1: iPerf-Server Konfiguration und Output

```
1 sudo ip netns exec blue iperf -s
2 -----
3 Server listening on TCP port 5001
4 TCP window size: 128 KByte (default)
5 -----
6 [ 4] local 10.1.1.1 port 5001 connected with 10.1.1.2 port 42662
7 [ ID] Interval Transfer Bandwidth
8 [ 4] 0.0-10.0 sec 1.10 GBytes 941 Mbits/sec
```

iPerf Client

Als Client wird im Beispiel das Interface aus dem Namenskontext `red` verwendet. Ein Client wird mit der Option `-c` gesetzt. Über `-p` wird der Destination-Port¹ eingestellt, welcher im iPerf-Server-Code in Zeile 3 steht. Mit `-t` kann die Dauer des Tests in Sekunden angegeben werden. `-i` beschreibt die Zeit in Sekunden, nach der Informationen über den Datenverkehr ausgegeben werden (hier: alle 2 Sekunden).

Codeauszug 5.2: iPerf-Client Konfiguration und Output

```
1 sudo ip netns exec red iperf -c 10.1.1.1 -p 5001 -t 10 -i 2
2 -----
3 Client connecting to 10.1.1.1, TCP port 5001
4 TCP window size: 1.23 MByte (default)
5 -----
6 [ 3] local 10.1.1.2 port 42662 connected with 10.1.1.1 port 5001
7 [ ID] Interval Transfer Bandwidth
8 [ 3] 0.0- 2.0 sec 226 MBytes 949 Mbits/sec
9 [ 3] 2.0- 4.0 sec 225 MBytes 943 Mbits/sec
10 [ 3] 4.0- 6.0 sec 224 MBytes 941 Mbits/sec
11 [ 3] 6.0- 8.0 sec 224 MBytes 941 Mbits/sec
```

¹für Layer4: TCP (RFC9293) oder UDP (RFC768)

```
12 [ 3] 8.0-10.0 sec 224 MBytes 940 Mbits/sec
13 [ 3] 0.0-10.0 sec 1.10 GBytes 943 Mbits/sec
```

5.2.3 Scapy-Tests

Scapy ist ein auf Python basierendes CLI-Tool. Es ermöglicht das Erzeugen von spezifischen Netzwerkpaketen und bietet auch die Möglichkeit selbst-definierte Header (s. Abschnitt 2.4) zu schreiben und in das Paket einzubinden. Scapy kann für alle Interfaces oder in den *ipnetns* Namenskontexten gestartet werden. Da Scapy ein vielfältiges Tool mit verschiedenen Funktionen ist, empfiehlt es sich die Online-Dokumentation zu lesen². Das Tool wird aus dem Terminal mittels `scapy` gestartet. Mit `exit` wird es beendet.

Erzeugen eines Pakets

Um ein Paket in Scapy zu erzeugen, muss dieses zuerst aus den Headern der gewünschten Netzwerkschichten gebaut werden. Dies wird hier anhand des Ethernet- und IPv4-Headers gezeigt. Nach dem Erzeugen der Header müssen diese in einem Paket verbunden werden.

Codeauszug 5.3: Erzeugen der einzelnen Header und Zusammenfügen dieser

```
1 # Erzeugen des Ethernet- und IPv4-Headers
2 ethernet_header=Ether(type=0x800, dst="00:26:55:ec:f4:41")
3 ip_header= IP(dst="10.1.1.2", proto=253,flags="DF")
4
5 # Zusammenfuegen der beiden Header in "packet"
6 packet = ethernet_header / ip_header
```

Die Quell-MAC-Adresse übernimmt Scapy vom Interface, welches das Paket sendet. In diesem Beispiel wurde Scapy direkt in einem Namenskontext ausgeführt.

Verifikation des Pakets

Es gibt zwei Funktionen (`show()` und `show2()`), welche den Inhalt des erstellten Pakets oder eines Headers anzeigen. Die Syntax für den Aufruf ist dabei folgende: `NAME.show()` bzw. `NAME.show2()`. Als `NAME` muss hierbei der Name des Pakets oder des jeweiligen Headers angegeben werden.

²<https://scapy.readthedocs.io/en/latest/usage.html>

Senden des Pakets

Es gibt zwei Funktionen (`send(PACKETNAME)` und `sendp(PACKETNAME, ARGS)`), um ein Paket zu senden. Wird Scapy aus einem der Namenskontexte gestartet, so reicht das Verwenden der ersten Funktion. Die zweite Funktion erlaubt es in `ARGS` das sendende Interface anzugeben.

Codeauszug 5.4: Befehle zum Senden von Paketen

```
1 # einfache Funktion
2 send(packet)
3 # erweiterte Funktion -> Senden via Interface "eth0"
4 sendp(packet, iface="eth0")
```

Sniffing

Sniffing bezeichnet den Vorgang des Mitschneidens von Netzverkehr auf einem Interface. Mit folgendem Befehl wird in Scapy sämtlicher Netzverkehr auf der Kommandozeile ausgegeben: `sniff(prn=lambda x: x.show())` Es kann auch der Interfacename angegeben werden, um nur Pakete des angegebenen Interfaces anzuzeigen.

5.2.4 Verifikation mit Wireshark

Wireshark³ ist ein Tool, um Netzverkehr mitzuschneiden und anzuzeigen. Es verfügt über ein *Graphical User Interface* (GUI) und gibt den Netzverkehr dort aus. Das Programm muss mit Admin-Rechten gestartet werden. Nach Starten der Software muss ein zu inspizierendes Interface ausgewählt werden. Die Ausgabe des Netzverkehrs kann mittels Filtern⁴ angepasst werden. Wireshark ist aktuell nur vom Ubuntu-PC nutzbar.

Die Übungen 3.3.2 und 3.3.3 können mit Wireshark in ihrer für die Hardware angepassten Version genutzt werden. Sämtlicher Netzverkehr zwischen den beiden Interfaces `blue` und `red` des Servers wird vom P4-Switch auf den 25Gbit/s-Port des PCs geklont. Die Konfiguration der Tabellen findet durch die Python-Skripte in den Ordnern der jeweiligen Übungen statt. Wireshark muss auf dem Interface `enp1s0f1np1` des PCs gestartet werden. Startet man einen iPerf-Test wie in Abschnitt 5.2.2, so wird der Netzverkehr dieses iPerf-Tests in Wireshark ausgegeben. Ein kleiner Ausschnitt dieses Netzverkehrs ist in Abbildung 5.3 zu sehen. Der gesamte Netzverkehr des 10-sekündigen Tests beinhaltet über 80000 Pakete (bei 1Gbit/s und 10 Sekunden Dauer). Anhand der *Source* und *Destination*⁵ in den Abbildungen 5.3 und 5.4 ist ersichtlich, dass es sich um Pakete zwischen `red` und `blue` handelt.

³<https://wiki.wireshark.org/Home>

⁴z.B. Ausgabe von nur IPv4-Traffic

⁵es handelt sich hierbei um IPv4-Adressen

No.	Time	Source	Destination	Protocol	Length	Info
81601	9.969169726	10.1.1.2	10.1.1.1	TCP	66	[TCP ACKed unseen segment] 5001 → 50214
81602	9.969169848	10.1.1.1	10.1.1.2	TCP	50746	50214 → 5001 [ACK] Seq=1171814273 Ack=50214
81603	9.969203365	10.1.1.2	10.1.1.1	TCP	66	5001 → 50214 [ACK] Seq=1 Ack=1171864
81604	9.969203468	10.1.1.1	10.1.1.2	TCP	14546	50214 → 5001 [PSH, ACK] Seq=1171864 Ack=50214
81605	9.969217426	10.1.1.1	10.1.1.2	TCP	8754	50214 → 5001 [ACK] Seq=1171879433 Ack=50214
81606	9.970200511	10.1.1.2	10.1.1.1	TCP	66	[TCP ACKed unseen segment] 5001 → 50214
81607	9.970200802	10.1.1.2	10.1.1.1	TCP	66	[TCP ACKed unseen segment] 5001 → 50214
81608	9.970200902	10.1.1.1	10.1.1.2	TCP	56538	50214 → 5001 [PSH, ACK] Seq=1171886 Ack=50214
81609	9.970267667	10.1.1.2	10.1.1.1	TCP	66	[TCP ACKed unseen segment] 5001 → 50214
81610	9.970267761	10.1.1.1	10.1.1.2	TCP	31922	50214 → 5001 [ACK] Seq=1171944593 Ack=50214
81611	9.970311230	10.1.1.2	10.1.1.1	TCP	66	[TCP ACKed unseen segment] 5001 → 50214
81612	9.970311324	10.1.1.1	10.1.1.2	TCP	33370	50214 → 5001 [PSH, ACK] Seq=1171976 Ack=50214
81613	9.970332144	10.1.1.1	10.1.1.2	TCP	8754	50214 → 5001 [ACK] Seq=1172009753 Ack=50214
81614	9.970338487	10.1.1.1	10.1.1.2	TCP	1514	50214 → 5001 [ACK] Seq=1172018441 Ack=50214
81615	9.971200109	10.1.1.2	10.1.1.1	TCP	66	5001 → 50214 [ACK] Seq=1 Ack=1172018441
81616	9.971200304	10.1.1.2	10.1.1.1	TCP	66	[TCP ACKed unseen segment] 5001 → 50214

Abbildung 5.3: Geklonter Netzverkehr auf 25GBit/s Interface des PCs zwischen blue und red mit iPerf

No.	Time	Source	Destination	Protocol	Length	Info
8	9.849266995	10.1.1.1	10.1.1.2	ICMP	98	Echo (ping) request id=0x4e06, seq=1/25
9	9.849444057	10.1.1.2	10.1.1.1	ICMP	98	Echo (ping) reply id=0x4e06, seq=1/25
10	9.853023386	10.1.1.1	10.1.1.2	ICMP	98	Echo (ping) request id=0x4e06, seq=1/25
11	9.853172807	10.1.1.2	10.1.1.1	ICMP	98	Echo (ping) reply id=0x4e06, seq=1/25
12	9.853830915	10.1.1.2	10.1.1.1	ICMP	98	Echo (ping) reply id=0x4e06, seq=1/25
13	9.854187253	10.1.1.2	10.1.1.1	ICMP	98	Echo (ping) reply id=0x4e06, seq=1/25
14	10.851489250	10.1.1.1	10.1.1.2	ICMP	98	Echo (ping) request id=0x4e06, seq=2/51
15	10.851489583	10.1.1.2	10.1.1.1	ICMP	98	Echo (ping) reply id=0x4e06, seq=2/51
16	11.857908483	10.1.1.1	10.1.1.2	ICMP	98	Echo (ping) request id=0x4e06, seq=3/76
17	11.857908740	10.1.1.2	10.1.1.1	ICMP	98	Echo (ping) reply id=0x4e06, seq=3/76
18	12.881855300	10.1.1.1	10.1.1.2	ICMP	98	Echo (ping) request id=0x4e06, seq=4/10
19	12.881855471	10.1.1.2	10.1.1.1	ICMP	98	Echo (ping) reply id=0x4e06, seq=4/10
20	14.929797523	HewlettP_ec:f4:41	HewlettP_ec:f4:40	ARP	60	Who has 10.1.1.1? Tell 10.1.1.2
21	14.929797915	HewlettP_ec:f4:40	HewlettP_ec:f4:41	ARP	60	Who has 10.1.1.2? Tell 10.1.1.1
22	14.929798028	HewlettP_ec:f4:40	HewlettP_ec:f4:41	ARP	60	10.1.1.1 is at 00:26:55:ec:f4:40
23	14.929798140	HewlettP_ec:f4:41	HewlettP_ec:f4:40	ARP	60	10.1.1.2 is at 00:26:55:ec:f4:41

Abbildung 5.4: ICMP und ARP Traffic in Wireshark

5.3 Fazit der Tests

In allen Übungen konnte sichergestellt werden, dass Netzverkehr zwischen den Hosts möglich ist. Ping ist nur in den Übungen der virtuellen Umgebung möglich, welche in Abschnitt 5.1.1 genannt werden. In den drei Übungen auf der Laborhardware ist ping erfolgreich implementiert und getestet worden. Der Netzverkehr kann in allen Übungen mittels Python-Skript (`receive.py`), Wireshark, iPerf oder dem *Sniffing* in Scapy analysiert werden. Da Ping in der Übung 3.3.3 (Double Clone) nicht funktioniert, muss der Netzverkehr mittels `receive.py` und Wireshark ausgewertet werden. Wichtig ist bei der Auswertung in dieser Übung, dass ein mit Scapy gesendetes Paket auf den anderen beiden Hosts der Übung zweimal ankommt. Dadurch wird verifiziert, dass beide Switches in der Übung jeweils einen Klon des ursprünglichen Pakets erstellen und sowohl Original als auch Klon senden. Die Übung 3.3.5 (Resubmit2-IP) ist weniger eine praktische Übung, sondern die Zwischenstufe der anderen beiden Resubmit-Übungen (s. Abschnitte 3.3.4 und 3.3.6). Resubmit2-IP stellt die Weiterentwicklung von Resubmit zu Resubmit3-ARP dar und zeigt, welche Schritte nötig sind, um ein Programm von statischem Forwarding auf dynamisches Forwarding

umzustellen. Es ist ersichtlich, dass das Implementieren des IPv4-Headers nicht ausreichend ist, sondern auch der ARP-Header und Broadcast angelegt werden müssen, um *ICMP*-Traffic zu ermöglichen. Dies spiegelt sich in der Übung 3.3.6 (Resubmit3-ARP) wieder, da dort ARP-Header und Broadcast implementiert wurden, wodurch der mögliche Netzverkehr erweitert wurde (→ ping kann eingesetzt werden). Dies ist ein erster Schritt für das Programm, um nicht nur in Labornetzen, sondern auch in realen Netzen genutzt zu werden.

Tabellen konnten in den Übungen 3.3.5 und 3.3.6 der virtuellen Umgebung sowie den Übungen 3.3.2 und 3.3.3 auf der Laborhardware erfolgreich angelegt, konfiguriert und eingesetzt werden.

Das 25GBit/s Interface des PCs wird bisher nur für die Analyse des Netzverkehrs und vor allem für die Verifikation des *Mirrorings* genutzt. In Zukunft soll dieses Interface auch für die Erzeugung von Paketen und Geschwindigkeitstests verwendet werden.

Kapitel 6: Fazit und Ausblick

Die Arbeit stellt eine Zusammenfassung der wichtigsten Bestandteile von *P4* dar, welche für einen Einstieg in die Programmiersprache benötigt werden. Dies sind vor allem die verschiedenen *Kontrollblöcke*, wie diese genutzt werden, *was* sie beinhalten und *welche* Unterschiede zwischen *Behavioral Model Version 2* (BMv2) und *Tofino Native Architecture* (TNA) existieren.

Für mich und auch andere Mitglieder des Forschungsprojektes ist es wichtig zu verinnerlichen, dass wir ein P4-Programm vollständig nach unseren Wünschen gestalten können und das Programm von äußeren Einflüssen wie Protokollen oder Best-Practice-Ansätzen prinzipiell unabhängig ist. Es müssen keine Standards implementiert werden und man muss sich auch nicht zwingend an Richtlinien orientieren, welche in normalen Netzen gültig sind. Dies ist eine Folge des Ziels *Protokollunabhängigkeit* (s. Abschnitt 2.2). Trotzdem ist es mit P4 auch möglich ein vollständig RFC-konformes *Target* zu programmieren, welches in bestehende Netze integriert werden kann (s. Übungen des P4-Language-Consortiums in der VM).

Bei den Möglichkeiten der Programmiersprache ist vor allem der eigene Entwurf von Protokollheadern sehr interessant, da Header schnell implementiert und mit den in dieser Arbeit genannten Tools getestet werden können. Dieser eigene Entwurf stellt für mich eine noch nie dagewesene Freiheit in der Netzwerktechnik dar, da man beim Entwurf nicht von Institutionen wie Hardwareherstellern und Standardisierungsorganisationen (z.B. *Internet Engineering Task Force* (IETF)) abhängig ist. Dadurch sind beim Entwurf keine Grenzen gesetzt. Trotzdem ist eine Zusammenarbeit mit diesen Organisationen beabsichtigt, sobald eigene Header ausreichend getestet wurden und einen Mehrwert für die Netzwerktechnik darstellen.

Durch die Nutzung der Programmiersprache begann ich mich mit dem detaillierten Aufbau von wichtigen Netzwerkprotokollen (IPv4, ARP, ICMP) zu beschäftigen. Dies resultiert aus der Tatsache, dass gewünschte Header zu Beginn des Programms angelegt werden müssen. Jedes Bit im Feld eines Protokoll-Headers muss exakt wie im dazugehörigen Standard (z.B. RFCs) definiert werden (→ Reihenfolge und Länge dieser Felder)(s. IPv4-Header in Abschnitt 2.4). Die Notwendigkeit Header einzuarbeiten folgt ebenfalls aus dem Ziel *Protokollunabhängigkeit*, da standardmäßig keine Netzwerkprotokolle in P4 implementiert sind. In meiner Zeit als Netzwerkadministrator fand ich den genauen Aufbau der Protokolle uninteressant, da die Netzwerkgeräte den Netzverkehr selbstständig weiterleiten und das Wissen darüber „*wo welches Bit im Header steht*“ für den damaligen Job nicht notwendig war. Dies hat sich zu meiner eigenen Verwunderung durch P4 geändert.

Die Übungen beginnen mit leicht verständlichen Regeln für die Weiterleitung von Paketen (s. statisches Forwarding in Übung 3.3.1 Hello World) und werden zunehmend komplexer. Die steigende Komplexität resultiert einerseits aus dem Implementieren von wichtigen Netzwerkprotokollen wie IPv4, ARP und ICMP. Andererseits ist auch das Forwarding mittels *Tabellen* dafür verantwortlich. Diese beiden Punkte sind enorm wichtig, da diese für den Betrieb von größeren Netzen und die Verbindung zu Netzen anderer Organisationen unabdingbar sind. Infolgedessen konnten einige Netzwerkprotokolle und damit verbundene Mechanismen wie *Routing* oder *Multicast/Broadcast* noch einmal dargestellt werden. Die *Matching-Kriterien ternary* und *Longest-Prefix-Match* (LPM) konnten durch die explizite Nutzung und anfängliche Fehler bei der Nutzung besser verstanden werden. Die Übungen sind zwar für den Einsatz in lokalen Netzen wie dem Labornetz geeignet, sind jedoch für öffentlichen Netzverkehr (Internet) nur bedingt nutzbar. Die Anpassung der Programme, damit der P4-Switch Internet-Traffic ohne Fehler weiterleiten kann, ist für die Zukunft gedacht. In Zuge dessen soll auch IPv6 implementiert werden.

In der Zukunft sollen Teile des Codes entfernt werden, um einen gewissen Eigenanteil von Verständnis überprüfen zu können. Dies ist in den Übungen, welche in der Virtuellen Maschine (VM) des *P4-Language-Consortiums* verfügbar sind, so konfiguriert. Zusätzlich dazu gibt es für jede Übung einen Ordner **solutions**, in welchem die Lösung der aktuellen Aufgabe gespeichert ist. Hierfür müssen auch gute Anleitungen zu den Übungen erstellt werden.

Außerdem sollen bestimmte Schritte bei der Konfiguration der *Control-Plane* des P4-Switches wie das Anlegen von Tabelleneinträgen automatisiert werden. Die Konfiguration der Switchports soll in dieser Automatisierung auch enthalten sein. Möglicherweise wäre ein Python-Skript mit Eingabeaufforderungen dafür geeignet.

Debugging des Codes und daraus folgende Erkenntnisgewinne sollen in der Zukunft dokumentiert werden. Hierbei ist zu erwähnen, dass die meisten Fehler im Code beim Kompilieren des Programms ausgegeben werden und sich in ihrer Art sehr stark unterscheiden können. Syntaxfehler wie fehlende Semikolons oder das nicht mögliche Auslesen von Daten (vgl. *Richtungen* der Parameter in Tabelle 2.3) gehören zu den gängigsten Fehlerausgaben beim Kompilieren.

Einige *Tabellen* und *Actions* müssen noch umbenannt werden, da diese aus verschiedenen Codebeispielen übernommen und zweckentfremdet wurden (z.B. `port_acl` aus Abschnitt 4.3.4). Die Tabelle `port_acl` ist nicht wie der Name vermuten lässt eine *Access Control Liste* (ACL)¹, sondern wird in den beiden *Clone*-Übungen auf der Hardware genutzt, um die Multicast-Gruppe für das *Mirroring* zu setzen. Deshalb sollte der Name der Tabelle in `set_mc_group` oder ähnlich geändert werden.

Der Tofino-Chip enthält einen internen Packet-Generator. Mit diesem können Tests unabhängig von der restlichen Laborhardware absolviert werden. Damit kann das eigentliche Ziel des Forschungsprojekts verfolgt werden und der selbst entworfene *Clone-Header* weiter angepasst, getestet und schlussendlich eingesetzt werden. Weitere Testszenarien können mit

¹<https://www.ittsystems.com/access-control-list-acl/>

dem 25GBit/s-Interface des PCs entworfen werden, um Tests mit dieser Geschwindigkeit laufen zu lassen.

Ein persönliches Anliegen ist die Veröffentlichung dieser Arbeit, um möglichst vielen Menschen den Einstieg in P4 zu erleichtern. Dies soll in Foren geschehen, welche sich mit P4 beschäftigen wie z.B. dem *Intel Connectivity Research Programm* (ICRP)². Außerdem wird die Arbeit an das Berliner Unternehmen AVM³ (→ FRITZ!Box) weitergegeben. AVM hatte bereits während meines Pflichtpraktikums (bei AVM) angefragt, ob ich Ihnen diese Arbeit zukommen lassen könne. Nachdem ich Ihnen erklärt hatte, was P4 ist, sahen Sie bereits mögliche Szenarien zum Einsatz von P4 im Unternehmen.

²<https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/connectivity-education-hub/research-program/research-papers.html>

³<https://avm.de/>

Anhang A: Aliasses auf Ubuntu-PC und/oder Server

Codeauszug A.1: Aliasses auf Ubuntu Server/PC

```
1 alias ..='cd ..'
2 alias ls='ls -lisa --color=auto'
3
4 // Kommandos zum An-/Ausschalten der Switche
5 alias pwr-off-1='curl ←
6   "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output1=0"'
7 alias pwr-off-2='curl ←
8   "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output2=0"'
9 alias pwr-off-3='curl ←
10  "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output3=0"'
11 alias pwr-off-4='curl ←
12  "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output4=0"'
13 alias pwr-off-ag5648='curl ←
14  "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output2=0"'
15 alias pwr-off-as4610='curl ←
16  "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output1=0"'
17 alias pwr-off-wedge='curl ←
18  "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output3=0"'
19
20 alias pwr-on-1='curl ←
21  "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output1=1" ,
```

```
22 alias ssh-wedge='ssh p4-user@wedge-p4'
23
24 //cdp4 1. Befehl fuer Server, 2. Befehl fuer PC
25 alias cdp4='cd ~/p4-projekt/'
26 alias cdp4='cd ~/Documents/p4-projekt/'
27
28 // nur auf Server -> Kommandos zum Pingen von Interfaces blau, rot oder gruen
29 alias blueping='sudo ip netns exec blue ping '
30 alias greenping='sudo ip netns exec green ping '
31 alias redping='sudo ip netns exec red ping '
32
33 // nur auf PC -> SSH-Login auf Server
34 alias ssh-liam='ssh marcel@liam'
```

Anhang B: /etc/hosts auf Ubuntu-Server

Codeauszug B.1: /etc/hosts auf Ubuntu-Server

```
1 # P4 LAN 172.16.1.0/24
2 172.16.1.1 cisco-router
3 172.16.1.2 ip-pwr
4 172.16.1.5 ag5648
5 172.16.1.8 as4610
6 172.16.1.10 wedgecore-bmc wedge-bmc
7 172.16.1.11 wedgecore-p4 wedge-p4 wedge-studio
8 172.16.1.12 p4-ubuntu-server p4-ubuntu-srv p4-srv liam
9 172.16.1.248 ipv6-p4-asw01 p4-asw01 asw01
10 172.16.1.249 ipv6-p4-asw02 p4-asw02 asw02
11 172.16.1.250 ipv6-p4-asw03 p4-asw03 asw03
```


Anhang C: Konfiguration Switchports auf Wedge 100BF-32X

Codeauszug C.1: Vollständige Konfiguration der Switchports des Wedge 100BF-32X

```
1 // Hinzufuegen Ports mit Geschwindigkeit und FEC
2 // Port 1 = Connection mit AS4610
3 port-add 1/0 10G NONE
4 port-add 1/1 10G NONE
5 port-add 1/2 10G NONE
6 port-add 1/3 10G NONE
7
8 // Port 11 = Connection zum Test-PC via 25Gbit/s Multimode-Fiber
9 port-add 11/0 25G NONE
10 port-add 11/1 25G NONE
11 port-add 11/2 25G NONE
12 port-add 11/3 25G NONE
13
14 // Auto-Negotiation
15 an-set 1/0 0
16 an-set 1/1 2
17 an-set 1/2 2
18 an-set 1/3 2
19
20 an-set 11/0 2
21 an-set 11/1 2
22 an-set 11/2 2
23 an-set 11/3 2
24
25 //Aktivieren der physischen Ports
26 port-enb 1/-
27 port-enb 11/-
```


Anhang D: vollständige IP-Netns-Konfiguration des Servers

```
1 // Erstellung Namespace
2 sudo ip netns add blue
3 sudo ip netns add red
4 sudo ip netns add green
5
6 // Aktivieren der physischen Interfaces
7 sudo ifconfig ens2f0 up
8 sudo ifconfig ens2f1 up
9 sudo ifconfig ens5f2 up
10
11 // Hinzufuegen der Interfaces zu Namespace
12 sudo ip link set dev ens2f0 netns blue
13 sudo ip link set dev ens2f1 netns red
14 sudo ip link set dev ens5f2 netns green
15
16 // Konfiguration IP-Adressen + Aktivieren Interfaces + Loopbacks
17 sudo ip netns exec blue ifconfig ens2f0 10.1.1.1/24 up
18 sudo ip netns exec blue ifconfig lo up
19 sudo ip netns exec red ifconfig ens2f1 10.1.1.2/24 up
20 sudo ip netns exec red ifconfig lo up
21 sudo ip netns exec green ifconfig ens5f2 10.1.1.3/24 up
22 sudo ip netns exec green ifconfig lo up
23
24 // Ausgabe der Namespace-Interface-Konfiguration
25 sudo ip netns exec blue ifconfig
26 sudo ip netns exec red ifconfig
27 sudo ip netns exec green ifconfig
```


Anhang E: Konfiguration AS4610

Codeauszug E.1: Konfiguration AS4610

```
1 accton-as4610:/etc/systemd/network$ cat 10-swbridge.netdev
2 [NetDev]
3 Name=swbridge
4 Kind=bridge
5
6 [Bridge]
7 VLANFiltering=1
8 DefaultPVID=none
9 accton-as4610:/etc/systemd/network$ cat 10-swbridge.network
10 [Match]
11 Name=swbridge
12
13 [BridgeVLAN]
14 VLAN=10
15 VLAN=20
16 VLAN=30
17 accton-as4610:/etc/systemd/network$ cat 20-port1+49.network
18 [Match]
19 Name=port1 port49
20
21 [Network]
22 Bridge=swbridge
23
24 [BridgeVLAN]
25 PVID=10
26 EgressUntagged=10
27 accton-as4610:/etc/systemd/network$ cat 20-port2+50.network
28 [Match]
29 Name=port2 port50
30
31 [Network]
32 Bridge=swbridge
33
34 [BridgeVLAN]
35 PVID=20
36 EgressUntagged=20
37 accton-as4610:/etc/systemd/network$ cat 30-port3+51.network
```

```
38 [Match]
39 Name=port3 port4 port51
40
41 [Network]
42 Bridge=swbridge
43
44 [BridgeVLAN]
45 PVID=30
46 EgressUntagged=30
47 accton-as4610:/etc/systemd/network$ cat 70-enp.network
48
49 [Match]
50 # You can also use wildcards. Maybe you want enable dhcp
51 # an all eth* NICs
52 Name=enp*
53 [Network]
54 #DHCP=v4
55 # static IP
56 # 192.168.100.2 netmask 255.255.255.0
57 Address=172.16.1.8/24
58 Gateway=172.16.1.1
59 DNS=141.45.2.100 141.45.3.100
60 accton-as4610:/etc/systemd/network$ cat 90-enp.link
61 [Match]
62 Path=*platform-18022000.ethernet*
63 [Link]
64 Name=enp0
```

Anhang F: Checksum berechnen

Codeauszug F.1: Checksum-Berechnung im Ingress Deparser

```
1  hdr.ipv4.hdr_checksum = ipv4_checksum.update({  
2      hdr.ipv4.version,  
3      hdr.ipv4.ihl,  
4      hdr.ipv4.diffserv,  
5      hdr.ipv4.total_len,  
6      hdr.ipv4.identification,  
7      hdr.ipv4.flags,  
8      hdr.ipv4.frag_offset,  
9      hdr.ipv4.ttl,  
10     hdr.ipv4.protocol,  
11     hdr.ipv4.src_addr,  
12     hdr.ipv4.dst_addr,  
13     hdr.ipv4_options.data  
14 });
```


Anhang G: Python-Skript für Tabelleneinträge aus Simple-Clone

```
1 from ipaddress import ip_address
2 #from ipaddress import ip_network # seems not to work, maybe because this ←
3 # creates another type of object (address vs network)
4 p4 = bfrt.tofino_simple_clone.pipe
5 #
6 # Program the default topology
7 # sometimes: put ".push()" after command
8
9 # Set Mirror Session to 1 for Ingress Ports 134-135 (QSFP1-3 und 1-4)
10 p4.Ingress.port_acl.add_with_acl_mirror(ingress_port=134, ←
11     ingress_port_mask=510, mirror_session=1)
12
13 # Set for clone-id=1 with action=normal: enable session, direction=BOTH ←
14 # (in+egress), egress_port=40, validate port
15 bfrt.mirror.cfg.add_with_normal(sid=1, session_enable=True, ←
16     direction='BOTH', ucast_egress_port=40, ucast_egress_port_valid=True)
17 #
18 p4.Egress.mirror_dest.add_with_just_send(ing_mirrored_mask=0, ←
19     egr_mirrored_mask=0, mirror_session=1)
20
21 bfrt.complete_operations()
22
23 # Final programming
24 print("""
25 **** PROGRAMMING RESULTS ****
26 """)
27 print ("\nTable Ingress Port ACL:")
28 p4.Ingress.port_acl.dump(table=True)
29 print ("\nTable Egress Mirror Destination:")
30 p4.Egress.mirror_dest.dump(table=True)
31 print ("\nTable mirror.cfg:")
```

```
30 | bfprt.mirror.cfg.dump(table=True)
```

Anhang H: Scapy File

```
1 // MBeausencourt
2 //This is a datagram class for a "Clone Datagram", which is designed for ←
3 // use with Scapy
4
5 //Clone-Header for Scapy -> copy paste
6 class Clone(Packet):
7     """This is header v0.1 for a Clone Datagram for P4 testing. """
8     name = "CloneDatagram"
9     fields_desc=[BitField("numberOfClones",0,64),
10                 BitField("cloneCounter",0,64),
11                 BitField("secureString",42,32),
12                 BitField("receiverOption",0,8),
13                 BitField("payloadLength",0,8),
14                 BitField("version", 0,8),
15                 BitField("free",0,8),
16                 BitField("receiverIP", 0,32),
17                 BitField("receiverMask",0,32),
18                 BitField("standardString",1234,64)]
19
20 // Header Version of Thomas
21 class Clone(Packet):
22     """This is header v0.1 for a Clone Datagram for P4 testing. """
23     name = "CloneDatagram"
24     fields_desc=[LongField("numberOfClones",0),
25                 LongField("cloneCounter",0),
26                 StrFixedLenField("secureString",42,4), # 4 Bytes
27                 ByteField("receiverOption",0),
28                 ByteField("payloadLength",0),
29                 ByteField("version", 0),
30                 ByteField("free",0),
31                 IPField("receiverIP", "10.0.0.1"),
32                 IPField("receiverMask","255.255.255.0"),
33                 StrFixedLenField("standardString",1234,8)]
34
35
36 //inside Scapy --> building headers and combining them into a packet
37 ethernet_header=Ether(type=0x800, dst="ff:ff:ff:ff:ff:ff")
```

```
38 ethernet_header=Ether(type=0x800, dst="08:00:00:00:02:22")
39 --> entweder broadcast (1) oder unicast (2) + no leading ZEROS in ETH
40 ip_header= IP(dst="10.0.1.2", proto=253,flags="DF")
41 clone_header = Clone(numberOfClones=10)
42
43
44 // Header zusammenfuegen :)
45 packet = ethernet_header / ip_header / clone_header
46
47 // packet senden
48 send(packet)
49 sendp(packet, iface="eth0")
50 // Show Befehle
51 packet.show()
52 packet.show2()
53 clone_header.show2()
54 //bind_layers(IP, Clone, protocol=253)
```

Anhang I: IPv4-Adressen im Labornetz

Gerät	Interface (Kontext)	IPv4-Adresse	Link-Speed
Ubuntu-Server	ens2f0 (blue)	10.1.1.1	1Gbit/s
	ens2f1 (red)	10.1.1.2	
	ens2f2 (green)	10.1.1.3	
Ubuntu-PC	enp1s0f1np1 (-)	10.1.1.5	25Gbit/s

Tabelle I.1: IPv4-Adressen im Labor-Test-Netz

Anhang J: Programmdateien als .zip

Die für diese Arbeit geschriebenen Programme und zugehörigen Dateien aus dem Git-Repository sind als separater .zip-Ordner (`p4-projekt-main.zip`) angehängt.

Abkürzungsverzeichnis

AN	Auto Negotiation
ASIC	Application-specific Integrated Circuit
BC	Broadcast
bf-sde	Barefoot Software Development Environment
BFRT	Barefoot Runtime
bfshell	Barefoot Shell
BMv2	Behavioral Model Version 2
C	Programmiersprache C
CE2E	Clone Egress to Egress
CI2E	Clone Ingress to Egress
CLI	Command Line Interface
CPU	Central Processing Unit
FEC	Forward Error Correction
FPGA	Field-programmable Gate Array
HTTPS	Hypertext Transfer Protocol Secure
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
LTS	Long Term Support
LWL	Lichtwellenleiter - Glasfaser
MC	Multicast
NPU	Network Processing Unit - Analogie zu CPU
OS	Operating System bzw. Betriebssystem
PRE	Packet Replication Engine
RAM	Random Access Memory
RID	Replication-ID
RJ45	Registered Jack 45 - "LAN-Kabel"
SDN	Software-Defined Networking

Abkürzungsverzeichnis

SFP	Small Form-factor Plugable
SRAM	Static random-access memory
SSH	Secure Shell
SSH	Secure-Shell
TCAM	Ternary Content-addressable memory
TCP	Transmission Control Protocol
TM	Traffic Manager
TNA	Tofino Native Architecture
TTL	Time to Live
UDP	User Datagram Protocol
VM	Virtuelle Maschine
VPN	Virtual Private Network

Abbildungsverzeichnis

2.1	Änderungen von $P4_{14}$ zu $P4_{16}$ [P4 Lang. Spec. Figure 3]	4
2.2	Schnittstellen eines P4-Programms[P4 Lang. Spec. Figure 12]	5
2.3	Packet-Flow in P4 (Quelle: Prof. Dr. Thomas Scheffler, HTW Berlin)	6
2.4	Paketfluss bei verschiedenen Arten der Vervielfältigung	17
3.1	VM-Settings beim Import in VirtualBox	24
3.2	VM nach Import in VirtualBox	25
3.3	Einrichten des Git-Repository in der VM	26
3.4	Ordnerstruktur des Projekt-Repos	27
3.5	Netzwerktopologie zu Hello_World	28
3.6	Netzwerktopologie zu Simple-Clone	29
3.7	Netzwerktopologie zu Double-Clone	31
3.8	Netzwerktopologie zu Resubmit	32
3.9	Netzwerktopologie zu Resubmit2-IP	34
3.10	Netzwerktopologie zu Resubmit3-ARP	38
3.11	Starten der Übung	45
3.12	Bildschirmausgabe der nützlichen Mininet-Befehle	46
3.13	Output <code>s1.log</code> aus Double-Clone	46
4.1	Ordnerstruktur tofino-ports	49
4.2	Netzwerktopologie im Labor	50
4.3	Ausgabe Portkonfiguration	54
5.1	Pings in VM	62
5.2	Python-Tests in der VM	63
5.3	Geklonter Netzverkehr auf 25GBit/s Interface des PCs zwischen <code>blue</code> und <code>red</code> mit iPerf	67
5.4	ICMP und ARP Traffic in Wireshark	67

Tabellenverzeichnis

2.1	P4-Datentypen [s. [P4 Lang. Spec. Abschnitt 7]]	8
2.2	Auflistung Parameter im Parser	11
2.3	Richtung der Parameter	11
2.4	Vordefinierte Actions	14
2.5	Match-Action Kriterien	14
3.1	Verfügbarkeit Virtualisierungssoftware auf OS	23
3.2	Forwarding-Regel für Hello_World	28
3.3	Weitere Übungen aus dem Git-Repo	40
3.4	Files für Mininet Konfiguration und deren Nutzen	40
3.5	Nützliche Befehle in Mininet	45
4.1	Tastaturbefehle zum Beenden des P4-Programms	55
4.2	Portkonfig AS4610 und korrespondierende Ports	56
I.1	IPv4-Adressen im Labor-Test-Netz	89

Quelltextverzeichnis

2.1	Aufbau Kontrollblöcke in BMv2	7
2.2	Aufbau Kontrollblöcke in TNA	7
2.3	Parsercode aus Resubmit3-ARP	9
2.4	Einfacher BMv2 Parser	10
2.5	Einfacher TNA Parser	11
2.6	Einfache BMv2 Match-Action	12
2.7	Einfache TNA Match-Action	12
2.8	Zwei Actions	13
2.9	Tabelle ipv4_host und Ausführen der Tabelle	14
2.10	Ausgabe Info und Ausgabe eines Eintrags mittels dump()	15
2.11	IPv4 Header in P4	18
2.12	Erstellung eines eigenen Typs für die IPv4-Adresse	18
2.13	Clone-Header Implementation im Code	19
2.14	Code für Clone-Pakete	20
3.1	CLI-Befehle für die Einrichtung von Git in der VM	26
3.2	Forwarding-Code für Hello_World	28
3.3	Leerer Egress-Kontrollblock	28
3.4	Identifier für Vervielfältigung	29
3.5	Forwarding-Code für Simple-Clone	30
3.6	Forwarding-Code für Double-Clone	30
3.7	Typen der Vervielfältigung	31
3.8	Forwarding-Code für Resubmit	32
3.9	Ethernet-Header in P4	33
3.10	Header-Struct	34
3.11	Ethertypes in P4	34
3.12	State-Machine des Parsers für Resubmit2-IP	34
3.13	Ingress-Processing in Resubmit2_IP	36
3.14	Checksum Computation in Resubmit2_IP	37
3.15	Deparser in Resubmit2_IP	37
3.16	Actions für Multicast/Broadcast/ARP	38
3.17	Ingress-Processing in Resubmit3_ARP	39
3.18	topology.json aus Double-Clone	41
3.19	s1-commands.txt aus Resubmit3_ARP	42
3.20	s1-runtime.json aus Resubmit3_ARP	42

3.21 Anlegen von Tabelleneinträgen auf einem Switch in Mininet	43
3.22 Übungsspezifisches Makefile	44
4.1 Treiber und Umgebungsvariable in .profile	51
4.2 Vollständige Befehle für Start und Kompilieren des Programms	52
4.3 Aliasses für Kompilieren und Start	52
4.4 Port-Konfig in BFShell	53
4.5 Konfigurationsbeispiel eines Switchports	53
4.6 Befehle für Bearbeitung von Tabelleneinträgen	54
4.7 Konfiguration Namespace blue auf Server	55
4.8 Ausführen eines Python-Skripts zum Anlegen von Tabelleneinträgen	58
4.9 Auszug aus Python-Skript für Simple-Clone	58
4.10 Vergleich Übung Hello_World: Setzen Egress-Port in BMv2 und TNA	59
5.1 iPerf-Server Konfiguration und Output	64
5.2 iPerf-Client Konfiguration und Output	64
5.3 Erzeugen der einzelnen Header und Zusammenfügen dieser	65
5.4 Befehle zum Senden von Paketen	66
A.1 Aliasses auf Ubuntu Server/PC	73
B.1 /etc/hosts auf Ubuntu-Server	75
C.1 Vollständige Konfiguration der Switchports des Wedge 100BF-32X	77
E.1 Konfiguration AS4610	81
F.1 Checksum-Berechnung im Ingress Deparser	83

Literaturverzeichnis

- [BMv2] P4 Language Consortium. *Behavioral Model*. Jan. 2023. URL: <https://github.com/p4lang/behavioral-model> (besucht am 26.01.2023).
- [Bos14] Pat Bosshart u. a. „P4: Programming Protocol-Independent Packet Processors“. In: *SIGCOMM Comput. Commun. Rev.* 44.3 (Juli 2014), S. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: <https://doi.org/10.1145/2656877.2656890>.
- [Ethertype-Numbers] Internet Assigned Number Authority (IANA). *IEEE 802 Numbers*. 2022. URL: <https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml> (besucht am 16.01.2023).
- [IP-NetNS] E. Biedermann. *ip-netns(8)*. Dez. 2022. URL: <https://www.man7.org/linux/man-pages/man8/ip-netns.8.html> (besucht am 23.01.2023).
- [IP-Proto-Nr.] Internet Assigned Number Authority (IANA). *Protocol Numbers*. 2022. URL: <https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml> (besucht am 09.01.2023).
- [MC-Replication-ID] V. Gurevich. *question about multicast #22*. März 2017. URL: <https://github.com/p4lang/tutorials/issues/22#issuecomment-289075465> (besucht am 19.01.2023).
- [P4 Lang. Spec.] *P416 Language Specification*. Version 1.2.3. P4 Language Consortium, Nov. 2022. URL: <https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html#sec-overview> (besucht am 07.01.2023).
- [P4 Runtime Spec.] *P4Runtime Specification*. Version 1.3.0. P4.org API Working Group, Juli 2021. URL: <https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html> (besucht am 06.02.2023).
- [PSA] *P416 Portable Switch Architecture*. P4 Language Consortium, Apr. 2021. URL: <https://p4.org/p4-spec/docs/PSA.html> (besucht am 13.01.2023).
- [RFC1112] S. Deering. *Host extensions for IP multicasting*. RFC 1112. Internet Engineering Task Force, Aug. 1989. URL: <https://www.rfc-editor.org/rfc/rfc1112.txt>.
- [RFC3692] T. Narten. *Assigning Experimental and Testing Numbers Considered Useful*. RFC 3692. Internet Engineering Task Force, Jan. 2004. URL: <https://www.rfc-editor.org/rfc/rfc3692.txt>.

- [RFC791] J. Postel. *Internet Protocol*. RFC 791. Internet Engineering Task Force, Sep. 1981. URL: <https://www.rfc-editor.org/rfc/rfc791.txt>.
- [RFC792] J. Postel. *Internet Control Message Protocol*. RFC 792. Internet Engineering Task Force, Sep. 1981. URL: <https://www.rfc-editor.org/rfc/rfc792.txt>.
- [RFC826] D. Plummer. *An Ethernet Address Resolution Protocol*. RFC 826. Internet Engineering Task Force, Nov. 1982. URL: <https://www.rfc-editor.org/rfc/rfc826.txt>.
- [TNA] *PUBLIC Tofino Native Architecture*. Version 1.2.3. Intel, Nov. 2022. URL: https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf (besucht am 09.01.2023).

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 09.02.2023



Marcel Beausencourt