Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Building a training platform for the programming language P4

Bachelor Thesis

by

Marcel Beausencourt

Matriculation number: 573019

Deparment 1 – Energy and Information –
at Hochschule für Technik und Wirtschaft Berlin (University of Applied Sciences)

for the attainment of the academic degree
Bachelor of Engineering (B. Eng.)
in the course of studies
Information and Communication Engineering

Submission date: 09.02.2023
Translation finished: 11.09.2023
Version: 1.0

First examiner:	Prof. Dr. Thomas Scheffler
Second examiner:	Prof. Dr. Markus Nölle

Table of contents

Abstract

This is the english translated version of my german Bachelor Thesis "Aufbau einer Lernplattform zur Programmiersprache P4".

This bachelor thesis should serve as a basis for an easier introduction to the programming language P4. Processed knowledge gains and special features of the language, which have been collected in the course of the associated research project, are presented, explained and deepened by means of self-created, practical exercises.

These exercises can be worked on both on real hardware and within a *Virtual Machine* (VM). The hardware and other equipment are available in the laboratory *Transmission Technology* (HTW Berlin Campus Wilhelminenhof G520). The VM can be used in the laboratory, but also on your own computer. Access to the laboratory hardware is also possible via the *Virtual Private Network* (VPN) of the university.

The setup and configuration of the VM should also be presented in this work. In addition, a guideline for working with the hardware in the laboratory is to be created.

The six self-created exercises work fully for the *Behavioral Model Version 2* (BMv2)[1]. Three selected exercises were ported and tested in the course of the work for use on a real P4 switch; the remaining exercises should follow.

Please feel free to contact me for any suggestions to change or whether it helped you with an easier start into P4.

[1] P4 architecture of the P4 Language Consortium

Chapter 1: Introduction

1.1 Background and motivation

As part of the research project[1] I have been working intensively since March 2022 with the programming language "P4" (**P**rogramming **P**rotocol-independent **P**acket **P**rocessors). It has been developed by the P4 Language Consortium[2] since 2013 and emerged from considerations about the *OpenFlow* network protocol. With the help of this imperative programming language, the *Data-Plane* of network devices such as switches and routers or the *Network Processing Unit* (NPU) built into them can be configured. The *data plane* is the hardware layer responsible for sending and receiving network traffic. The counterpart to the *data plane* is the *control plane*. Routing protocols run in it and it is responsible for creating table entries that are used by the routing protocols. The created table entries are used by the *data plane* to forward packets.

P4 is also applicable to FPGAs and programmable ASICs. The aim of the research project is to implement a network load generator and - in connection with this - to get to know the capabilities and opportunities that P4 offers.

Although I already had very good prior knowledge of network technology, I noticed in the first few months of the research project how difficult it is to familiarize myself with the topic. Due to the fact that P4 is a relatively new programming language, I found that research on the Internet turned out to be more difficult and time-consuming than expected. Because of this, I came up with the idea of preparing my previous experiences and knowledge for interested people in such a way that they can familiarize themselves with the language faster and in a more result-oriented manner.

1.2 Task description

In this work all steps are to be explained that are necessary to be able to complete the exercises on both virtual and real hardware. For this it is important to give a basic, content-related introduction to P4 and the exercises that have already been created. Network topologies, associated protocols and the written code will be explained in more detail. Automated processes should also be mentioned and explained. The porting of the code

[1] https://www.ifaf-berlin.de/projects/nettraffic-p4/
[2] https://p4.org/

from the virtual environment to the hardware is then discussed. Above all, structural differences between *Behavioral Model Version 2* (BMv2) and *Tofino Native Architecture* (TNA) should be presented. Finally, the functionality of the network (including hosts) is checked using *Internet Control Message Protocol* (ICMP) [s. [RFC792]], Scapy[3] and other tools.

1.3 Approach

In chapter 2, essential components of the P4 architecture and - as a result - the code are first explained. The differences between virtual (BMv2) and real hardware (TNA) are already discussed before these are explained using the exercises. At the end of the chapter the possibilities of the language should be illustrated using a protocol header that I designed myself.

The six exercises already created are explained in chapter 3. These include:

- Criteria for forwarding network packets
- Implemented network protocols
- Network topologies

It also explains how the VM is installed and which steps are necessary in order to configure it according to the design of the exercises (e.g. network configuration in Mininet[4], initialization of the hosts).

Porting the code from BMv2 to TNA is considered in chapter 4. The structure of the P4 laboratory network is also presented. Then the effects (of the porting) on the program code and the configuration of the remaining hardware are worked out.

Testing the code and the nets is explained in chapter 5. This also includes the correct use of the hosts. Rudimentary checking for functionality can be done using *ICMP*. With the help of the packet manipulation program Scapy or iPerf[5] extremely precisely adapted test scenarios can be created and checked. In chapter 6 I give a summary of the work and an outlook for the future of the project.

[3] https://scapy.net/
[4] http:// mininet.org/
[5] https://iperf.fr/

Chapter 2: P4 Basics

This chapter provides information about the historical background of P4 and what impact it can have on our networks today. In addition the basic structure of a P4 program and some tips for creating the code are given. Different possibilities for the replication of network packets are briefly explained, since the duplication of packets is an important part of the research project as well as of the exercises. The end of the chapter deals with the possibilities of P4 using a self-designed protocol header. Network packets consist of their user data (*payload*) and the control data, which can come either before (then: *header*) or after (then: *footer* or *trailer*) the user data and are encapsulated in the respective layer of the OSI layer model.[1] The *user data* is the data requested by the user, such as the content of a website or a TV stream. *Header, Footer* and *Trailer* contain data from network protocols. For example Ethernet in the data link header (Layer 2) contains the *Media Access Control* (MAC) address and the *Internet Protocol* (IP) in the networking header (Layer 3) contain the source and destination IP addresses [s. [RFC791]].

2.1 Historical background

The P4 programming language has been in development since 2013. It was originally intended to facilitate the configuration of networks in conjunction with *Software Defined Networking* (SDN) and serve as an orientation for how *OpenFlow*[2] should continue to develop. At this point, *OpenFlow* did not have the desired flexibility to include additional protocol headers. Instead of importing more protocol headers into *OpenFlow* as before (within four years the number of header fields grew from 12 to 41), the consortium already pursued the approach in the first publication of P4 that future switches should have flexible mechanisms for parsing network packets (short: packets) and their forwarding. Even then, there were computer chips that were capable of this flexibility and switching at terabit speeds. [p. [Bos14]]

In order to be able to transfer data as quickly as possible so-called *Fixed-function ASICs* are installed in industrial switches and routers. The disadvantage of these is that they have to be able to process many network protocols out of the box, even if they may not be used at all. Experience has also shown that it takes a long time before new protocols are incorporated into the ASICs (∼3 years at Cisco with release cycles) and as a result it

[1] https://www.netzwerke.com/OSI-schichten-Modell.htm
[2] https://opennetworking.org/

takes even longer before they can be used on the devices since new hardware first has to be purchased.

Out of the just mentioned reasons and considerations the first paper about P4 was published on July 28, 2014. From its title "'**P4**: programming protocol-independent p acket processors"' is also where the abbreviation P4 comes from. The language has since been reworked, resulting in the current version $P4_{16}$. Since then, the older version has been called $P4_{14}$ for clear differentiation. In the course of this, the language was greatly simplified (only 40 instead of 70 keywords[3]) and many constructs were outsourced to libraries. This work deals exclusively with the current version $P4_{16}$, which is not compatible with the older version. [p. [P4 Lang. Spec. Section 3.2]]

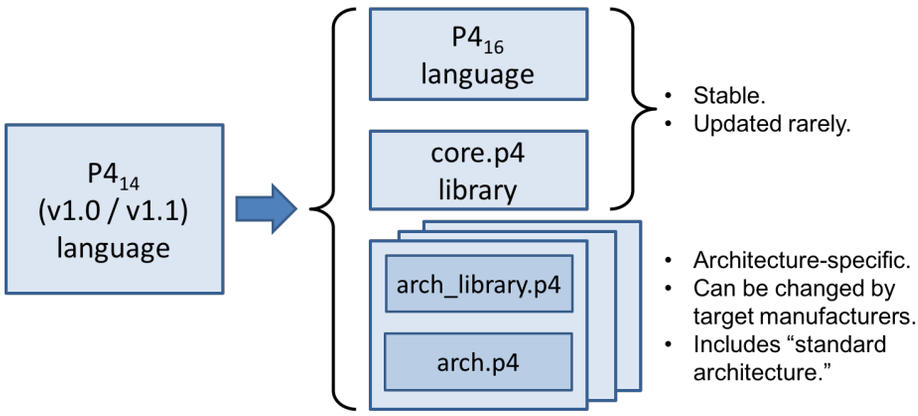


Figure 2.1: Changes from $P4_{14}$ to $P4_{16}$ [P4 Lang. Spec. Figure 3]

P4 was originally developed for use on switches, but has been extended to support various devices including routers, FPGAs and programmable ASICs. All the devices just mentioned are grouped together in this work under the term *target*. In the practical part of this work this means either the BMv2 virtual switch or the Intel Tofino[4] chip in the Edge-Core Wedge 100BF-32X.

2.2 Objectives of P4

P4 should fulfil three objectives:

1. Reconfigurability
2. Protocol independence
3. Target independence

[3] cf. C: int, double, struct, ...
[4] https://www.intel.de/content/www/de/de/products/network-io/programmable-ethernet-switch/tofino-series.html

These mean the following:

1. The behaviour of the unit, how packets are forwarded, can be changed even after the programme has been deployed.

2. The executing instance should not be bound to specific package formats, but should work based on the header data extracted by the parser and defined match action tables (see section 2.3.4 Tables).

3. A target architecture-independent description (P4 code) is to be translated by the compiler into a target architecture-dependent program [cf. [Bos14]]. Similar to the C programming language a P4 programmer should not have to deal with the hardware-specific details of the target device.

2.3 General operating principle

The P4 architecture works with *Programmable Blocks*. These include *Parser, Deparser, Ingress Control Flow* and *Egress Control Flow*. Once a packet enters the target, it is initialized with the *Intrinsic Metadata*. This is hardware-dependent information such as the port[5] where the packet was received or the *timestamp*[6]. There is also the additional possibility of adding *user-defined metadata* to the packet as it is processed by the target. These may prove useful in deciding „how to forward the packet “Ṫhe following figure 2.2 illustrates these parts of the architecture.

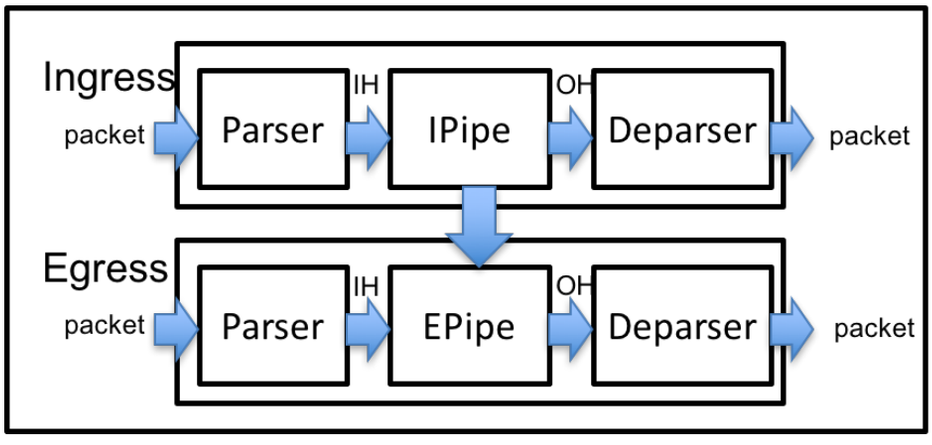


Figure 2.2: Interfaces of a p4 program[P4 Lang. Spec. Figure 12]

P4 has a C-like syntax and `.p4` as file extension. Comments are written with `//` or `/* COMMENT */`.

[5] physical network interface of the target
[6] the time at which the packet was received

2.3.1 General structure of the two architectures

The two architectures TNA (see section TNA) and BMv2 (see section BMv2) basically contain the same control blocks. Nevertheless, there are differences in the structure. Therefore, these are presented here and explained in the following points. The general path when processing the data of a packet can be seen in Figure 2.3. It is interesting to note that the *Payload or Full Packet Data* in Figure 2.3) is not processed in the *Match Action*. Only the *headers* play a role in processing and forwarding for this *control block*. In the figure, you can also see the custom *metadata* and *intrinsic metadata*. These are exchanged separately from the actual packet in the *Meta-Data-Bus* between the various *control blocks*.

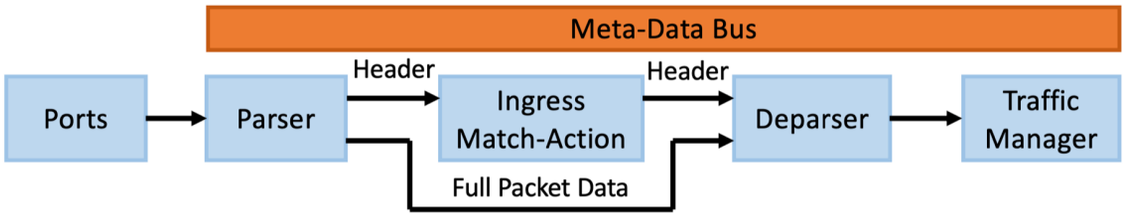


Figure 2.3: Packet flow in P4 (Source: Prof. Dr. Thomas Scheffler, HTW Berlin)

BMv2

The Behavioural Model Version 2 (BMv2) is a software-based switch of the P4 Language Consortium. The behaviour of packet processing in it is determined by the P4 programme. *BMv2* is not designed to be used in a production environment. It is intended to be used as a tool for development, testing and debugging. [see [BMv2]] BMv2 has the following six control blocks. These are used in the programme as in the code extract in this section on the target.

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

Code listing 2.1: Control blocks in BMv2

```
1 /*************************************************************************
2 ************************ S W I T C H *******************************
```

```
3 ************************************************************************/
4
5 V1Switch(
6 MyParser(),
7 MyVerifyChecksum(),
8 MyIngress(),
9 MyEgress(),
10 MyComputeChecksum(),
11 MyDeparser()
12 ) main;
```

TNA

The *Tofino Native Architecture* (TNA) is a P4 architecture developed by Intel for their P4-capable hardware. It is used e.g. on the Tofino chip. TNA also has six control blocks. These can be divided into two logical blocks: Ingress and Egress block. The two blocks are structured according to the following scheme:

1. Parser
2. Match-Action
3. Deparser

The blocks are combined in a *pipeline* and loaded into the respective pipe of the target via the command in line 11. The target can have 1-4 different or identical pipelines. These would have to be inserted in line 11 after `pipe` separated by commas (see the following code extract).

Code listing 2.2: Control blocks in TNA

```
1 /************ F I N A L P A C K A G E ****************************/
2 Pipeline(
3     IngressParser(),
4     Ingress(),
5     IngressDeparser(),
6     EgressParser(),
7     Egress(),
8     EgressDeparser()
9 ) pipe;
10
11 Switch(pipe) main;
```

Important components of the TNA are also the *Traffic Manager* (TM) and the *Packet Replication Engine* (PRE) contained therein. The *TM* stores or queues packets in the buffer until they are sent and implements the necessary commands from the code, such as setting the egress port. The *PRE* is responsible for the duplication of packets (e.g. with

multicast). [*TM* and *PRE* are not configurable from within the programme. They obtain the data they use for forwarding from the *Intrinsic Metadata* and entries of the *tables* (see section 2.3.4)

2.3.2 Important data types and keywords

The following table 2.1 lists important data types and keywords of P4. In addition to these data types, there are so-called *externs*. These are predefined functions which can e.g. create a *mirror* (s. section 4.3.3).

Name	P4-Code	Description
Boolean	`bool`	as in other programming languages: Values: true or false
Integer	`int`	32-bit Integer
Bit	`bit<x>`	Data type, which contains X Bit X must be a positive integer
Error	`error`	signals errors
Struct	`struct`	Creation of user-defined data types
Definition	`#define`	as in C: pre-procedural instruction also comparable with Alias for Linux
Type definition	`typedef`	Creating important/much-used types such as IPv4 address
Header	`header`	To define protocol headers
Constant	`const`	Constant follow by data type, name and value
Control block	`control`	all P4 control blocks except the Parser
Parser	`parser`	Declaration of the Parser
Tables	`table`	Declaration of a table
Function	`action`	Comparable with functions from other programming languages

Table 2.1: P4 data types [s. [P4 Lang. Spec. Chapter 7]]

2.3.3 Parser

The first control block of a P4 programme - the *Parser* - writes hardware-specific data (e.g. the incoming port) of the incoming packet into the *Intrinsic Metadata*. The *Parser* is to be used as a state machine. Here, all possible headers that the target can understand and process must be defined at the beginning, processed according to specified criteria and stored (as long as the packet has not yet been forwarded). If a header is parsed, this header is validated. This is important for the *control blocks* in the 2.3.4 (Match-Action and Control Flow) and 2.3.5 (Deparser) sections to know whether or not a particular header exists within the current packet.

The following parser code from the exercise 3.3.6 (Resubmit3-ARP) in the virtual environment serves as an illustration. In this exercice, headers for Ethernet and IPv4 were created and added in `struct headers`. This is followed by the code for the state machine starting with the mandatory state `start`, which initialises the *Intrinsic Metadata*. By using the keyword `transition` one instructs the parser at which `state` to proceed. This `transition` can be complemented by a `select`, making it similar to a `switch case` in C. With the command `pkt.extract(hdr.X)`[7] again extracts header data of the packet. In the `state parse_ethernet` it can be seen that the `select` condition is made based on the *Ethertype* of the Ethernet header. The *Ethertype* signals which protocol is used in the next header (Layer3). The two possibilities were defined at the beginning of the sample code. If none of the defined conditions are met, the `default` state occurs.

Code listing 2.3: Parsercode from 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 [...]
27 struct headers {
28     ethernet_t ethernet;
29     ipv4_t ipv4;
30 }
31 [...]
32     state start {
```

[7] X: placeholder for the header type e.g. „IPv4 “

```
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          }
```

`[...]` signals here and throughout the work that code has been omitted from the programme for a better overview. The complete programme code can be viewed in the attached zip folder (see appendix J).

The keyword `accept` signals the parser to end its successful execution. The counterpart to this is the keyword `reject`, which indicates a parser error. The respective behaviour of the programme in these two states depends on the architecture. The programme will then start with the next *control block*. In the parser, metadata can already be initialised. `if..else`-conditions and *Actions* (see section 2.3.4) cannot be used in the *parser*.

The basic structure of the parser in the virtual environment is as follows:

Code listing 2.4: Simple 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  }
```

Basic parser structure in TNA:

Code listing 2.5: Simple 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 }
```

Table 2.2 lists the four parameters of the Parser and shows differences in the nomenclature between both architectures.

Parameter	**Direction**	**Architecture**	
		BMv2	*TNA*
incoming paket	without direction	packet_in	
Header data	Out	headers	
Metadata	Out	metadata	
Intrinsic Metadata	Out	standard_metadata_t	ingress_intrinsic_metadata_t

Table 2.2: Listing of the Parser parameters

The „direction“ of those data mean the following:

Direction	**Description**
in	Read-only data
inout	Read and write data
out	Write-only data

Table 2.3: Direction of the parameters

Both the order in which this data is set during declaration and the direction are unchangeable in the respective block. This becomes clear once again in the following blocks. The name of the *control blocks* and *parameter* is freely selectable.

2.3.4 Match-Action and Control Flow

General information

The *Parser* is followed by the *Match-Action-*, *Control-Flow-* or in general *Processing-Block*. The last term combines the first two and all three can be used. In this block, decisions are made about the further processing of the packets (*Match-Action*). This can be a simple

discarding, setting the outgoing port or an application of a table such as the routing table. The routing table is a table within routers that the router uses to make forwarding decisions. It contains network addresses including subnet mask and outgoing port. *Control-Flow* refers to the order in which *Actions*, conditions and *tables* references are processed. [s. [Bos14, Chapter 4.6]] Below, from the 3.3.1 (Hello World) exercise, is the *match-action* code for both architectures. In the programmes, the decision to forward the packets is made based on the incoming port (if-condition: `ig_intr_md.ingress_port`). Thereupon the `egress port egress_spec` (outgoing port) is set. Port here refers to a physical interface - also called *interface* - of a *target*.

BMv2:

Code listing 2.6: Simple BMv2 Match-Action

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    apply {
        if (standard_metadata.ingress_port == 1) {
            standard_metadata.egress_spec = 2;
        }
        else if (standard_metadata.ingress_port == 2) {
            standard_metadata.egress_spec = 1;
        }
    }
}
```

TNA:

Code listing 2.7: Simple TNA Match-Action

```
control MyIngress(inout headers hdr,
                   inout metadata meta,
                   in ingress_intrinsic_metadata_t ig_intr_md,
                   in ingress_intrinsic_metadata_from_parser_t ig_prsr_md,
                  inout ingress_intrinsic_metadata_for_deparser_t ig_dprsr_md,
                  inout ingress_intrinsic_metadata_for_tm_t ig_tm_md)
{
// 134 = QSFP1-3; 135 = QSFP1-4
    apply {
        if (ig_intr_md.ingress_port == 134) {
            ig_tm_md.ucast_egress_port = 135;
        }
        else if (ig_intr_md.ingress_port == 135) {
            ig_tm_md.ucast_egress_port = 134;
        }
    }
```

```
17 }
```

It is noticeable when comparing the code that in TNA six parameters are passed to the *match-action-control-block*. The last four parameters are combined in one parameter in BMv2. In addition, parameters have changed directions (compared to the parser). The structure of each block can be found for the TNA in the file `tofino1_arch.p4`[8]. The content of any intrinsic data, such as variable name or data type, can be found in the file `tofino1_base.p4`[9]. The files are available on Barefootnetworks[10]'s Github.

Actions

Functions are called *Actions* in P4 and declared via the same keyword. It is possible to create *Actions* without and with parameters. This can be seen in the following two *Actions*. The passing parameter(s) are written directly into the code or read from a table (see the following section Tables). The `action drop` causes the target to drop the packet and not to forward it. The `action send` sets the desired egress port for the packet.

Code listing 2.8: Two Actions

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

Actions can be cascaded, local variables initialised and *if-conditions* used. P4 has some predefined *actions* [see. [Bos14, section 4.5]], which are briefly explained in table 2.4:

Name	Description
`set_field`	Set value for specific header field; masks are supported
`copy_field`	Copy field to another
`add_header`	Set header incl. fields valid
`remove_header`	Remove header incl. fields
`increment`	In- or decrement value by 1
`checksum`	Calculate header checksum

Table 2.4: Pre-defined actions

[8] https://github.com/barefootnetworks/Open-Tofino/blob/master/share/p4c/p4include/tofino1_arch.p4
[9] https://github.com/barefootnetworks/Open-Tofino/blob/master/share/p4c/p4include/tofino1_base.p4
[10] Barefoot Networks developed the Tofino chip and was later bought by Intel

Tables

Tables are the preferred way to make forwarding decisions in P4. These are declared with the keyword `table` and a freely selectable name. At least one `key` with *match criterion*, `actions`, `default action` and `size` (= size) must be specified here. They are called with `TABELLENNAME.apply()`. Table entries must be added via the control plane, which means outside of the P4 code. The `key`[11] is the criterion to be checked. It is compared with the key entries of the respective table. Any data occurring in the match action that can be read can be used as `key`.[12]. The three match criteria are explained in table 2.5.

Match-Criterion	Explanation
exact	all bits have to match exactly
lpm	Longest Prefix Match: the key with the most matching and consecutive bits will be selected[13]
ternary	with use of a mask[14]: the mask defines, which consecutive bits between value and key are compared

Table 2.5: Match-Action Criterions

Code listing 2.9: Table ipv4_host and executing the table

```
1   table ipv4_host {
2       key = { hdr.ipv4.dst_addr : exact; }
3       actions = {
4           set_nexthop;
5           @defaultonly NoAction;
6       }
7       const default_action = NoAction();
8       size = IPV4_HOST_TABLE_SIZE;
9   }
10  [...]
11  ipv4_host.apply();
```

The previously declared `Actions` are added to the corresponding section of the table; analogously, the selected `Default-Action`. The `Default-Action` occurs if no suitable entry is found in the table. The `size` specifies how many entries the respective table can store. In the example it can be seen that not necessarily an integer, but also previously defined values (`#define`) can be specified there. The first 12 lines of the following code extract show the output of the table `mirror.cfg`. `Capacity` corresponds to the previously configured `size`. `Usage` indicates the number of entries already created in the table. The specification of the table allows the compiler to decide how much memory to allocate for

[11] of the data of the package currently being processed
[12] data with direction `in` or `inout`
[13] compare to LPM routing
[14] compare to subnet mask: the number of least matching, consecutive bits

it in *Ternary Content-Addressable Memory* (TCAM) or *Static Random Access Memory* (SRAM). [s. [Bos14, Chapter 4.4]] This table was output in the *Barefoot Runtime* (BFRT) (see section 4.2.1). Information about the *key* can be found in lines 9-12. Subsequently, the first entry in the table is output together with the function parameters (incl. the set values). In line 20 it can be seen that for this *key* the *action* `normal` is executed. Further information on this section can be found in the [P4 Runtime Spec. section 9].

Code listing 2.10: Output of information and a table entry via 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     $ucast_egress_port : 0x00000028
24     $ucast_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

The *Deparser* is the counterpart of the *Parser* and is applied after the *Match-Action*. It prepares the previously extracted and modified values of the packet for sending by reassembling them. The *Deparser* has only this role in BMv2. In TNA it is also used to duplicate packets or recalculate checksums. Verification and recalculation of the checksum are created in separate control blocks in BMv2 (see section 2.3.6).

2.3.6 More blocks

This section refers to the two checksum control blocks of BMv2. In TNA, these are declared as external functions and are applicable in both ingress parser and egress deparser. The checksum is a 16-bit long field in the IPv4 header (see section 2.4). It is used to check whether a packet has been transmitted without error or whether it has been manipulated.

Checksum calculation

Before sending an IP packet, a checksum is calculated from the remaining fields in the IPv4 header. This value is then entered into the designated field of the IPv4 header. If data in the IPv4 header is changed, the checksum must be recalculated.

Checksum Verification

This is performed when a packet is received. If an error is detected, the packet is discarded. [s. [RFC791]]

2.3.7 Packet Replication

In order to be able to duplicate packets, there are various possibilities in P4. They are summarised under the term *Mirror* or *Mirroring*. These are:

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

These four mechanisms all represent different paths through which the duplicated packet must pass again before it is sent. These can be seen in the following figure 2.4:

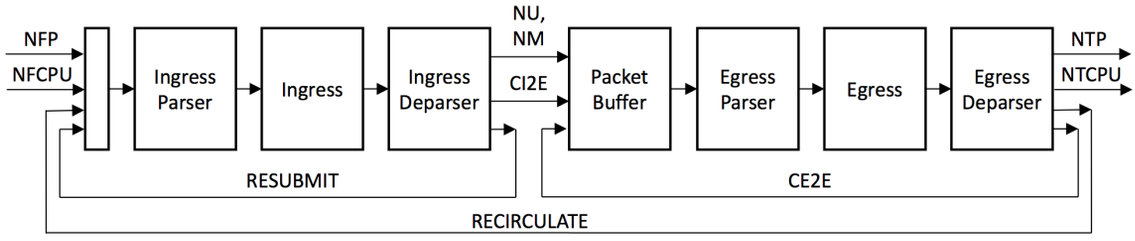
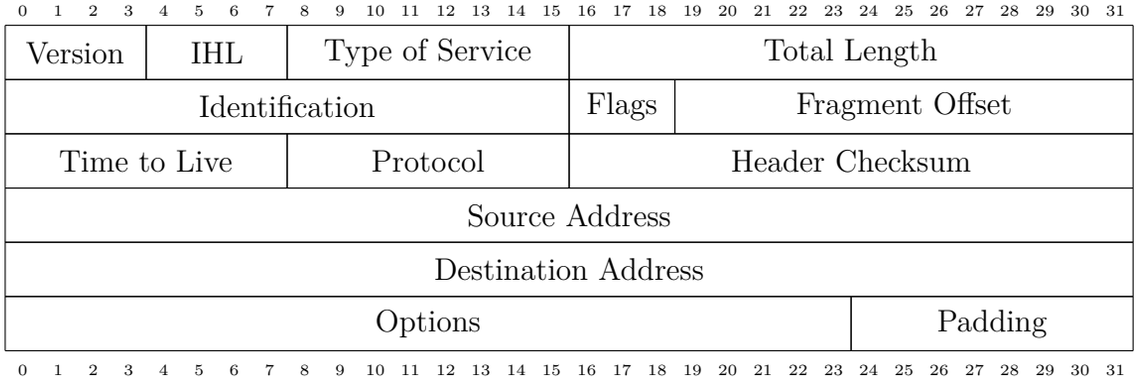


Figure 2.4: Packet flow for different types of mirroring
[s. [PSA, Figure 2]]

2.4 Capabilities of P4

Due to the goal of *2. protocol independence* (see section 2.2), no network protocols are implemented in P4 by default. These must be created in the code using the corresponding *Request for Comments* (RFC) or other standards. The IPv4 header according to [RFC791] serves as an example. This is structured as follows:



The IPv4 header is created in the programme as follows:

Code listing 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  }
```

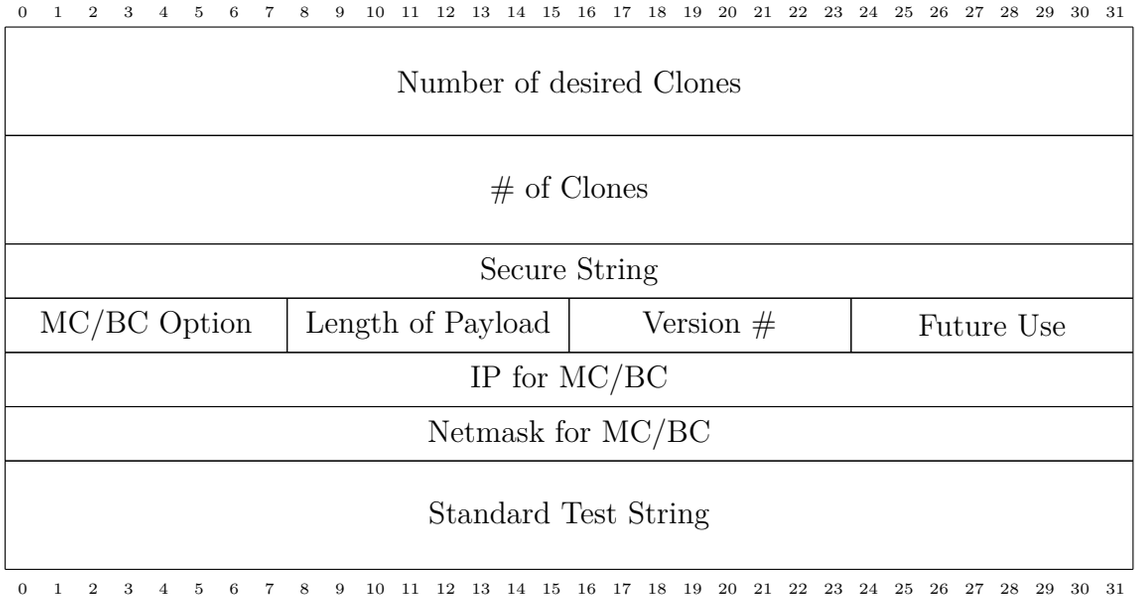
The type `ip4Addr_t` was created using `typedef` (see section **??**):

Code listing 2.12: Creation of IPv4 type

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

Padding from the IPv4 header does not have to be used here, but can be implemented. It is used to ensure that the header also has the correct length (→ multiple of 32-bit). If the header is not an integer multiple of 32-bit, missing bits with value=0 (zero) will be added to the header until it reaches the correct length. [see [RFC791]].

Since all standards have to be integrated into the programme, it is also possible to design and use headers/protocols yourself. These must be written in the same way as the IPv4 header. Afterwards, the programme must be instructed to process this new header correctly in the *control blocks*. The self-created *Clone* header from the research project serves as an example for this:



Code listing 2.13: Clone-Header Implementation in the programme

```
1   header clone_t {
2     bit<64> numberOfClones;
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 not
8     bit<8> receiverOption;
```

```
9   //identifier whether a packet should be just transmitted to the IP in ↩
        IP-Header(0), to 1 extra host(1), broadcasted(2) or multicasted (3)
10  bit<8> payloadLength;
11  //identifier how long the payload is --> payloadLength * 255 Byte (1st ↩
        idea); payloadLength = 0 --> only 64bit payload field is used
12  bit<8> version;
13  //version identifier
14  bit<8> free;
15  // future use --> maybe identifier for type of retransmission (clone, ↩
        resubmit, recicle, ...)
16  ip4Addr_t receiverIP;
17  //shows Unicast, Multicast or Broadcast address
18  bit<32> receiverMask;
19  //Netmask for receiverIP
20  bit<64> standardString;
21  //preconfigured Test String aka "Hallo ihr"
22 }
```

This header was designed for layer 4 in the OSI layer model and is based on Ethernet (layer 2) and IPv4 (layer 3). The number of times a packet is to be cloned is entered in the header field (`number of desired clones`). The following field `# of Clones` is used to check how often the package has already been duplicated. If the numbers in these two fields match, the package has been cloned often enough. After that, no more clones are created and sent. The remaining fields are only planned for the time being and are explained in the comments in the code.identifier[15] here is the field *Protocol* of the IPv4 header. The value *253* (cf. code extract 2.14, line 3) from the list of protocols in [IP-Proto-Nr.] was used to signal to the programme that a self-designed header - the *Clone-Header* - follows on layer 4. The values *253* and *254* were created specifically for experimental and testing purposes [see [RFC3692]].

Code listing 2.14: Code for Clone-Packets

```
1  [...]
2  // IP-Protocol Field
3  const bit<8> PROTOCOL_CLONE = 0xFD; // = 253 binary -> indicates a Clone ↩
       L4 datagram
4  [...]
5  state parse_ipv4 {
6         packet.extract(hdr.ipv4);
7      meta.mymeta.counter = 0;
8      transition select(hdr.ipv4.protocol){
9        PROTOCOL_CLONE: parse_clone;
10       default: accept;
11     }
12                 //transition accept;
```

[15] identifies in the Layer3 header which protocol is present in Layer4

```
13        }
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  [...]
```

As can be seen from the previous work, P4 offers the possibility to design and implement a completely self-defined processing and forwarding of packages. Hardware resources can be used more efficiently, as only required protocols, actions or similar are included and tables are configured very precisely (e.g. size of the table). Since the P4 programme can be written largely independently of the hardware and the compiler takes care of linking code and hardware, independent further development of programming language and hardware is also possible. [s. [P4 Lang. Spec. Section 3.1.]]

Chapter 3: Virtual Environment

This chapter deals with the exercises that can be completed in the virtual environment. The virtual environment is a *virtual machine* (VM). On this there are programs which can emulate networks including switches, hosts (end devices) and other devices (see section 3.4). In 3.1 the installation and in 3.2 the configuration of the VM is explained. Section 3.3 explains the six exercises including network topologies and forwarding rules. At the end of the chapter, the tools included and their use (3.5) as well as configuration (3.4) are discussed.

3.1 VM Installation

The VM is provided by the P4 Language Consortium and is based on the operating system (OS) Ubuntu[1] 20.04. The latest version of the VM can be downloaded here:
Download link
Here one of the files from the *Release VM Image link* tab should be downloaded and none of the *Development VMs*. To install them, virtualization software is needed. For this purpose, for example, Oracles[2] *VirtualBox VM Manager*[3], VMwares[4] *VMware Workstation Player*[5], Microsofts[6] *Hyper-V*[7] or even the free software *QEMU*[8] can be used. Table 3.1 gives information on which operating system the just mentioned programs run:

	Windows	**MacOS**	**Linux**
VirtualBox	√	√	√
Workstation Player	√	X	√
Hyper-V	√	X	X
QEMU	√	√	√

Table 3.1: Available VM-Software for OSs

[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/

The installation is explained using VirtualBox in Windows 7. It was carried out in the same way on Ubuntu. In VirtualBox, the VM can be imported using `\Import appliance...` or the keyboard command `Ctrl+I`. The software automatically makes settings, which can be seen in the following illustration:

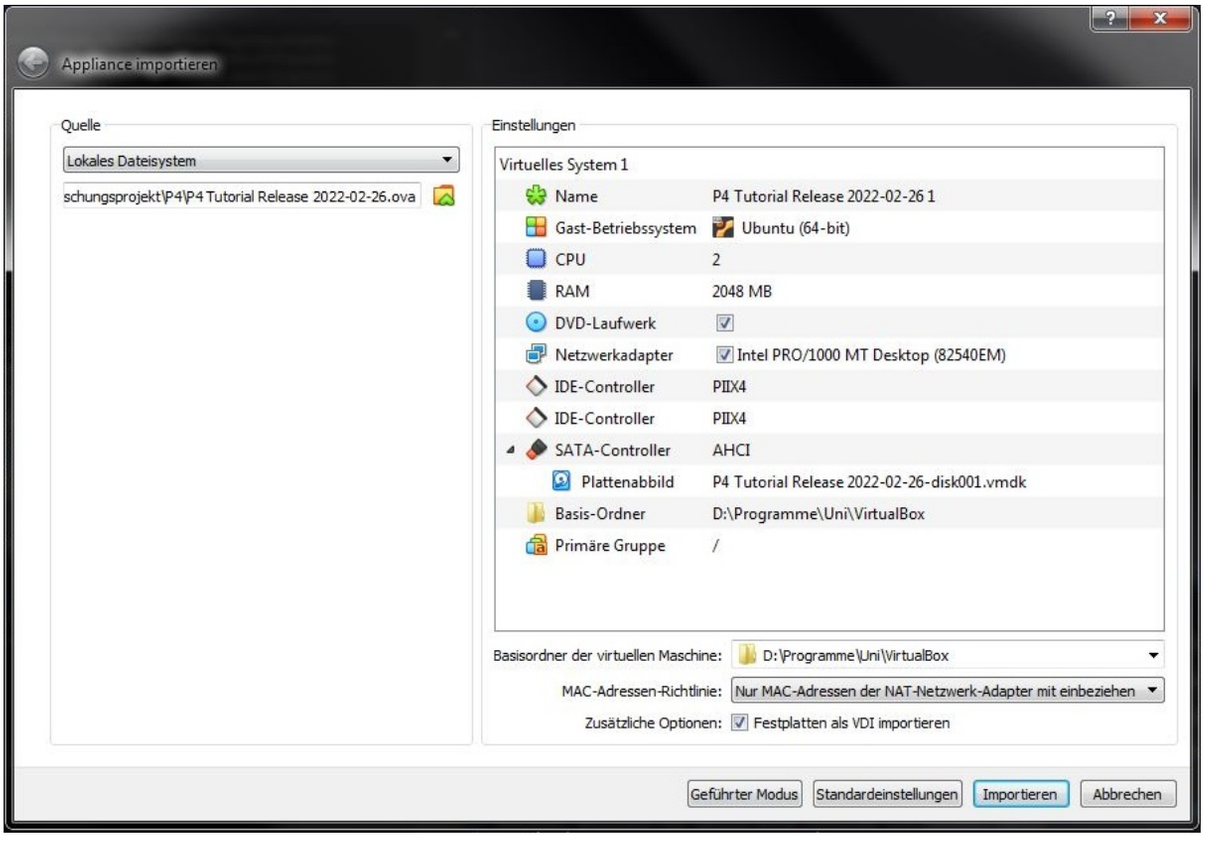


Figure 3.1: VM-Settings for import in VirtualBox

These settings can be adopted directly. The `name` is freely selectable. `CPU` and `RAM` should not be set lower, but can be set higher. The `network adapter` depends on the hardware of the device on which VirtualBox is used. `Base folder` refers to the location of the VM. This can also be adjusted. These settings can be changed again in the main menu of VirtualBox after importing (`main menu` → `change`). The VM must be switched off for this.

The VM can then be imported using the `Import` button. This may take a few minutes depending on the hardware of the host machine. If the import was successful, the VM can now be seen in the main menu of VirtualBox (cf. blue highlight in figure 3.2). The settings made in Figure 3.1 and the remaining options of the VM can also be seen here. The VM can then be started using the `Start` button.

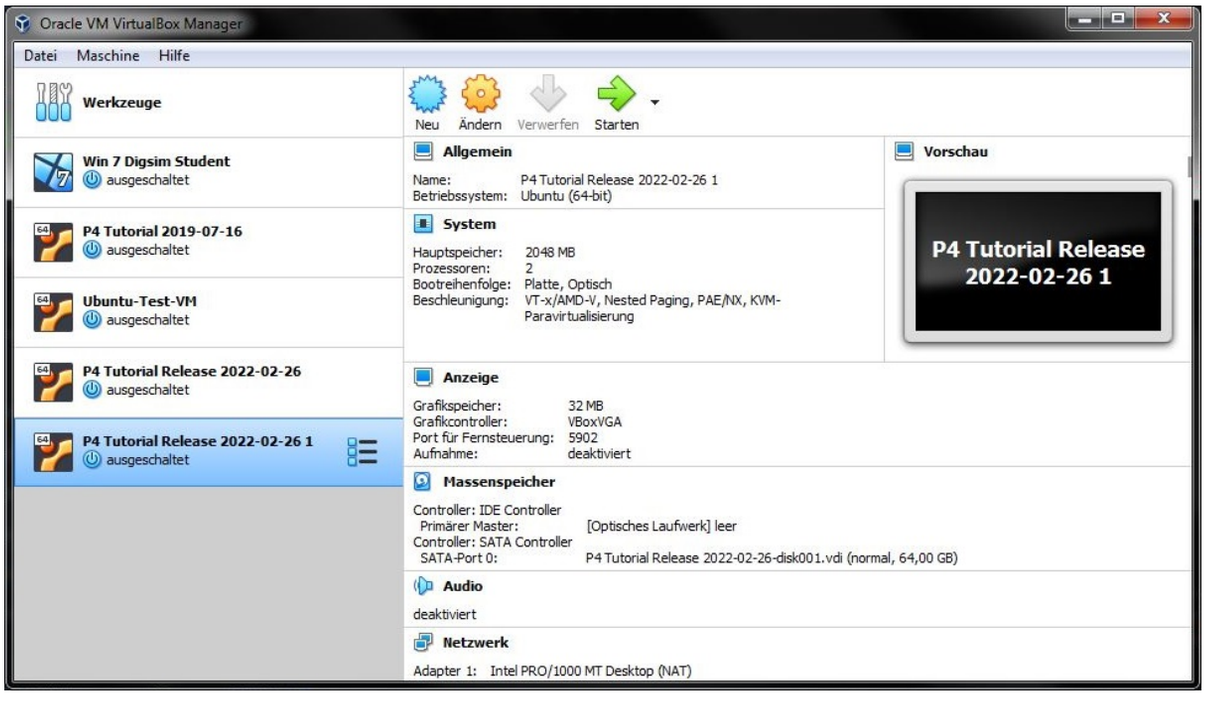


Figure 3.2: VM after VirtualBox import

3.2 VM configuration

After starting the VM, log in with the account `P4` and the password `p4`. The configuration and later work in the VM is done using *Terminal*. This is linked on the desktop. In the folder `~/tutorials` there are exercises, instructions and solutions to the exercises of the P4 Language Consortium. As this is a Git[9] repository, the current files can be downloaded via `git pull` after going to the folder via `cd ~/tutorials`. Git is pre-installed on the VM. At the beginning of the research project, we tried to familiarise ourselves with P4 using these exercises, but quickly realised that they are unsuitable for getting started with the programming language unless you already have a good understanding of P4. Therefore, these exercises are not discussed further and are mentioned for the sake of completeness. They can be completed in addition to the exercises listed later in section 3.3.

To import the exercises created for this work, it is recommended to create a folder via `mkdir FOLDERNAME` and change to it with `cd FOLDERNAME`. In the example, the folder is called `p4-htw`. The Git repository (short: repo) of the research project is cloned to this folder. The repo can be cloned to any place in the file system, as another folder is created for it (here: `p4-projekt`). After entering the account data (username and password), the repo is cloned. To do this, access must first be permitted by the research team. Then the repo folder is changed and the folder structure is output using `ls -lisa`. The repo was cloned here using *Hypertext Transfer Protocol Secure* (HTTPS). Cloning is also possible via *Secure-Shell* (SSH).

[9] https://git-scm.com/

```
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/
```

Figure 3.3: Setting up the Git-Repository inside VM

Code listing 3.1: CLI-commands for setting up the Git-repo inside the VM

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

To avoid having to enter the account data for every operation with Git, such as `git pull` (downloading the data) or `git push` (uploading the data), it is advisable to store them using `git config credential.helper store`. It should be noted that this data is stored as plain text. This data will be saved the next time it is entered. In order to be able to authenticate push operations, one's own name and mail address should also be specified using the following commands.

Code listing 3.2: User name and mail config for Git

```
1 git config --global user.name "FIRSTNAME LASTNAME"
2 git config --global user.email p4iscool@example.yeah
```

The following exercises from the repo will be explained in more detail in this chapter:

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
```

Figure 3.4: Folder structure of the Project-Repo

3.3 Exercises

This section explains the six exercises that can be done in the VM. Network topologies, implemented protocols and forwarding rules are explained and code from the respective P4 programme is shown. The rest of the configuration of the VM is explained in section 3.4.

3.3.1 Hello World

As in other programming languages, there is also a „Hello_World“ programme for P4. A screen output with this text is not to be expected here. Instead, a simple rule for forwarding the packets is implemented (see table 3.2). For this, the outgoing port (*Egress Port*) is set with the command `standard_metadata.egress_spec = 2;`. Before that it is checked on which incoming port (*Ingress Port*) the packet was received (`if (standard_metadata.ingress_port == 1)`. Headers and custom metadata are not used in the programme. The network here consists of two PCs (*H1* and *H2*) and a switch (*S1*) on which the P4 programme runs. The hosts are connected to the corresponding switch ports (H1 → P1[10]; H2 → P2) via their interface `Eth0` (see figure 3.5).

Incoming Port	Outgoing Port
1	2
2	1

Table 3.2: Forwarding rule for Hello_World

[10]= port 1

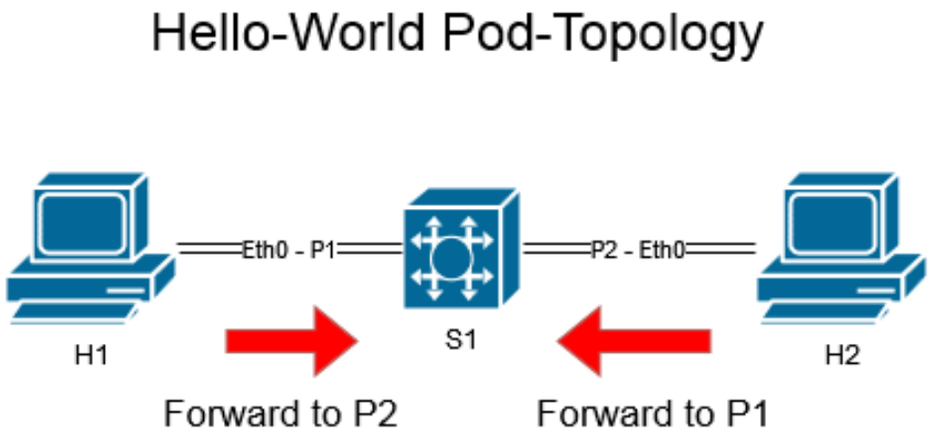


Figure 3.5: Network topology for Hello_World

The forwarding in the *match action control block* (`control MyIngress`) can be implemented as in code extract 3.2.

Code listing 3.3: Forwarding code for Hello_World

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    apply {
        if (standard_metadata.ingress_port == 1) {
            standard_metadata.egress_spec = 2;
        }
        else if (standard_metadata.ingress_port == 2) {
            standard_metadata.egress_spec = 1;
        }
    }
}
```

The remaining, unused *control blocks* must be declared and initialised with an empty `apply` statement. The following source code excerpt from the *egress-processing* illustrates this:

Code listing 3.4: Empty Egress-Controlblock

```
    control MyEgress(inout headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {
    apply { }
}
```

3.3.2 Simple Clone

The programme *Simple-Clone* is the first of the exercises to use a mechanism for duplicating or cloning packages. The network is constructed as in exercise 3.3.1. In addition, a third host *H3* has been added analogous to the first two. As soon as the packets sent by H1 arrive at the switch, they are copied 1:1 and sent via P3; i.e. cloned to H3. The original packet is sent to H2. The original (packet) can be sent by H1 as well as H2. The recipient is then the non-sending host.

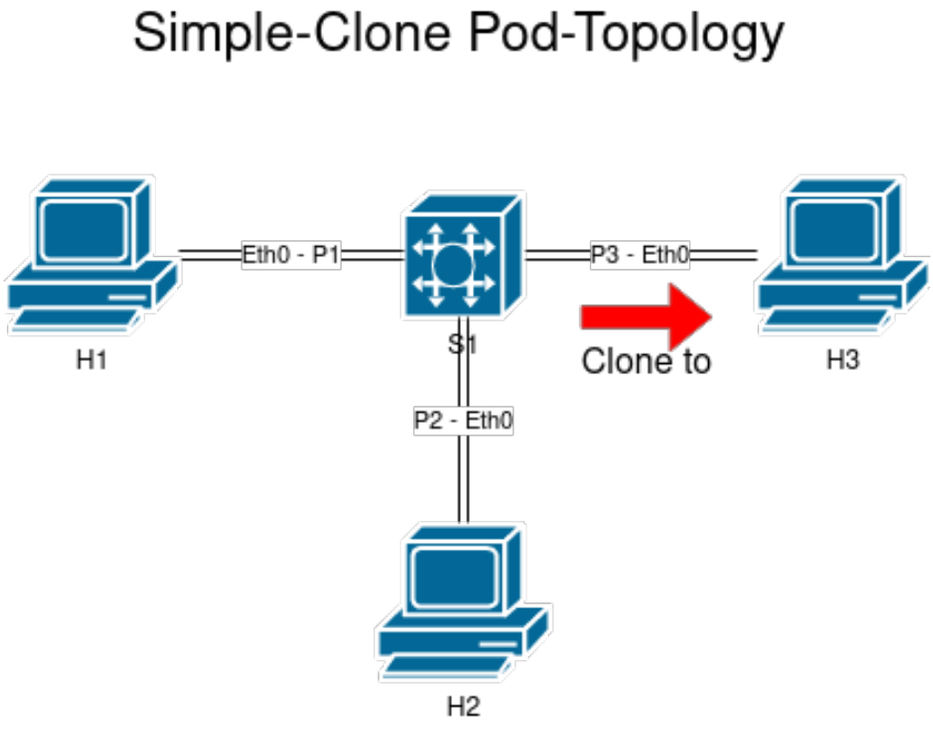


Figure 3.6: Network topology for Simple-Clone

To prepare the P4 programme for duplication, 32-bit long identifiers (ID) must be created according to the following scheme:

Code listing 3.5: Identifier for clone duplication

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

This ID is passed as a parameter in the subsequent function call and must also be created in section 3.4 (Mininet configuration) on the switch. The ID gives the switch the necessary information where to send the cloned packets. In this exercise, *Clone Ingress to Egress* (CI2E) (cf. section 2.3.7) is used. The function call thus takes place in *Ingress-Processing*. In *Behavioral Model Version 2* (BMv2) there is the function `clone_preserving_field_list(CLONETYPE, SESSION_ID, ownMETADATA_Field)` for this purpose. The first parameter specifies which type of cloning is used (here: *CI2E*). The second is the *Clone-Session-ID* just mentioned. With the third parameter it is possible to

process own metadata. The specified number signals which field of the own metadata is to be used. The forwarding for this exercise looks like this:

Code listing 3.6: Forwarding code for 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 this exercise, certain network packets are cloned twice. For this purpose, another P4 switch (S2) was added to the network from section 3.3.2. S2 clones packets from *H1* resp. *P1* to *P4*[11]. Packets received on port 4 are cloned to *P2*. Headers are not defined. Custom metadata is created but not used here. *Clone-Session-IDs* are declared as in exercise 3.3.2 (Simple Clone).

The forwarding was implemented as follows:

Code listing 3.7: Forwarding code for 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;
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 }
```

[11] here: Port 4

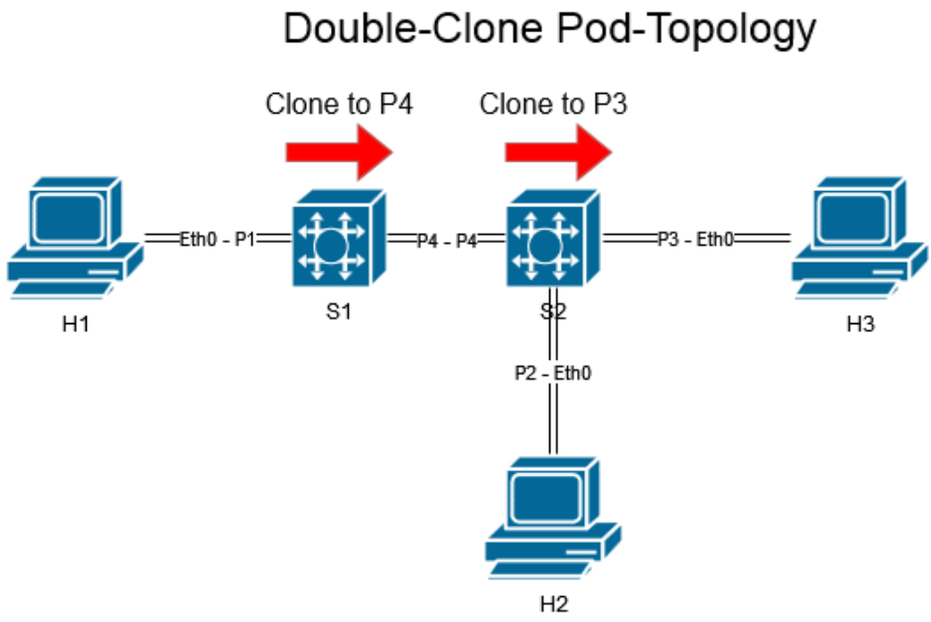


Figure 3.7: Network topology for Double-Clone

3.3.4 Resubmit

The next three exercises use *Resubmit* as a way of duplication. The network is constructed as in exercise 3.3.1.

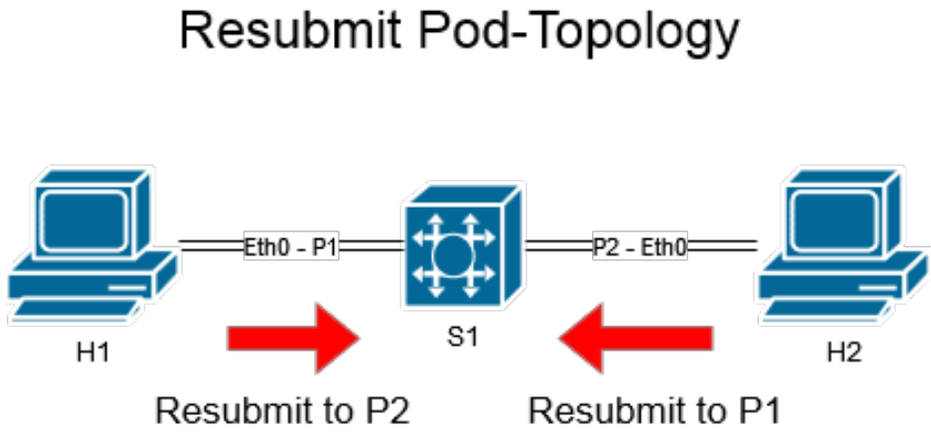


Figure 3.8: Network topology for Resubmit

Headers are not extracted. The packet is forwarded unchanged. For forwarding, it is important to distinguish duplicated packets from normal ones. This is achieved by means of the following constants:

Code listing 3.8: Types of duplication/mirroring

```
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;
```

Forwarding also uses the *own metadata* for the first time. The *ingress port* is stored in these.

Code listing 3.9: MyMeta struct

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

The *ingress processing* in this exercise is much more sophisticated than in the previous exercises. To begin with, the duplication types declared above are used to check whether a package has already been duplicated (`if (standard_metadata.instance_type == BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT)`. If this is the case, the ingress port in the metadata (`if (meta.mymeta.port == 1)`) shall be used to set the *Egress-Port*. If neither of the first two conditions is true, the *ingress port* (`standard_metadata.ingress_port == 0`) of the resubmit packet, the *Egress-Port* shall be set to *P2*. When the original package (`else if (standard_metadata.instance_type == BMV2_V1MODEL_INSTANCE_TYPE_NORMAL)` is processed, first the *ingress port* is stored in the *port field* of the *metadata* (`meta.mymeta.port = _metadata.ingress_port;`). Then the *resubmit* command is set (`resubmit_preserving_field_list(1);`). The `1` signals in which field of the own metadata should be stored as long as the packet has not yet been sent. Then, using the *Ingress-Port*, the *Egress-Port* is set as in exercise 3.3.1.

Code listing 3.10: Forwarding code for Resubmit

```
1  apply {
2    // Check whether packet was resubmitted
3    if (standard_metadata.instance_type == ↩
         BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT){
4    // packet is already resubmitted --> change outgoing port :)
5
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 == ↩
          BMV2_V1MODEL_INSTANCE_TYPE_NORMAL){
18    // standard, incoming packet --> resubmit
19      // --> saves ingress port in "MYMETA.PORT" --> important later :)
20      meta.mymeta.port = standard_metadata.ingress_port;
21      resubmit_preserving_field_list(1);
22      // set outgoing port -> does nothing because "original" packet isnt ↩
            transmitted, only resubmitted packet
23      if (standard_metadata.ingress_port == 1) {
24                      standard_metadata.egress_spec = 2;
25          }
26          else if (standard_metadata.ingress_port == 2) {
27              standard_metadata.egress_spec = 1;
28          }
29      }
30  }
```

3.3.5 Resubmit2-IP

The network of exercise „Resubmit2-IP“ is constructed as in exercise 3.3.2. Since forwarding is done by means of a *table*, headers for Ethernet and IPv4 are created in the programme. These headers are extracted in the *parser* to be able to work with them in the programme. The IPv4 header is created as in section 2.4. The type `macAddr_t` was declared using `typedef`. The Ethernet header has been implemented as follows:

Code listing 3.11: Ethernet-Header in P4

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

These two headers must be entered in the `struct headers`:

Code listing 3.12: Header-Struct

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

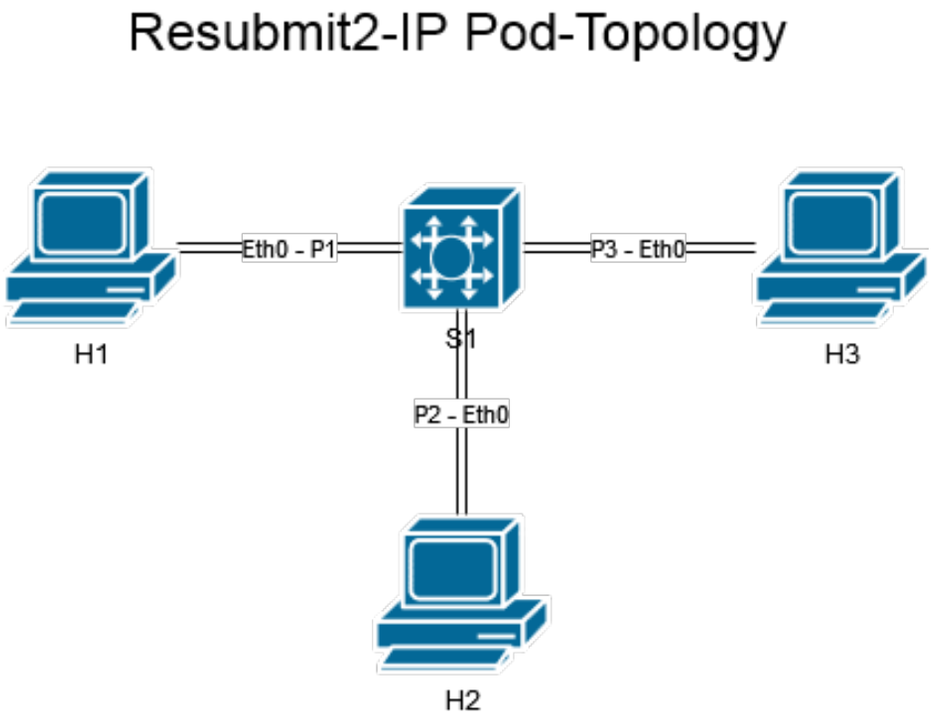


Figure 3.9: Network topology for Resubmit2-IP

The *parser* must now be instructed to read these two headers and decide which `state` to process next based on the data they contain. In this exercise, this decision is made based only on the *Ethertype* in the Ethernet header. The *Ethertype* is used to identify the protocol at Layer3 (similar to the *Protocol-Field* in the IPv4 header, see section 2.4 (Capabilities of P4). For this, two IDs of the *Ethertypes* were created as in [Ethertype-Numbers]:

Code listing 3.13: Ethertypes in P4

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

The *parser* is instructed to go directly from `state start` to `state parse_ethernet`. There, the Ethernet header is extracted and the *Ethertype* is used to decide which `state` to perform next (`transition select(hdr.ethernet.etherType)`).

Code listing 3.14: Parser State-Machine for 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  }
```

Two *actions* have been defined. The `action drop` marks the field in the `standard_metadata` which signals to the *Traffic Manager* (TM) that the packet is dropped and not sent. The `action ipv4_forward` sets the outgoing port and the destination *media access control* (MAC) address using the data from the `table ipv4_lpm`. In addition, the *Time to Live* (TTL) is decremented and the source MAC address is rewritten. The `table ipv4_lpm` uses as `key` the destination IP address and the *match criterion* `lpm` (cf. section 2.3.4). If an entry matching the *Key* is found when searching the *table*, one of the two previously explained *Actions* or `NoAction` can be executed. `NoAction` is the *Default-Action*. This *table* can have up to 1024 entries (`size = 1024`). The entries must be inserted into the *table* via the *Control-Plane*. This is discussed in the section 3.4. In this example, the interaction of *Control-* and *Data-Plane* becomes clear again. The entries previously inserted from the *Control-Plane* into a *table* are used by the *Data-Plane* to decide for each packet individually whether and on which port this packet is sent.

The `apply` block is configured as in exercise 3.3.4. The static forwarding based on the *ingress-port* has been replaced here by applying the *table* (`ipv4_lpm.apply();`). The condition `if(hdr.ipv4.isValid())` can be used to check whether an IPv4 header is present in the current packet. For easier testing, this condition has been commented out. Should this line want to be used again, the next *else condition* must also be used again. Otherwise, for all packets that do not have an IPv4 header (e.g. *Address Resolution Protocol* (ARP)), no *outgoing port* will be set and the packets will not be forwarded.

Code listing 3.15: 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 == ↩
            BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT){
29      // packet is already resubmitted --> change outgoing port :)
30              // if(hdr.ipv4.isValid()){
31                      ipv4_lpm.apply();
32              // }
33              // else{
34            // standard_metadata.egress_spec = 2;
35            // }
36      }
37
38      else if (standard_metadata.instance_type == ↩
            BMV2_V1MODEL_INSTANCE_TYPE_NORMAL){
39        // standard, incoming packet --> resubmit
40        meta.mymeta.port = standard_metadata.ingress_port;
41        resubmit_preserving_field_list(1);
42      }
43      }
```

Since in the *Ingress-Processing* the data in the IPv4 header is changed, the *Checksum* must also be recalculated. In the *Checksum-Computation-Control-Block* all fields of the IPv4 header must be used for this purpose:

Code listing 3.16: Checksum computation for 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 }
```

Since both IPv4 and Ethernet headers (MAC addresses) have been changed, these two headers must be explicitly sent in the *deparser* (`packet.emit(hdr.X`[12]):

Code listing 3.17: Deparser for 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

The network for this exercise is constructed as in exercise 3.3.5 Resubmit2-IP (see figure 3.10). If one host wants to send packets to another, the sender must know the MAC address of the destination. To find this out, there is the *ARP* protocol. It assigns a destination MAC address to a destination IP address. To do this, the sender sends a broadcast[13] packet. It is sent to the destination IP address using the broadcast MAC address (ff:ff:ff:ff:ff:ff). If a host has the requested IP address, it replies to the original sender using its own MAC address. This data is stored in the *ARP table*. [see [RFC826]] Ṫhis mechanism was also implemented in this exercise to make the network more dynamic and require less manual configuration. In addition, the *Internet Control Message Protocol* (ICMP) makes use of this technique after [RFC792]. *Headers, Clone Session IDs, Clone Types, Ethertypes, IPv4 Protocol Numbers, Custom Metadata, Checksum Computation, Deparser, Tables and Actions* were taken from exercise Resubmit2-IP. In order to be able to forward broadcasts, an *Action* must be declared in the *Ingress-Processing*. The mechanism *Multicast* is used for

[12] X = placeholder for the desired header
[13] send to the entire subnet

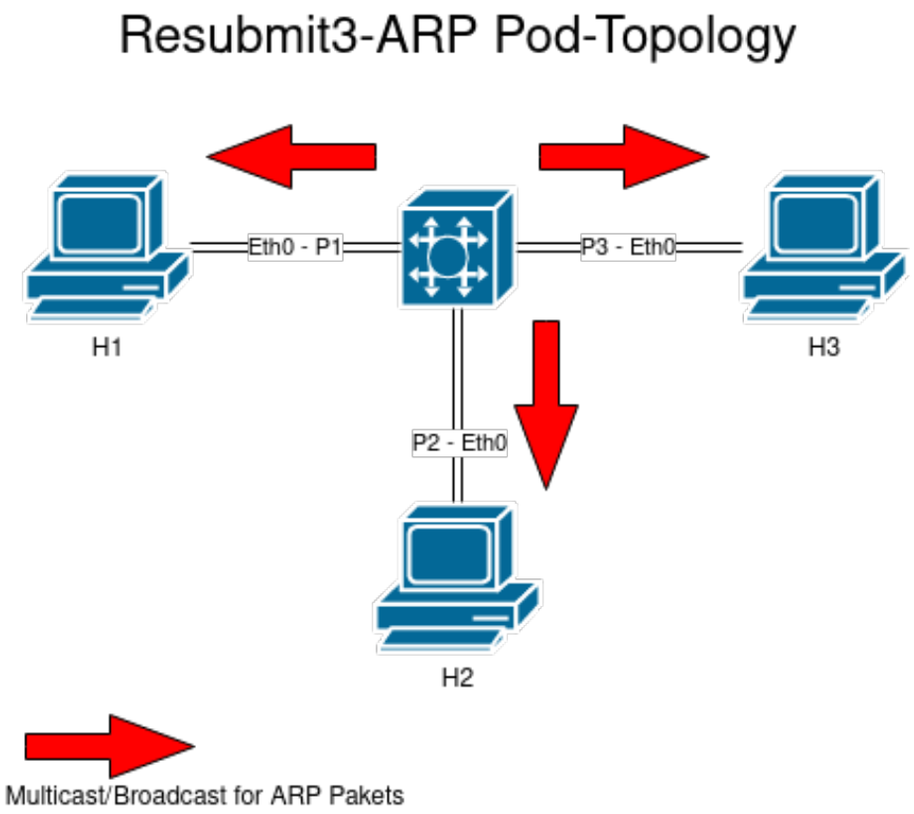


Figure 3.10: Network topology for Resubmit3-ARP

this, as there is no broadcast mechanism in P4. Multicast enables communication between a sender and several receivers [see [RFC1112]].
For this, a group must be assigned to multicast (`standard_metadata.mcast_grp = 1;`) (here: 1) and the TTL must be changed. This group signals when sending to which ports the original packet should be sent (see section 3.4 Mininet configuration).

Code listing 3.18: Actions for 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 }
```

In *Ingress-Processing*, *EtherType* is used to check whether an ARP or IPv4 packet is currently being processed. If it is an ARP packet (`if (hdr.ethernet.etherType == TYPE_ARP)`), it is sent as a broadcast using `action arp_forward()`. If it is an IPv4 packet (`else if (hdr.ethernet.etherType == TYPE_IPV4)`), this is handled as in exer-

cise Resubmit2-IP. The exact assignment of literals to variables as in the previous code extract in line 5 (`+ 8w255`) can be found in [P4 Lang. Spec. section 6.3.3.2].

Code listing 3.19: Ingress-Processing for 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 == ↩
           BMV2_V1MODEL_INSTANCE_TYPE_RESUBMIT){
8      // packet is already resubmitted --> change outgoing port :)
9            // if(hdr.ipv4.isValid()){
10                ipv4_lpm.apply();
11           // }
12     }
13
14     else if (standard_metadata.instance_type == ↩
           BMV2_V1MODEL_INSTANCE_TYPE_NORMAL){
15       // standard, incoming packet --> resubmit
16       // --> saves ingress port in meta.mymeta.port --> important later :)
17       meta.mymeta.port = standard_metadata.ingress_port;
18       resubmit_preserving_field_list(1);
19     }
20 }
```

3.3.7 Further exercises

Other exercises/programmes have been created, which are briefly explained here.

Name	Description
100-clones	100 clones of a packet are made and sent
Calculator	Exercise programme from an Intel course: the switch is repurposed to a calculator and sends the result of the calculation back to the requesting host
Clone-Draft	Clone header implementation as in section 2.4

Table 3.3: Further exercises in the Git-Repo

3.4 Mininet configuration

Mininet is installed on the VM. Mininet can initialise a virtual network including network devices and hosts. Switches incl. control planes and entries in tables as well as host PCs are configured here. In each folder of the exercises there is the folder `pod-topo`. This contains the three files needed for the configuration:

File name	Use
s1-commands.txt	Mirror-Session configuration Multicast configuration
s1-runtime.json	Table configuration incl. Match-Action Adding table entries
topology.json	Host creation incl. IP- und MAC-addresses Switch creation virtual device cabling

Table 3.4: Files for Mininet configuration and their respective use

The file types and names must not be changed. If several switches are used as in exercise 3.3.3 (Double Clone), there are also the specific command and runtime files for these (cf. code extract in section 3.4.1, line 14f).

3.4.1 Topology configuration

The configuration of the network is explained using the network from the exercise Double Clone, as two switches are used in this exercise. In this way, the interconnection of two network devices can also be explained. In the first section (lines 2-8) of the following code, the `hosts` are configured with name, IP and MAC address. In the second part (lines 10-13), the `switches` are initialised with names and input files (cf. table 3.4). At the end (lines 18ff), switches and hosts are connected to each other. For hosts, their name (e.g. `h3`) and the corresponding switch port (e.g. `s2-p3` for switch 2 port 3) are specified; for switch interconnections, the port description (`["s1-p4", "s2-p4"]`) is given in each case.

Code listing 3.20: topology.json for 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 Switch configuration

Command and runtime files must be created for each switch (see table 3.4). These files are explained in the next two subsections.

Mirror and Multicast settings

The *Clone-Session-IDs* used in the 3.3.2 and 3.3.3 exercises must be created on the switch in the `s1-commands.txt` file. Using `mirroring_add 100 3`, the *target* is instructed to send all clones with `Clone-Session-ID = 100` out on port3. Via `mc_mgrp_create 1` the multicast group 1 is created (cf. Resubmit3-ARP).

`mc_node_create X Y` assigns a *Replication-ID* (rid) (`intrinsic_metadata.egress_rid`) X to an *Egress-Port* Y. The *Replication-ID* can be used as an additional *Key* for *Tables* and *Actions*. The *replication ID* can be used as an additional *key* for *tables* and *actions*. For example, a packet can be sent twice over the same port but have different destination IP addresses [see [MC-Replication-ID]]. Since this mechanism is not used in the exercises and therefore it does not matter what is entered there, `rid = 0` was set. `mc_node_associate X Y` assigns node Y to the multicast group X. Note that the first host *H1* from code 3.4.1 was assigned the node index 0 (zero). Analogously, the remaining hosts.

Code listing 3.21: s1-commands.txt for 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
```

Table creation and adding entries

Tables created in the P4 programme must be configured in the control plane of the *target*. In the first section of the runtime file (`s1-runtime.json`), `bmv2` is specified as the target architecture and the info and JSON configuration file are declared. Then the `table ipv4_lpm` from the *control block* `MyIngress` together with *Default-Action* including parameters are declared. Then three entries are added to this table. Here the *Key* from the programme is called `„match"`. Then follows the *action* to be executed (`„action_name"`) including parameters and transfer values. If for the first entry (line 12-22) the *key* is true, the *action* `MyIngress.ipv4_forward` will be executed. The MAC address `08:00:00:01:11` and the *Egress port 1* (`"port": 1`) are then set in the *Action* in the programme.

Code listing 3.22: s1-runtime.json for 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_lpm",
8        "default_action": true,
9        "action_name": "MyIngress.drop",
10       "action_params": { }
11     },
12     {
13       "table": "MyIngress.ipv4_lpm",
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 }
```

The import of table entries with the file `s1-runtime.json` did not work at first. Therefore, the necessary commands that are used for a manual configuration are listed here. To do this, log on to the switch (S1) (`simple_switch_CLI -thrift-ip 10.0.2.15`). This command must be entered from the project folder (terminal) and not in Mininet. In the CLI of the switch, entries can be added according to the following example:

Code listing 3.23: Adding table entries to a mininet switch

```
1 // Basic command
2 table_add CONTROLBLOCKNAME.TABLENAME CONTROLBLOCKNAME.ACTIONNAME KEY => ↩
      PARAMETER
3
4 // Adding an entry with key="10.0.1.2" in table "ipv4_lpm" in the control ↩
      block "MyIngress" with action "ipv4_forward" with parameters ↩
      "08:00:00:00:02:22" und "2" (=Port)
5 table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.1.2 => ↩
      08:00:00:00:02:22 2
```

The automated import of these entries worked by the end of this work.

3.4.3 Adjusting remaining files

As the last step of the configuration, the *Makefiles* must be adapted. These are needed for compiling the programmes and topologies. In each folder of an exercise there is an exercise-specific Makefile. There is also a general Makefile, which is not stored in the Git repo, but in ∼`/tutorials/utils/`[14].

[14]folder from the P4 Language Consortium

exercise-specific makefile

The exercise-specific Makefiles first specify which switch model to use. This line should not be changed. Then the desired topology (cf. section 3.4.1) is referenced. The parent Makefile is included last. For these two files it is important to specify the correct folder path and file name. Depending on the folder in which the Git repo is located, this path must be adjusted if it is a relative path specification. To avoid this, the absolute path was specified. This works independently of the installation location of the git repo.

Code listing 3.24: Exercise specific Makefile

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

Parent Makefile

Here it is important to check that the following path and file name are specified correctly:
`RUN_SCRIPT = ~/tutorials/utils/run_exercise.py`
All paths or files can be specified as absolute or relative.

3.5 Use of the exercises

3.5.1 Start the exercise

The P4 programme and Mininet can be started via `make`. To do this, change to the exercise folder in the terminal via `cd FOLDERPATH`. If there are errors in one of the two Makefiles, they will be displayed in the terminal. If the files are compiled successfully, the configured hosts including port, IP and MAC addresses and files used for the configuration of the target(s) are displayed. It also shows how to change the configuration of the running target, view its logs and record network traffic.

```
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>
```

Figure 3.11: Starting the exercise

Useful *Command Line Interface* (CLI) commands in Mininet can be seen in table 3.5.

Command	Description
net	lists devices and where interfaces are connected to
nodes	lists all devices
links	lists connections between devices and their status
exit	Terminate Mininet

Table 3.5: Useful commands in Mininet

The output of these commands for exercise Double Clone looks like this:

```
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)
```

Figure 3.12: Screen output of the useful Mininet commands

3.5.2 Log-Files

After starting the exercise, the folder `logs` is also created in the respective exercise directory. Three files are created there for each switch. The most important is `SWITCHNAME.log`. This records the processing of each incoming packet including all *control blocks* and the *actions, conditions* and other processes contained therein. If a programme does not work as desired, this file is a good clue to identify possible sources of error. A short excerpt of the log of the first switch (S1) from exercise 3.3.3 can be seen in figure 3.13.

```
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [cxt 0] Table 'tbl_do_clone_i2e': miss
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [cxt 0] Action entry is MyIngress.do_clone_i2e -
[10:10:35.063] [bmv2] [T] [thread 1534] [34.0] [cxt 0] Action MyIngress.do_clone_i2e
[10:10:35.063] [bmv2] [T] [thread 1534] [34.0] [cxt 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] [cxt 0] Pipeline 'ingress': end
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [cxt 0] Cloning packet at ingress
[10:10:35.063] [bmv2] [D] [thread 1534] [34.1] [cxt 0] Parser 'parser': start
[10:10:35.063] [bmv2] [D] [thread 1534] [34.1] [cxt 0] Parser 'parser' entering state 'start'
[10:10:35.063] [bmv2] [D] [thread 1534] [34.1] [cxt 0] Parser state 'start' has no switch, going to default next state
[10:10:35.063] [bmv2] [T] [thread 1534] [34.1] [cxt 0] Bytes parsed: 0
[10:10:35.063] [bmv2] [D] [thread 1534] [34.1] [cxt 0] Parser 'parser': end
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [cxt 0] Cloning packet to egress port 4
[10:10:35.063] [bmv2] [D] [thread 1534] [34.0] [cxt 0] Egress port is 4
[10:10:35.063] [bmv2] [D] [thread 1535] [34.1] [cxt 0] Pipeline 'egress': start
[10:10:35.063] [bmv2] [D] [thread 1535] [34.1] [cxt 0] Pipeline 'egress': end
[10:10:35.063] [bmv2] [D] [thread 1535] [34.1] [cxt 0] Deparser 'deparser': start
[10:10:35.063] [bmv2] [D] [thread 1535] [34.1] [cxt 0] Deparser 'deparser': end
[10:10:35.063] [bmv2] [D] [thread 1535] [34.0] [cxt 0] Pipeline 'egress': start
[10:10:35.063] [bmv2] [D] [thread 1535] [34.0] [cxt 0] Pipeline 'egress': end
[10:10:35.063] [bmv2] [D] [thread 1535] [34.0] [cxt 0] Deparser 'deparser': start
[10:10:35.063] [bmv2] [D] [thread 1535] [34.0] [cxt 0] Deparser 'deparser': end
[10:10:35.063] [bmv2] [D] [thread 1539] [34.1] [cxt 0] Transmitting packet of size 70 out of port 4
[10:10:35.064] [bmv2] [D] [thread 1539] [34.0] [cxt 0] Transmitting packet of size 70 out of port 4
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [cxt 0] Processing packet received on port 4
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [cxt 0] Parser 'parser': start
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [cxt 0] Parser 'parser' entering state 'start'
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [cxt 0] Parser state 'start' has no switch, going to default next state
[10:12:13.368] [bmv2] [T] [thread 1534] [35.0] [cxt 0] Bytes parsed: 0
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [cxt 0] Parser 'parser': end
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [cxt 0] Pipeline 'ingress': start
[10:12:13.368] [bmv2] [T] [thread 1534] [35.0] [cxt 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] [cxt 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] [cxt 0] Applying table 'tbl_doubleclone68'
[10:12:13.368] [bmv2] [D] [thread 1534] [35.0] [cxt 0] Looking up key:
```

Figure 3.13: Output `s1.log` for Double-Clone

In the `pcap` folder, network traffic is recorded for each incoming and outgoing switch port and can be inspected with programs such as *Wireshark*[15] or *tcpdump*[16].

3.5.3 Operation of hosts

The console of the hosts can be started in the CLI via `xterm HOSTNAME`. Multiple hosts can also be specified here (separated by *space*). In the console, ICMP packets (`ping`) and with the two Python files `send.py` and `receive.py` network packets can be sent and received. Common Linux network commands like `ifconfig` can be used. Scapy is also started and configured from here. The use of these tools is described in chapter 5 Tests.

3.5.4 Terminating the exercise

With `exit` Mininet is terminated. After that, log and build files still need to be deleted. This is done with the CLI commands `make stop` and `make clean`. These folders are in principle excluded from upload to the git repo. This has been stored in the file `.gitignore`.

[15] https://www.wireshark.org/
[16] https://www.tcpdump.org/

Chapter 4: Porting the programme to real hardware

This chapter deals with the hardware used in the laboratory. This includes the network topology in the lab, the configuration and use of the devices. In addition, some of the exercises from chapter 3 Virtual Environment are ported for use on the hardware. The programmes are stored in the Git repo in the `tofino-ports` folder.

```
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
```

Figure 4.1: Folder structure tofino-ports

4.1 Laboratory hardware

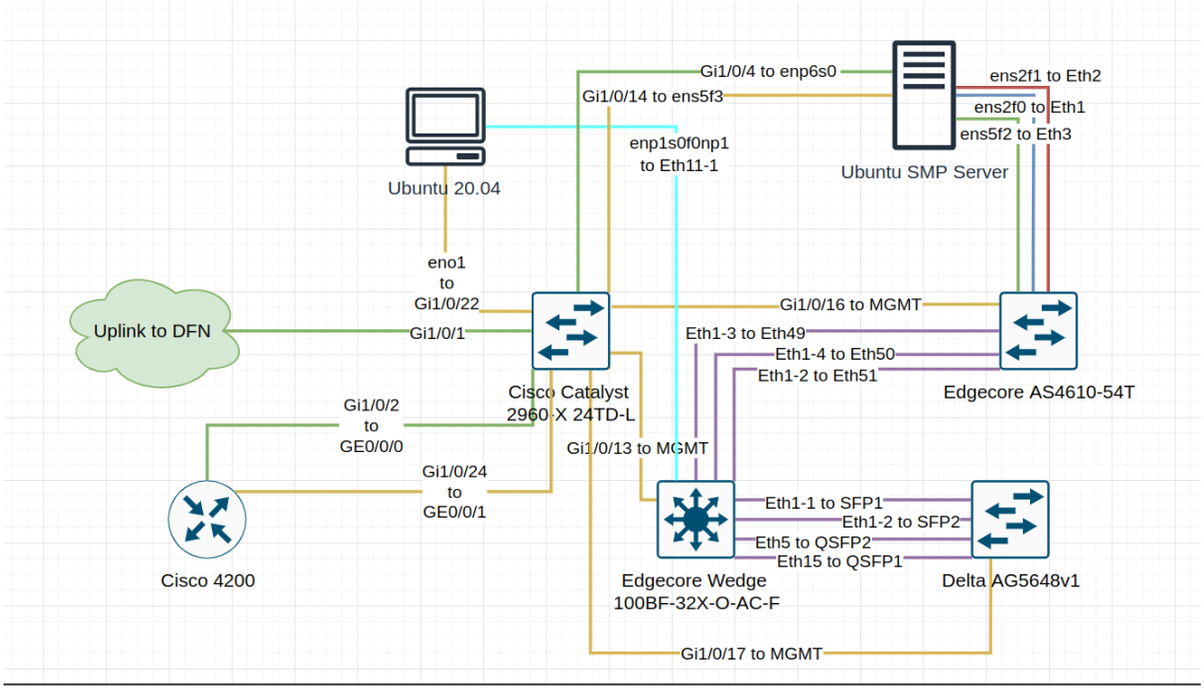
Three switches are installed in a rack in the laboratory:

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

These switches are so-called *whitebox switches*. An operating system must be installed on them. Ubuntu was used for this. A Linux PC and a Linux server (both Ubuntu 20.04.5 LTS) are connected to the switches for *Secure Shell* (SSH) connections and tests. A virtual instance of *Tofino Native Architecture* (TNA) is installed on the server. It contains a software switch with TNA, which is similar to the virtual switch from chapter 3 Virtual Environment. Since the exercises can be completed on real hardware, the virtual instance is not discussed further here. Various cables (RJ45[1], fibre optics) and the corresponding

[1]Registered Jack 45 = LAN cable

Small Form-factor Pluggables (SFPs)[2] are ready in the lab and already installed in the devices.
Both the server and the P4 switch have the *Barefoot Software Development Environment* (bf-sde) installed. The server runs version 9.9.0; the P4 switch 9.9.1. This contains all files for compiling and starting the P4 programme. Important shell scripts are contained here and also in the folder ∼/`tools`. Since there is a generic account (p4-user) on the switch, the installation of *bf-sde* and the importing of the git repo do not need to be done again. Therefore, this is not explained here.



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	

Figure 4.2: Network topology in the lab

Figure 4.2 represents the network in the lab. *Cisco 4200* is a router which routes between the different networks and also the internet. *Cisco Catalyst 2960-X* is the central switch in the lab, on which the different networks (*Virtual Local Area Networks* (VLANs)) are configured. *Edgecore AS4610* also has several VLANs separating the different test ports of the server at Layer2 so that packets cannot be mistakenly exchanged between the three different test ports (blue, red, green). This ensures that any P4 network traffic[3] from the

[2] optical modules, https://community.fs.com/de/blog/sfp-module-what-is-it-and-how-to-choose-it.html
[3] to distinguish it from Internet network traffic

server must always be sent to the P4 switch via ports Eth49, Eth50 or Eth51 of the AS4610. The cyane link from the Ubuntu PC to the P4 switch is a special feature here. Packets can be sent directly from the P4 switch to this port of the PC and do not have to take the normal route via AS4610. In addition, the speed of the port (25 Gbit/s) is suitable for running tests with a high data rate. For this purpose, an optical network card with SFPs was installed in the PC. The devices were connected with multimode optical fibres[4].
The switch *Delta AG5648V1* is not used in the exercises and is available for future applications.

Due to energy-saving measures, the three switches are switched off after each use. For this purpose, a multiple plug is installed in the rack, which can be operated via the network. For this purpose, *alias* were created on the PC and server (see appendix A). A `alias` is a short command under Linux, which can be set independently. Before switching off the P4 switch, make sure that the git repo is up to date and that changes have been uploaded. P4 programmes should also be terminated beforehand. Otherwise, data may be lost, which may entail a complete reset of the Git repo. Alternatively, buttons for manual on/off are also available on this multiple plug.

For SSH connections in this network, additional aliases have been created on the PC and/or server. The aliasses can be found in Appendix A. Useful entries for the IP addresses in this network have been created in the file `/etc/hosts` and can be seen in Appendix B. The IPv4 addresses of the interfaces of the server and the PC for the tests can be seen in Appendix I.

4.2 Configuring the hardware

4.2.1 Wedge 100BF-32X

The P4 switch has 32x 100 Gbit/s ports. Each of these ports can also be switched to 4x 10 or 25 Gbit/s or 2x 50 GBit/s via breakout cable[5] and configuration. To set the environment variable for *bf-sde* and install the associated driver, the following commands were entered in the file `.profile` on the switch:

Code listing 4.1: Driver and environment variable in .profile

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

They were entered there because they would otherwise have to be entered manually for each new shell session. Entering them in the file ensures that these two commands are executed with each new SSH session to the switch. If the driver has already been installed

[4] https://cleerlinefiber.com/2021/01/27/multimode-fiber-types/
[5] cable which multiplexes from 1 to 4 ports

(only required once after reboot), a negligible error message is displayed on the switch's console. Without these commands it is not possible to start the programme.

Compiling and starting the programme

P4 programmes can be compiled and started using the following commands:

Code listing 4.2: Complete commands for starting and compiling the programme

```
// Compile
~/tools/p4_build.sh PROGRAMMENAME.p4
// Starting the programme
./run_switchd.sh -p PROGRAMMENAME
```

To start the programme, it is important to change to the SDE folder beforehand (`sde`). When compiling, the `PROGRAMMENAME` with file extension (`.p4`) must be given; when starting, only the name without file extension. Aliasses were created for this on the P4 switch:

Code listing 4.3: Aliasses for compiling and starting

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

These are used like the full commands, specifying the programme name. After successful compilation and start of the programme, one switches to the shell (= `bfshell`) of *bf-sde*.

BFShell

Switch ports must be configured in the `bfshell`. Entries can be added to *tables* incl. *Actions*. In addition, this shell contains *Barefoot Runtime Python* (`bfrt_python`). This can be used to run Python[6] scripts which are used to configure the switch. Further tools are available in the `bfshell`.

UCLI

The configuration of the switch ports is started from the `bfshell` using `ucli`. Three commands are required per port to activate forwarding:

Code listing 4.4: Port-Config in BFShell

```
// 1 - add port
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)>
```

[6] https://www.python.org/

```
// 2 - setup autonegotiation
an-set <port_str> <autonegotiation (0 = automatisch, 2 = ausgeschalten)
// 3 - enable port
port-enb <port_str>
```

As an example, the second breakout port of the physically first port (`1/2`) is configured with 10GBit/s (`10G`) and none (`NONE`) *Forward Error Correction* (FEC)[7]. In the second command, *Auto Negotiation* (AN) is turned off (`2`). With this feature turned on, two directly connected network devices can negotiate the physical parameters of the connection (e.g. speed, duplex mode) independently. Since the AS4610 switch does not support this feature, it must be switched off on the ports of the P4 switch that are connected to the AS4610. The third command enables the entire first port.

Code listing 4.5: Configuration example of a switchport

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

The complete configuration of the switch ports can be seen in the Appendix C. The port configuration can then be output using `pm show`. Important columns here are `D_P` (port name for the P4 programme and tables) and `OPR` (= Operational Status: is the port actively connected to another device? UP ⇒ yes; DWN[8] ⇒ no). The two `FRAMES` columns show how many Layer2 frames have been received (`RX`) or sent (`TX`) on the respective port. `exit` terminates `ucli` and switches back to the `bfshell`.

```
bf-sde> pm show
-----+----+---+----+-------+----+--+--+---+---+---+--------+---------------+---------------+-
PORT |MAC |D_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|
```

Figure 4.3: Output port configuration

BFRT

With the command `bfrt` it is possible to switch to the *Barefoot Runtime* to create *table entries*. Here, standard nodes[9] (e.g. `mirror` or `port`) and the node, which automatically takes the name of the P4 programme, are displayed first. The *tables* created in the

[7] https://www.rfc-editor.org/rfc/rfc6363
[8] DWN = Down
[9] these are created automatically

programme can be found in this node. In `mirror` the tables are stored which are used for package duplication. The following scheme can be used to switch to one of the programme tables:

Code listing 4.6: Accessing a table in BFRT

```
bfrt.P4-PROGRAMMENAME.pipe.CONTROLBLOCKNAME.TABLENAME
```

These commands can also be entered individually, one after the other. Autocompletion of the commands via *Tab* key is possible as in Linux.

The `info()` command can be used to display the size (`capacity`), number of entries (`Usage`), *Key-Fields* with their *Match-Action-Types*, parameters (`Data Fields`) and *Actions* of the table. The entries of a *table* can be dumped using `dump()`. The following commands can be used to modify table entries:

Code listing 4.7: Commands for editing table entries

```
// Add entry with action
add_with_ACTIONNAME(KEY, PARAMETER)
// Edit entry with action
mod_with_ACTIONNAME(KEY, PARAMETER)
// Configure default-action
set_default_with_ACTIONNAME(KEY, PARAMETER)
```

Key, *Action* and their parameters must always be specified. For entries with *match-criterion* `ternary`, an additional *mask* must be specified (see appendix G).

Terminating the programme

The running P4 programme can be terminated from the *bfshell* using the following commands:

Operating system	**Keyboard command**
Windows	Ctrl + 4
Linux	Ctrl + AltGr + ß
macOS	Ctrl + Alt + Shift + 7

Table 4.1: Keyboard commands for exiting the P4 programme

This section may be a little bit different concerning your keyboard layout. These work with a standard german keyboard (QWERTZ).

4.2.2 IP configuration of the server and PC

IP configuration server

The three test interfaces of the server (cf. figure 4.2: blue, red, green link) are configured with the tool `ip netns`[10]. `netns` stands for *network namespaces*. It allows the creation of *namespaces*. These naming contexts allow the configuration of interfaces with their own routing table and firewall rules. This prevents direct communication between the interfaces [see [IP-NetNS]]. This is important for the tests to ensure that the sent packets pass through the P4 switch. Each namespace must be assigned an interface, a name and an IP address before the interface is activated. The following code excerpt serves as an example. The complete configuration can be seen in Appendix D. A complete IPv4 address list for the test interfaces can be seen in Appendix I.

Code listing 4.8: Configuring namespace blue on the server

```
1  // Creating namespace 'blue'
2  sudo ip netns add blue
3
4  // Activating the physical interface ens2f0
5  sudo ifconfig ens2f0 up
6
7  // Adding the interface to the namespace
8  sudo ip link set dev ens2f0 netns blue
9
10 // IP address configuration + activating the 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 // Output of the namespace interface configuration
15 sudo ip netns exec blue ifconfig
```

IP-Konfiguration PC

On the Ubuntu PC, only the optical interface `enp1s0f1np1` with the IP address `10.1.1.5/24` was configured without a gateway in the *Graphical User Interface* (GUI).

4.2.3 Configuring the AS4610

The P4 switch has only optical interfaces (*portspeed* $\geq$ 10 GBit/s). Since the PC originally only had 1GBit/s RJ45 ports, these had to be connected to the AS4610. As in section 4.2.2, to ensure that all packets are processed through the P4 switch, AS4610 was also configured

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

accordingly. The connected test ports (Eth1-Eth3) of the server were separated using *VLANs*. To enable communication with the P4 switch, a counterpart with a connection to the P4 switch was created for each of these ports and VLANs (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

Table 4.2: Port config AS4610 and corresponding ports

The files containing these configurations are stored in the ∼`/etc/systemd/network/` directory. The contents of these files can be seen in Appendix E. The documentation for configuring the switch is available online:
https://docs.bisdn.de/platform_configuration.html

4.3 Porting the code from BMv2 to TNA

Since *TNA* has a different structure than *BMv2* (see section 2.3.1) and *actions* are used differently, a P4 programme written for *BMv2* must be adapted. The exercises 3.3.1, 3.3.2 and 3.3.3 have been ported for use on the P4 switch. These were stored in the Git repo under `tofino-ports/`. A folder for tests was also created in this directory. Tests here refers to changes in the P4 code. Python scripts - as described in section 4.2.1 - were added to the two clone exercises to populate their *tables*.

4.3.1 Common

The *TNA* must be imported at the beginning of the programme using `#include <tna.p4>`. The control blocks must be created as in section 2.3.1. The parameters of each *control block* must be passed as in `tofino1_arch.p4`[11]. The file `tofino1_base.p4`[12] also shows which variables are contained in the individual parameters.

4.3.2 Changes in the Parser

In the *parser* it is important that in the `state start` the *Ingress Intrinsic Metadata* are extracted with the command `pkt.extract(ig_intr_md)`[13]. `Pkt` and `ig_intr_md` are the names of the *parameters* `packet_in` and `ingress_intrinsic_metadata`. To verify a *checksum* with `ipv4_checksum.verify()`, it must be declared using `checksum() NAME`.

[11] https://github.com/barefootnetworks/Open-Tofino/blob/master/share/p4c/p4include/tofino1_arch.p4
[12] https://github.com/barefootnetworks/Open-Tofino/blob/master/share/p4c/p4include/tofino1_base.p4
[13] in the egress parser: eg_intr_md

4.3.3 Changes in the Deparser

If a *header* has been modified during processing, `pkt.emit(hdr)` in the `apply` block must be executed to send the packet. If there are no changes to the header data, there is no need to use `emit` in either *TNA* or *Behavioral Model Version 2* (BMv2) (see exercise Hello World).
To recalculate the *Checksum*, it must be declared via `Checksum() NAME`. The code for this can be seen in appendix F. To duplicate packages a mirror instance must be created with `Mirror() MIRRORNAME` and `emit<mirror_hdr>(MIRROR_SESSION_ID, {MIRROR_DATA})`. For this it is mandatory to also check the *mirror type* beforehand by means of *if-condition* [see [TNA]].

4.3.4 Changes in the packet duplication

The *Clone Session IDs* from exercise 3.3.2 (Simple Clone) must also be used in the *TNA* for packet duplication. These are described in the exercises *Simple-Clone* and *Double-Clone* in the *table* `port_acl`. After the packet has been duplicated using `emit`, the *clone session ID* is read from the *table* `mirror` in the *BFRT* and sent with the configured *action*. Python scripts are created in the project folders of these two exercises, which can be applied to the P4 switch with the following command after starting the programme and configuring the switch ports:

Code listing 4.9: Execution of a Python script for creating table entries

```
1 // Common syntax
2 ~/bf-sde-9.9.1/run_bfshell.sh -b ~/PROJEKTPFAD/SKRIPTNAME.py -i
3 // Example for 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
```

When executing the script, it is important that no other instance of *BFRT* is running. Otherwise an error message will be displayed. If an instance is open, it can be terminated with `exit` and the script can be executed.

The following code extract shows the creation of two table entries from a Python script. The first command sets the *clone-session-ID* (`mirror_session=1`) based on the *ingress-port*. The 9-bit long `ingress_port_mask`[14] with the value `510` is the *match criterion* (= `ternary`) of the entry. It signals, in conjunction with the *key* (`ingress_port=134`), to set `mirror_session = 1` for packets received on ports 134 and 135. The second command sets the *egress port* for all duplicated packets with *clone session ID 1* (`sid=1`) and validates it. In addition, this *Clone-Session* is enabled for both incoming and outgoing network traffic (`direction='BOTH'`)

[14]Ingress_Ports are 9 bits long, therefore the mask must also be so long

(`session_enable=True`).
At least these five specifications must be made for the table `mirror.cfg`. Otherwise the duplication on the P4 switch will not work. There are many, many more options that can be set in this table. The whole script can be seen in attachment G.

Code listing 4.10: Excerpt from the Python script for 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 Adapting the ports

One of the most important adjustments in porting is that of the physical interfaces (ports). The *egress port* is set with the following command
`ig_tm_md.ucast_egress_port` (BMv2: `standard_metadata.egress_spec`)
The port number can be determined as in section 4.2.1 (see figure 4.3, column D_P) and then entered in the programme.

Code listing 4.11: Comparison exercise Hello_World: Setting the Egress-Port in BMv2 and TNA

```
1 // Setting BMv2-Switch port 2 as Egress-Port
2     standard_metadata.egress_spec = 2
3
4 // Setting P4-Switch port Eth1-4 as Egress-Port
5     ig_tm_md.ucast_egress_port = 135
```

Chapter 5: Tests

This chapter deals with different ways to test the P4 programmes for functionality. The tools `ip-netns`, `ping`, `iperf` and `scapy` are already installed on the server. `Wireshark` is installed on the PC and the virtual hosts. In the virtual environment, `ping` and two Python scripts are also used. The tests with the different tools have the following goal:

1. Generating packets on a host
2. Sending those packets to another host
3. Successful receiving those packets on the second host
4. Displaying and verification of the network traffic

5.1 Tests in the virtual environment

After starting the programme, the consoles of the hosts must first be opened (`xterm HOSTNAME` see section 3.5.3).

5.1.1 Ping tests

`Pings` according to [RFC792] only work for the following exercises:

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

This results from the fact that the *Internet Control Message Protocol* (ICMP) (= `ping`) uses the *Address Resolution Protocol* (ARP) to obtain the MAC address of the destination host in addition to the desired destination IP address. This mechanism uses broadcast for this purpose. Broadcasts and ARP were only fully implemented in exercise *Resubmit3-ARP*. The other three exercises use static port forwarding. Therefore, `ping` is possible in these exercises. The test shown was taken in the exercise *Simple Clone*. In some tests, after the individual line outputs in Figure 5.1, there may still be a `(DUP!)` at the end of the line. This signals that the package is a duplicate and is a consequence of *mirroring*.

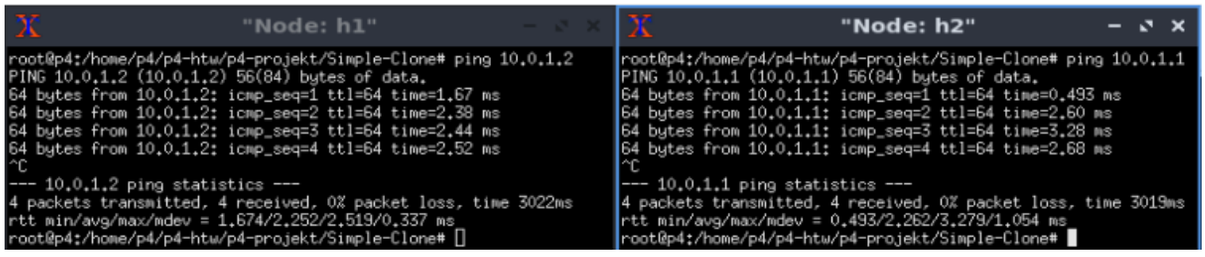


Figure 5.1: Ping in the VM

5.1.2 Python scripts

The Python scripts `send.py` and `receive.py` are created in each exercise folder. The first can be used to send test packets and the second to display them after they have been received. These must be started from the console of the hosts according to the following syntax:

```
1 // send.py
2 python3 send.py DESTINATION-IP-ADDRESS "MESSAGE"
3 // receive.py
4 python3 receive.py
```

The two scripts must be started on different hosts. `TARGET IP ADDRESS` is the IP address of the host on which `receive.py` is run. As an example, in exercise 3.3.3 a packet was sent from the first host (H1) to the second (H2) with the content „this is a p4-test“.
The packet is cloned on the first switch (S1). The original and clone are then sent to the second switch (S2), which in turn clones them again. Two packets each are then sent to H2 and the third host (H3) and are output to the console (see figure 5.2). *pings* are not displayed with `receive.py`.

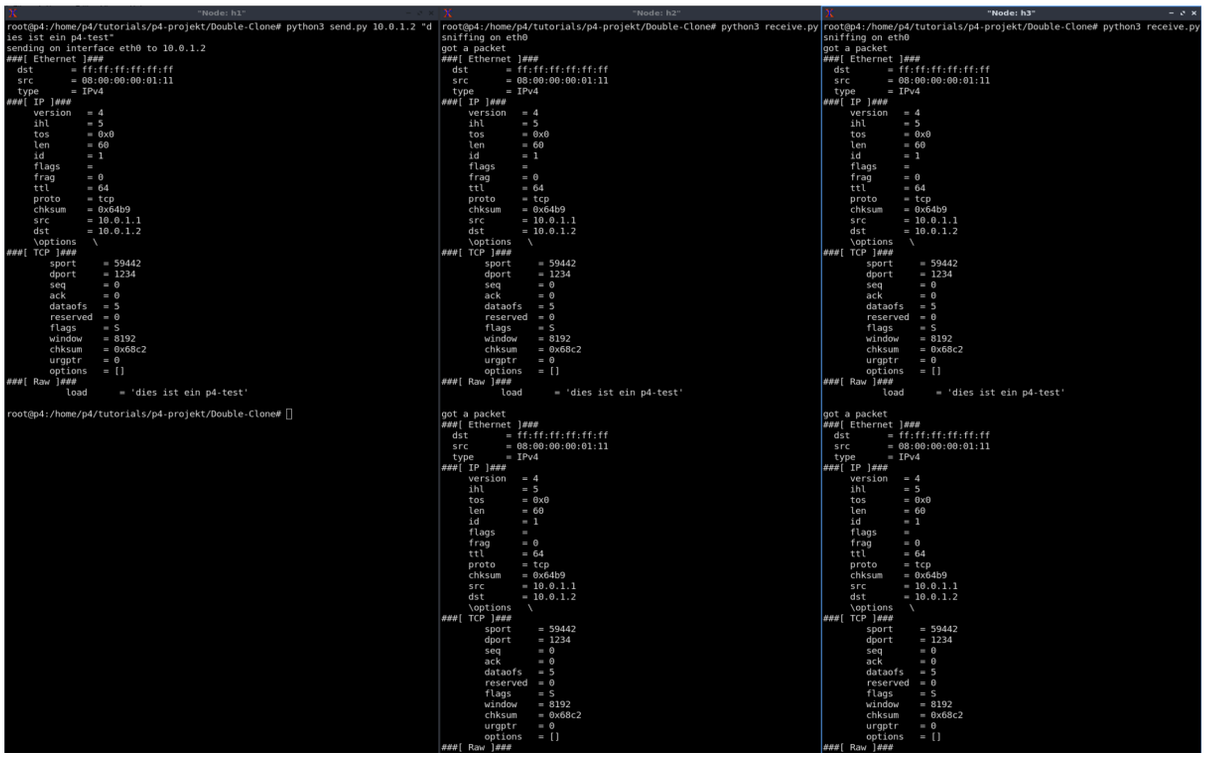


Figure 5.2: Python tests in the VM

5.2 Tests on the laboratory hardware

Tests on the hardware can be started as soon as the P4 programme has been executed on the switch, the switch ports configured and required table entries added (see chapter 4).

5.2.1 Ping on the hardware

`Ping` works on the hardware as in the Virtual Environment. For these tests, the three interfaces of the server from the *ip-netns naming contexts* are used (see section 4.2.2). For each interface a separate *ping alias* was created (see appendix A lines 29-31). With the command `blueping 10.1.1.2` the IP address of the naming context `red` is pinged by the interface of the context `blue`.

5.2.2 iPerf-Tests

iPerf is a tool to actively measure the data speed in IP networks. Many useful options can be set to optimise the tool for the corresponding needs and requirements. A few of these options are explained below. Further options of iPerf can be found in the online documentation of the tool: https://iperf.fr/iperf-doc.php
iPerf is used once again in the 5.2.4 section to generate network traffic.

iPerf Server

To be able to run tests with iPerf, one must start an iPerf server on one of the test ports of the Ubuntu server (`blue, red, green`). This is signalled in iPerf with the option `-s`. In the example, the interface from the ip-netns context `blue` with the IP address 10.1.1.1 was used in the CLI (line 1). As soon as a client establishes a connection to this server, this is displayed (line 6). Subsequently, the duration, amount and speed of the exchanged data is output.

Code listing 5.1: iPerf server configuration and 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

The interface from the naming context `red` is used as the client in the example. A client is set with the option `-c`. With `-p` the destination port [1] is set, which is in the iPerf server code in line 3. `-t` can be used to specify the duration of the test in seconds. `-i` describes the time in seconds after which information about the data traffic is output (here: every 2 seconds).

Code listing 5.2: iPerf client configuration and 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
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
```

[1] for Layer4: TCP (RFC9293) or UDP (RFC768)

5.2.3 Scapy tests

Scapy is a Python-based CLI tool. It allows the creation of specific network packets and also offers the possibility to write self-defined headers (see section 2.4) and include them in the packet. Scapy can be started for all interfaces or in the *ip-netns* name contexts. As Scapy is a versatile tool with various functions, it is recommended to read the online documentation[2]. The tool is started from the terminal with `scapy`. It is terminated with `exit`.

Generating packets

To create a packet in Scapy, it must first be built from the headers of the desired network layers. This is shown here using the Ethernet and IPv4 headers. After creating the headers, they must be merged into one packet.

Code listing 5.3: Generating the single headers and merging those

```
1 # Generating the Ethernet- and 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 # Merging those two headers in "packet"
6 packet = ethernet_header / ip_header
```

Scapy takes the source MAC address from the interface that sends the packet. In this example, Scapy was executed directly in a name context.

Verification of the packet

There are two functions (`show()` and `show2()`) which display the contents of the created packet or a header. The syntax for the call is as follows: `NAME.show()` or. `NAME.show2()`. The name of the packet or the respective header must be specified as `NAME`.

Sending the packet

There are two functions (`send(PACKETNAME)` and `sendp(PACKETNAME, ARGS)`) to send a packet. If Scapy is started from one of the name contexts, using the first function is sufficient. The second function allows to specify the sending interface in `ARGS`.

[2]https://scapy.readthedocs.io/en/latest/usage.html

Code listing 5.4: Commands for sending packets

```
1 # simple function
2 send(packet)
3 # extended function -> Sending via interface "eth0"
4 sendp(packet, iface="eth0")
```

Sniffing

Sniffing refers to the process of sniffing network traffic on an interface. The following command will output all network traffic on the command line in Scapy: `sniff(prn=lambda x: x.show())` The interface name can also be specified to show only packets of the specified interface.

5.2.4 Verification with Wireshark

Wireshark[3] is a tool to record and display network traffic. It has a graphical user interface (GUI) and displays the network traffic there. The programme must be started with admin rights. After starting the software, an interface to be inspected must be selected. The output of the network traffic can be adjusted by means of filters[4]. Wireshark can currently only be used from the Ubuntu PC.
The exercises 3.3.2 and 3.3.3 can be used with Wireshark in their version adapted for the hardware. All network traffic between the two interfaces `blue` and `red` of the server is cloned from the P4 switch to the 25 GBit/s port of the PC. The configuration of the tables takes place through the Python scripts in the folders of the respective exercises. Wireshark must be started on the `enp1s0f1np1` interface of the PC. If one starts an iPerf test as in section 5.2.2, the network traffic of this iPerf test is output in Wireshark. A small section of this network traffic can be seen in figure 5.3. The total network traffic of the 10-second test contains over 80000 packets (at 1 Gbit/s speed and 10 seconds duration). Based on the *source* and *destination*[5] in figures 5.3 and 5.4, it can be seen that these are packets between `red` and `blue` name spaces.

5.3 Test conclusion

In all exercises it could be ensured that network traffic between the hosts is possible. `Ping` is only possible in the virtual environment exercises mentioned in section 5.1.1. In the three exercises on the lab hardware, `ping` has been successfully implemented and tested.

[3] https://wiki.wireshark.org/Home
[4] e.g. output of IPv4 traffic only
[5] these are IPv4 addresses

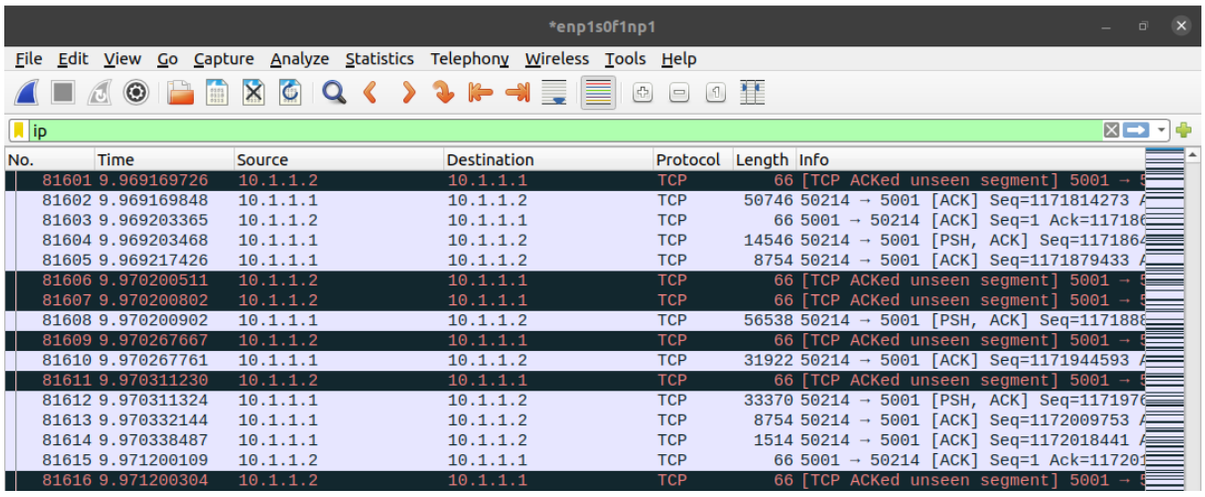


Figure 5.3: Cloned network traffic on the 25 GBit/s interface of the PC between `blue` and `red` using iPerf

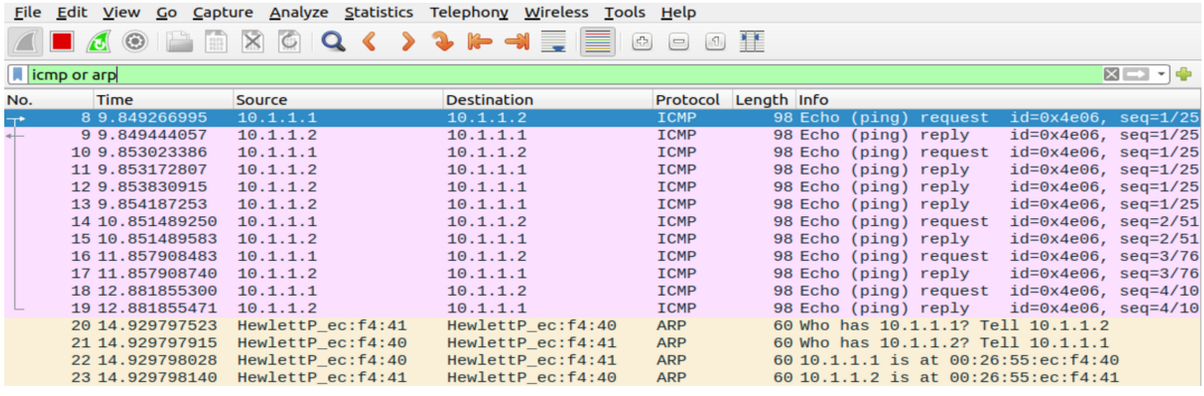


Figure 5.4: ICMP and ARP traffic in Wireshark

The network traffic can be analysed in all exercises using Python script (`receive.py`), `Wireshark`, `iPerf` or the *sniffing* in `Scapy`.
Since `Ping` does not work in the exercise 3.3.3 (Double Clone), the network traffic must be evaluated using `receive.py` and `Wireshark`. It is important in the evaluation in this exercise that one packet sent with `Scapy` arrives at the other two hosts in the exercise twice. This verifies that both switches in the exercise each create a clone of the original packet and send both original and clone.
The exercise 3.3.5 (Resubmit2-IP) is not so much a practical exercise, but the intermediate stage of the other two *Resubmit* exercises (see sections 3.3.4 and 3.3.6). Resubmit2-IP represents the evolution of Resubmit into Resubmit3-ARP and shows the steps required to switch a programme from static forwarding to dynamic forwarding. It can be seen that implementing the IPv4 header is not sufficient, but that the ARP header and broadcast must also be created to enable *ICMP* traffic. This is reflected in the exercise 3.3.6 (Resubmit3-ARP), as ARP header and broadcast have been implemented there, extending the possible network traffic (→ `ping` can be used). This is a first step for the programme to be used not only in laboratory networks but also in real networks.

Tables could be successfully created, configured and deployed in the exercises 3.3.5 and 3.3.6 of the virtual environment and the exercises 3.3.2 and 3.3.3 on the lab hardware. The 25 GBit/s interface of the PC is so far only used for the analysis of the network traffic and especially for the verification of the *mirroring*. In the future, this interface will also be used for generating packets and speed tests.

Chapter 6: Conclusion and prospects

The thesis presents a summary of the most important components of *P4* that are needed for an introduction to the programming language. These are mainly the different *control blocks*, *how* they are used, *what* they contain and *what* differences exist between *Behavioural Model Version 2* (BMv2) and *Tofino Native Architecture* (TNA).
For me and also other members of the research project, it is important to internalise that we can design a P4 programme completely according to our wishes and that the programme is in principle independent of external influences such as protocols or best practice approaches. No standards have to be implemented and one does not necessarily have to follow guidelines that are valid in normal networks. This is a consequence of the goal *protocol independence* (see section 2.2). Nevertheless, with P4 it is also possible to program a fully RFC-compliant *target* that can be integrated into existing networks (see exercises of the P4 Language Consortium in the VM).
With the possibilities of the programming language, especially the own design of protocol headers is very interesting, because headers can be implemented quickly and tested with the tools mentioned in this thesis. For me, this own design represents an unprecedented freedom in network technology, as one is not dependent on institutions such as hardware manufacturers and standardisation organisations (e.g. *Internet Engineering Task Force* (IETF)). This means that there are no limits to the design. Nevertheless, cooperation with these organisations is intended as soon as one's own headers have been sufficiently tested and represent added value for network technology.
By using the programming language, I began to deal with the detailed structure of important network protocols (IPv4, ARP, ICMP). This results from the fact that desired headers must be created at the beginning of the programme. Each bit in the field of a protocol header must be defined exactly as in the associated standard (e.g. RFCs) ($\rightarrow$ order and length of these fields)(see IPv4 header in section 2.4). The need to include headers also follows from the goal of *protocol independence*, as no network protocols are implemented in P4 by default. In my time as a network administrator, I found the exact structure of the protocols uninteresting, as the network devices forward network traffic on their own and the knowledge of *„where which bit is located in the header“* was not necessary for the job at the time. To my own amazement, this has changed with P4.

The exercises start with easy-to-understand rules for forwarding packets (see static forwarding in exercise 3.3.1 Hello World) and become increasingly complex. On the one hand, the increasing complexity results from the implementation of important network protocols such as IPv4, ARP and ICMP. On the other hand, forwarding using *tables* is also responsible for

this. These two points are enormously important, as they are indispensable for the operation of larger networks and the connection to networks of other organisations. As a result, some network protocols and related mechanisms such as *routing* or *multicast/broadcast* could be presented once again. The *matching criteria* `ternary` and *Longest-Prefix-Match* (LPM) could be better understood through explicit use and initial errors in use. Although the exercises are suitable for use in local networks such as the laboratory network, they are of limited use for public network traffic (Internet). The adaptation of the programmes so that the P4 switch can forward internet traffic without errors is intended for the future. In the course of this, IPv6 is also to be implemented.

In the future, parts of the code are to be removed in order to be able to check some intrinsic part of understanding. This is configured in the exercises available in the Virtual Machine (VM) of the *P4-Language-Consortium*. In addition to this, for each exercise there is a folder `solutions` in which the solution to the current task is stored. Good instructions for the exercises must also be created for this.
In addition, certain steps in the configuration of the *Control-Plane* of the P4 switch, such as the creation of table entries, are to be automated. The configuration of the switch ports should also be included in this automation. Perhaps a Python script with prompts would be suitable for this.

Debugging of the code and the resulting findings will be documented in the future. It should be mentioned here that most errors in the code are output when the programme is compiled and can vary greatly in their nature. Syntax errors such as missing semicolons or the impossibility of reading out data (cf. *directions* of the parameters in table 2.3) are among the most common error outputs when compiling.

Some *tables* and *actions* still need to be renamed, as they were taken from various code examples and misappropriated (e.g. `port_acl` from section 4.3.4). The table `port_acl` is not an *Access Control List* (ACL)[1] as the name suggests, but is used in the two *Clone* exercises on hardware to set the multicast group for *Mirroring*. Therefore, the name of the table should be changed to `set_mc_group` or similar.
The Tofino chip contains an internal packet generator. With this, tests can be completed independently of the rest of the laboratory hardware. This allows the actual goal of the research project to be pursued and the self-designed *Clone header* to be further adapted, tested and finally used. Further test scenarios can be designed with the 25 GBit/s interface of the PC to run tests at this speed.

A personal concern is to publish this work in order to make it easier for as many people as possible to get started with P4. This should happen in forums that deal with P4 such as the *Intel Connectivity Research Program* (ICRP)[2]. In addition, the work is awarded to the Berlin-based company AVM[3] (→ FRITZ!Box). AVM had already asked during my

[1] https://www.ittsystems.com/access-control-list-acl/
[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/

compulsory internship (at AVM) whether I could send them this work. After I explained to them what P4 is, they already saw possible scenarios for using P4 in the company.

Appendix A: Aliasses on Ubuntu-PC and/or server

Code listing A.1: Aliasses on Ubuntu Server/PC

```
1  alias ..='cd ..'
2  alias ls='ls -lisa --color=auto'
3
4  // Commands for turning on/off the switches
5  alias pwr-off-1='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output1=0"'
6  alias pwr-off-2='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output2=0"'
7  alias pwr-off-3='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output3=0"'
8  alias pwr-off-4='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output4=0"'
9  alias pwr-off-ag5648='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output2=0"'
10 alias pwr-off-as4610='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output1=0"'
11 alias pwr-off-wedge='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output3=0"'
12 alias pwr-on-1='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output1=1"'
13 alias pwr-on-2='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output2=1" '
14 alias pwr-on-3='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output3=1"'
15 alias pwr-on-4='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output4=1"'
16 alias pwr-on-ag5648='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output2=1" '
17 alias pwr-on-as4610='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output1=1"'
18 alias pwr-on-wedge='curl ↩
       "http://172.16.1.2/netio.cgi?pass=24A42C3926D6&output3=1"'
19
20 // SSH-Login commands
21 alias ssh-as4610='ssh basebox@172.16.1.8'
```

```
22 alias ssh-wedge='ssh p4-user@wedge-p4'
23
24 //cdp4: command 1 -> server, command 2 -> PC
25 alias cdp4='cd ~/p4-projekt/'
26 alias cdp4='cd ~/Documents/p4-projekt/'
27
28 // only on server -> commands for pinging from interfaces blue, red or green
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 // only on PC -> SSH-Login to server
34 alias ssh-liam='ssh marcel@liam'
```

Appendix B: /etc/hosts on Ubuntu-Server

Code listing B.1: /etc/hosts on Ubuntu-Server

```
# P4 LAN 172.16.1.0/24
172.16.1.1 cisco-router
172.16.1.2 ip-pwr
172.16.1.5 ag5648
172.16.1.8 as4610
172.16.1.10 wedgecore-bmc wedge-bmc
172.16.1.11 wedgecore-p4 wedge-p4 wedge-studio
172.16.1.12 p4-ubuntu-server p4-ubuntu-srv p4-srv liam
172.16.1.248 ipv6-p4-asw01 p4-asw01 asw01
172.16.1.249 ipv6-p4-asw02 p4-asw02 asw02
172.16.1.250 ipv6-p4-asw03 p4-asw03 asw03
```

Appendix C: Configuration of switchports on Wedge 100BF-32X

Code listing C.1: Complete configuration of switchports on Wedge 100BF-32X

```
// Adding ports with speed and FEC configuration
// Port 1 = Connections to AS4610
port-add 1/0 10G NONE
port-add 1/1 10G NONE
port-add 1/2 10G NONE
port-add 1/3 10G NONE

// Port 11 = Connection to Test-PC via 25Gbit/s Multimode-Fiber
port-add 11/0 25G NONE
port-add 11/1 25G NONE
port-add 11/2 25G NONE
port-add 11/3 25G NONE

// Auto-Negotiation
an-set 1/0 0
an-set 1/1 2
an-set 1/2 2
an-set 1/3 2

an-set 11/0 2
an-set 11/1 2
an-set 11/2 2
an-set 11/3 2

// Activating the physical ports
port-enb 1/-
port-enb 11/-
```

Appendix D: Complete IP-Netns-configuration of the server

```
// Creating namespaces
sudo ip netns add blue
sudo ip netns add red
sudo ip netns add green

// Activating physical interfaces
sudo ifconfig ens2f0 up
sudo ifconfig ens2f1 up
sudo ifconfig ens5f2 up

// Adding those interfaces to the namespace
sudo ip link set dev ens2f0 netns blue
sudo ip link set dev ens2f1 netns red
sudo ip link set dev ens5f2 netns green

// Configuring IP-Addresses + activating interfaces and loopbacks
sudo ip netns exec blue ifconfig ens2f0 10.1.1.1/24 up
sudo ip netns exec blue ifconfig lo up
sudo ip netns exec red ifconfig ens2f1 10.1.1.2/24 up
sudo ip netns exec red ifconfig lo up
sudo ip netns exec green ifconfig ens5f2 10.1.1.3/24 up
sudo ip netns exec green ifconfig lo up

// Output of Namespace-Interface-Configuration
sudo ip netns exec blue ifconfig
sudo ip netns exec red ifconfig
sudo ip netns exec green ifconfig
```

Appendix E: Configuration AS4610

Code listing E.1: Configuration AS4610

```
accton-as4610:/etc/systemd/network$ cat 10-swbridge.netdev
[NetDev]
Name=swbridge
Kind=bridge

[Bridge]
VLANFiltering=1
DefaultPVID=none
accton-as4610:/etc/systemd/network$ cat 10-swbridge.network
[Match]
Name=swbridge

[BridgeVLAN]
VLAN=10
VLAN=20
VLAN=30
accton-as4610:/etc/systemd/network$ cat 20-port1+49.network
[Match]
Name=port1 port49

[Network]
Bridge=swbridge

[BridgeVLAN]
PVID=10
EgressUntagged=10
accton-as4610:/etc/systemd/network$ cat 20-port2+50.network
[Match]
Name=port2 port50

[Network]
Bridge=swbridge

[BridgeVLAN]
PVID=20
EgressUntagged=20
accton-as4610:/etc/systemd/network$ cat 30-port3+51.network
```

```
[Match]
Name=port3 port4 port51

[Network]
Bridge=swbridge

[BridgeVLAN]
PVID=30
EgressUntagged=30
accton-as4610:/etc/systemd/network$ cat 70-enp.network

[Match]
# You can also use wildcards. Maybe you want enable dhcp
# an all eth* NICs
Name=enp*
[Network]
#DHCP=v4
# static IP
# 192.168.100.2 netmask 255.255.255.0
Address=172.16.1.8/24
Gateway=172.16.1.1
DNS=141.45.2.100 141.45.3.100
accton-as4610:/etc/systemd/network$ cat 90-enp.link
[Match]
Path=*platform-18022000.ethernet*
[Link]
Name=enp0
```

Appendix F: Checksum calculation

Code listing F.1: Checksum calculation at 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               });
```

Appendix G: Python script for table entries in Simple-Clone

```
from ipaddress import ip_address
#from ipaddress import ip_network # seems not to work, maybe because this creates another type of object (address vs network)
p4 = bfrt.tofino_simple_clone.pipe

#
# Program the default topology
# sometimes: put ".push()" after command

# Set Mirror Session to 1 for Ingress Ports 134-135 (QSFP1-3 und 1-4)
p4.Ingress.port_acl.add_with_acl_mirror(ingress_port=134, ingress_port_mask=510, mirror_session=1)


# Set for clone-id=1 with action=normal: enable session, direction=BOTH (in+egress), egress_port=40, validate port
bfrt.mirror.cfg.add_with_normal(sid=1, session_enable=True, direction='BOTH', ucast_egress_port=40, ucast_egress_port_valid=True)

#
p4.Egress.mirror_dest.add_with_just_send(ing_mirrored_mask=0, egr_mirrored_mask=0, mirror_session=1)

bfrt.complete_operations()

# Final programming
print("""
****************** PROGRAMMING RESULTS *****************
""")
print ("\nTable Ingress Port ACL:")
p4.Ingress.port_acl.dump(table=True)
print ("\nTable Egress Mirror Destination:")
p4.Egress.mirror_dest.dump(table=True)
print ("\nTable mirror.cfg:")
bfrt.mirror.cfg.dump(table=True)
```

Appendix H: Scapy File

```
// MBeausencourt
//This is a datagram class for a "Clone Datagram", which is designed for ↩
    use with Scapy

//Clone-Header for Scapy -> copy paste
class Clone(Packet):
     """This is header v0.1 for a Clone Datagram for P4 testing. """
     name = "CloneDatagram"
     fields_desc=[BitField("numberOfClones",0,64),
                BitField("cloneCounter",0,64),
                BitField("secureString",42,32),
                BitField("receiverOption",0,8),
                BitField("payloadLength",0,8),
                BitField("version", 0,8),
                BitField("free",0,8),
                BitField("receiverIP", 0,32),
                BitField("receiverMask",0,32),
                BitField("standardString",1234,64)]


// Header Version of Thomas
class Clone(Packet):
      """This is header v0.1 for a Clone Datagram for P4 testing. """
      name = "CloneDatagram"
      fields_desc=[LongField("numberOfClones",0),
                 LongField("cloneCounter",0),
                 StrFixedLenField("secureString",42,4), # 4 Bytes
                 ByteField("receiverOption",0),
                 ByteField("payloadLength",0),
                 ByteField("version", 0),
                 ByteField("free",0),
                 IPField("receiverIP", "10.0.0.1"),
                 IPField("receiverMask","255.255.255.0"),
                 StrFixedLenField("standardString",1234,8)]


//inside Scapy --> building headers and combining them into a packet
ethernet_header=Ether(type=0x800, dst="ff:ff:ff:ff:ff:ff")
```

```
ethernet_header=Ether(type=0x800, dst="08:00:00:00:02:22")
--> entweder broadcast (1) oder unicast (2) + no leading ZEROS in ETH
ip_header= IP(dst="10.0.1.2", proto=253,flags="DF")
clone_header = Clone(numberOfClones=10)


// Merging headers :)
packet = ethernet_header / ip_header / clone_header

// send packet
send(packet)
sendp(packet, iface="eth0")
// Show commands
packet.show()
packet.show2()
clone_header.show2()
//bind_layers(IP, Clone, protocol=253)
```

Appendix I: IPv4-Addresses for the laboratory net

Device	Interface (context)	IPv4-Address	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

Table I.1: IPv4-Addresses for the laboratory net

Appendix J: Programme files as .zip

The programs written for this work and associated files from the Git repository are attached as a separate .zip folder (`p4-project-main.zip`).

Abbreviations

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	Programming language 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	Fiber optic cable
MC	Multicast
NPU	Network Processing Unit - Analogy to CPU
OS	Operating System
PRE	Packet Replication Engine
RAM	Random Access Memory
RID	Replication-ID
RJ45	Registered Jack 45 - "LAN cable"
SDN	Software-Defined Networking

SFP ..... Small Form-factor Plugable
SRAM ..... Static random-access memory
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 ..... Virtual Machine
VPN ..... Virtual Private Network

List of Figures

List of Tables

Listings

Bibliography

[BMv2] P4 Language Consortium. *Behavioral Model*. Jan. 2023. URL: https://github.com/p4lang/behavioral-model (visited on 01/26/2023).

[Bos14] Pat Bosshart et al. "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 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 (visited on 01/16/2023).

[IP-NetNS] E. Biedermann. *ip-netns(8)*. Dec. 2022. URL: https://www.man7.org/linux/man-pages/man8/ip-netns.8.html (visited on 01/23/2023).

[IP-Proto-Nr.] Internet Assigned Number Authority (IANA). *Protocol Numbers*. 2022. URL: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml (visited on 01/09/2023).

[MC-Replication-ID] V. Gurevich. *question about multicast #22*. Mar. 2017. URL: https://github.com/p4lang/tutorials/issues/22#issuecomment-289075465 (visited on 01/19/2023).

[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 (visited on 01/07/2023).

[P4 Runtime Spec.] *P4Runtime Specification*. Version 1.3.0. P4.org API Working Group, July 2021. URL: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html (visited on 02/06/2023).

[PSA] *P416 Portable Switch Architecture*. P4 Language Consortium, Apr. 2021. URL: https://p4.org/p4-spec/docs/PSA.html (visited on 01/13/2023).

[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, Sept. 1981. URL: https://www.rfc-editor.org/rfc/rfc791.txt.

[RFC792] J. Postel. *Internet Control Message Protocol*. RFC 792. Internet Engineering Task Force, Sept. 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 (visited on 01/09/2023).

Declaration of Ownership

I hereby certify that I have written this Bachelor Thesis independently and only with the use of the sources and aids indicated. The thesis has not been submitted in the same or similar form to any other examination authority to date.

Berlin, 09.02.2023

Marcel Beausencourt