P4 Language - A practical introduction

Nathan Camiola

CYBER DEFENCE LAB

May 12, 2024

P4 Language - A practical introduction
Nathan Camiola
May 12, 2024

Cyber Defence Lab
https://cylab.be

BibTeX citation:

```
@techreport{citekey,
title = {P4 Language - A practical introduction},
author = {Nathan Camiola},
institution = {Cyber Defence Lab},
year = {2024},
month = {5},
}
```

Abstract

Keywords: Programming Protocol-independent Packet Processors, P4, Software Defined Networking, SDN, Network

Contents

1. Introduction

Programming Protocol-independent Packet Processors (P4) is a domain-specific language for network devices, specifying how data plane devices (switches, NICs, routers, filters, etc.) process packets.

P4 programs are designed to be implementation-independent: they can be compiled against many different types of execution machines such as general-purpose CPUs, FPGAs, system(s)-on-chip, network processors, and ASICs.

When you start setting up a network, you need to look at all the features you want to implement. To do this, we look at how the device's packet-processing chip works to best meet our needs, but there will be many features we don't need. The system will then be built according to what the chip allows us to do. This is what we call bottom-up design. The drawback is that if we want a new feature, we'll have to change the chip or hardware, which can take a long time and/or cost a lot of money [1].

P4 attempts a reverse, top-down design, allowing us to determine exactly how the device does packet-processing using a P4 program compiled directly on the device [1].

P4 does not define the mechanisms by which data is received and transmitted to another system. This differs from system to system and is defined by the system's P4 architecture.

1.1. Getting Started

It's very important to emphasize that the content described in this document relates to the **V1Model** architecture. It is not entirely correct for the **TNA** (Tofino Native Architecture) of the **S9180-32X** switch.

The **code is basically identical, since they use the same language**, but the standard metadata mentioned varies from one architecture to another (we'll come back to this point later). In addition, the TNA architecture requires additional Ingress and Egress control blocks.

If you're reading this document with a view to using TNA, please be aware that it may not be fully adapted to your needs, but it will help you to understand many of the concepts involved in the P4 language.

2. Architecture

The P4 architecture determines the P4-programmable blocks (e.g. Parser, Ingress, etc.) and their data plane interfaces. In other words, the architecture determines which part of the pipeline can be programmed. The P4 architecture can be seen as a link between the program and the switch. It defines what can and cannot be done, and how. The architecture definition must be supplied by each hardware manufacturer (e.g. Intel Tofino), together with a P4 compiler [2].

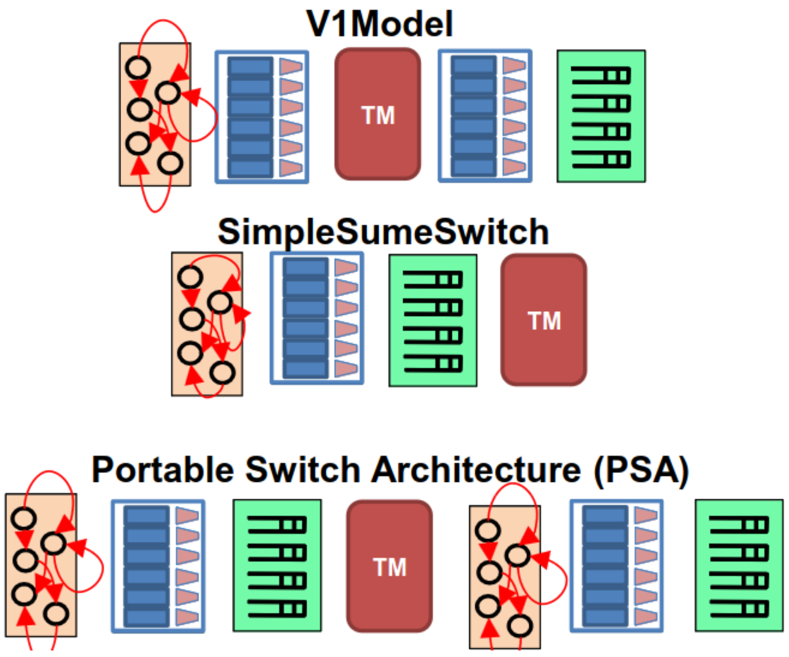

Figure 2.1. P4 Architectures

The package arguments indicate the programmable blocks that have to be instantiated by the user:

Listing 2.1. V1Switch Architecture Example

```
package V1Switch<H,M>(Parser<H,M> p,
                            VerifyChecksum<H,M> vr,
                            Ingress<H,M> ig,
                            Egress<H,M> eg,
                            ComputeChecksum<H,M> ck,
                            Deparser<H> dep
                            );
```

2.1. Some Well-known Architectures

- **V1Model** : The V1Model architecture is commonly used with the BMv2 Simple Switch. It defines the programmable structure of the pipeline [2].
- **PSA** *(Portable Switch Architecture)* : The PSA architecture describes common capabilities of network switch devices that process and forward packets across multiple interface ports [2], [3].
- **TNA** *(Tofino Native Architecture)* : This TNA architecture describes the structure used by the Intel Tofino switch ASICs (Application Specific Integrated Circuits).
- **PNA** *(Portable NIC architecture)* : The PNA architecture describes the structure and common capabilities of network interface controller (NIC) devices that process packets going between one or more interfaces and a host system [2].
- **SimpleSumeSwitch** : The SimpleSumeSwitch architecture describes the structure of used by the NetFPGA SUME board and by the BMv2 Mininet.
- **PISA** *(Protocol Independent Switch Architecture)* : The PISA architecture describes a single pipeline forwarding architecture. $P4_{14}$ targeted PISA-like devices but $P4_{16}$ outgrown PISA [2].

V1Model and TNA are two more or less similar $P4_{16}$ architectures but still different. They both correspond to a switch architecture whose packet flow follows the same pattern `Ingress`, `Traffic Manager` & `Egress`, but they are not identical architectures. Indeed, TNA has a Parser, Match-Action Pipeline and Deparser for the Ingress and Egress pipeline [2], [4].

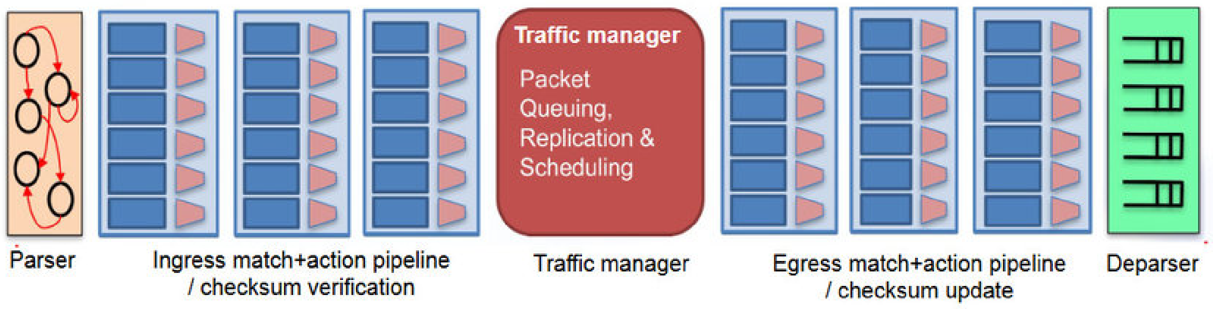
Parser
Ingress match+action pipeline / checksum verification
Traffic manager
Packet Queuing, Replication & Scheduling
Traffic manager
Egress match+action pipeline / checksum update
Deparser

Figure 2.2. V1Model P4 Switch Architecture

On the other hand, architectures such as PNA (Portable NIC Architecture) are even more different from V1Model and TNA, in that way, they are not designed to model a switch ASICs, but a programmable NIC. It doesn't follow the `Ingress`, `TM` & `Egress` pattern at all [2].

In conclusion, P4 programs are not expected to be portable across different architectures, the same program with the same behavior will be written differently depending on its architecture. However, P4 programs written for a given architecture should be portable across all targets that implement the corresponding model, provided there are sufficient resources [2].

3. Programming a P4 Target

P4 is initially based on the high-speed packet-processing device PISA. Basically, P4 programming works as follows [1], [5]:

- A programmable `Parser` block which determines which packets headers will be recognized by the data plane program.
- A `Match-Action Table` programmable block that matches tables of entries and executes actions based on matches.
- A `Deparser` programmable block that recomposes by serializing the last header and metadata into a packet.

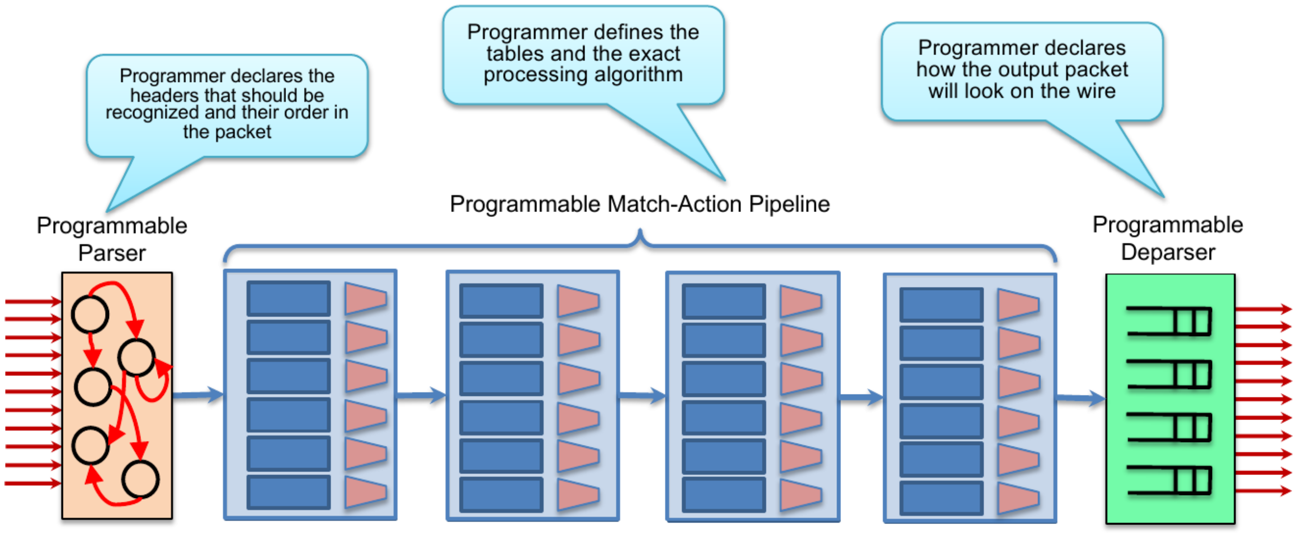
Programmer declares the headers that should be recognized and their order in the packet
Programmer defines the tables and the exact processing algorithm
Programmer declares how the output packet will look on the wire
Programmable Parser
Programmable Match-Action Pipeline
Programmable Deparser

Figure 3.1. Protocol Independant Switch Architecture (PISA)

The target provides the hardware or software implementation framework, a **P4 Architecture Model** and a **P4 Compiler**. The programmer writes a **P4 Program to describe** all the functionality required for each programmable block defined by the Architecture Model provided by the target [2], [5].

The program is compiled by the vendor's P4 Compiler and generates a data plane configuration that implements the forwarding logic described in the input program and an API for managing the state of the data plane objects from the control plane. The programmer also provides the **Control Plane** implementation. The

Control Plane can interact with the Data Plane via runtime mechanisms such as **P4Runtime** [2].

A P4 program defines a packet-processing pipeline, but the rules within each table are inserted by the control plane. When a rule matches a packet, its action is invoked with parameters supplied by the control plane as part of the rule [2].

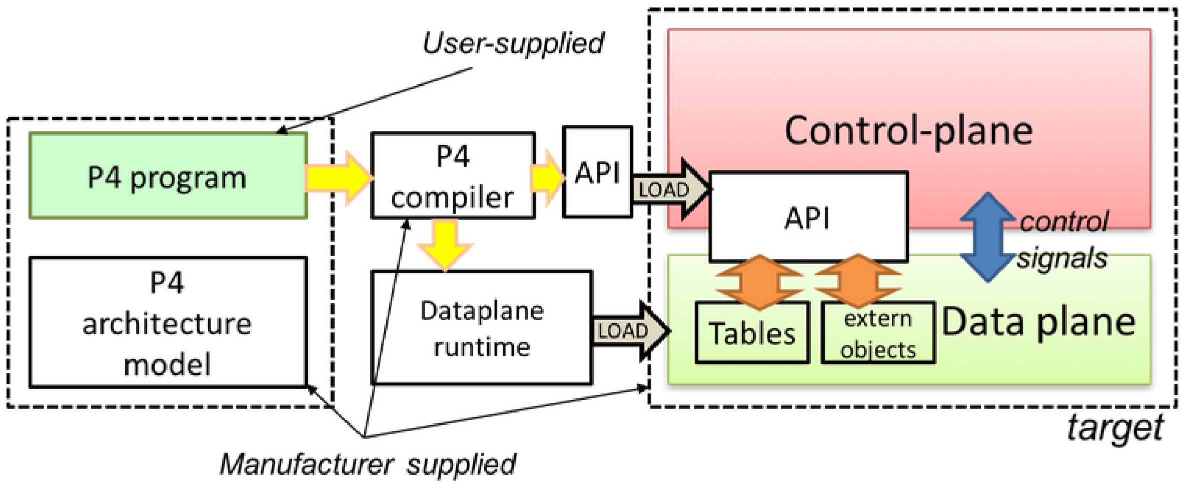

Figure 3.2. P4 Programming

4. P4 Capabilities

The language itself can do only the following things [4]:

- It can convert an external packet representation (usually a byte stream) into a set of parsed headers and metadata using a standardized packet_in extern and its methods.
- It can manipulate headers and metadata using assignments, standard arithmetic and logical operations as well as standard methods defined on headers, like .setValid() or .setInvalid().
- It can perform match-action operations using tables.
- It can further convert some of the headers or metadata into output packets in their external representation using the standardized packet_out extern and its methods.
- It can call methods of the architecture-specific externs (or call architecture specific extern functions).
- It can interact with the architecture-specific fixed-function components using intrinsic metadata, using the same methods as are available to the regular header and metadata fields (variables).

Note, that the language provides no standard way to even drop a packet or to send it to an output port, not to mention sending a packet to several ports (multicasting), creating independent copies of a packet (cloning/mirroring), counting bytes in a packet, enqueueing a packet into a specific queue, etc., etc. (In fact, the language does not have a concept of “port” to begin with). All these and a lot more truly “interesting” things are accomplished using architecture-specific facilities, be they either fixed-function devices and their intrinsic metadata or externs and extern functions [2], [6].

4.1. Layers

The concept of Layers (L2, L3, ...) for P4 switches doesn't really make sense any more as soon as any functionality can be developed insofar as possible. As mentioned above, all possible functionality will be determined by the architecture more than by the capabilities of the language since the language itself can't do much. When we characterise a P4 switch as L2, it means that it doesn't have any L3 functionality implemented, not because it is not possible to implement them from scratch.

5. Some P4 Concepts

5.1. Header definition

Since P4 is protocol-independent, it does not recognize any incoming header. It is therefore necessary to describe each field with their size of the protocol header. So, each different protocol used in the program must be declared.

Listing 5.1. IPv4 header definition

```
header ipv4_t {
    bit<4> version;
    bit<4> ihl;
    bit<8> diffserv;
    bit<16> totalLen;
    bit<16> identification;
    bit<3> flags;
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    bit<32> srcAddr;
    bit<32> dstAddr;
}
```

But since it does not recognize any incoming header, it is also possible to declare custom headers which do not correspond to any existing protocol.

Here an example for a basic tunneling p4 program where a new header type is used to encapsulate the IP packet.

Listing 5.2. Custom header definition

```
header myTunnel_t {
    bit<16> proto_id;
    bit<16> dst_id;
}
```

It will also sometimes be necessary to declare headers of variable size, in which case we use `varbit`. The `varbit` type cannot be used in a header with `bit` fields.

Listing 5.3. Variable TCP options definition

```
header tcp_opt_t {
    varbit<320> options;
}
```

5.2. Parsing & Deparsing

5.2.1. Parser

All parsers start with a `state start` pointing to the first header to be extracted. Then, based on extractions, we `transition` to other states.

Listing 5.4. Parser definition

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata)
                   {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
            packet.extract(hdr.ethernet);
            transition select(hdr.ethernet.etherType) {
                TYPE_IPV4: parse_ipv4;
                default: accept;
            }
        }

        state parse_ipv4 {
            packet.extract(hdr.ipv4);
            transition accept;
        }
    }
```

The parser tries to `extract` the fields included in the header definition. Extractions are performed in the same order as defined in the code. If it succeeds, it sets a validity bit for this header [2], [6].

The `select` keyword is used here to create an *if else* condition (the *if else* as, we know it, cannot be used in parsers but only in the match-action pipeline) [2], [6]. The condition is the **etherType**; if it is equal to **TYPE_IPV4** (which of course had to be declared in the code and equal to `0x0800`), we transition to the `parse_ipv4` state until we `accept` or `reject` the packet.

Parsing variable-size header

Variable size headers must also be extracted. It's a bit different for this than for fixed-size headers. An extraction "condition" will need to be provided, i.e. a way of calculating the precise bits to be extracted.

```
    state parse_tcp {
       packet.extract(hdr.tcp);
       verify(hdr.tcp.do >= 5, error.InvalidTCPpacket);
       packet.extract(hdr.tcp_opt, (((bit<32>)hdr.tcp.do - 5) *
          32));
       transition accept;
    }
```

The first line checks if the Data Offset (do) field in the TCP header is greater than or equal to 5. Note that a data offset is minimum 5 and maximum 15. If the condition is not met, it triggers an error *InvalidTCPpacket*. The verify statement is a mechanism to enforce certain conditions on the packet headers during parsing.

The next line extracts the TCP options field from the packet. The size of the TCP options field is determined based on the value of the Data Offset (do) field in the TCP header. The formula `(((bit<32>)hdr.tcp.do - 5)* 32)` calculates the size of the options field in bits. The cast `(bit<32>)hdr.tcp.do` converts the do field to a 32-bit value, and subtracting 5 represents the minimum size of the TCP header in 32-bit words. Multiplying by 32 converts the size from words to bits. The extracted options are stored in the `hdr.tcp_opt` structure.

5.2.2. Deparser

Nothing could be simpler than deparsing. By calling the `emit` method, we serialize the header only if it is valid (validity bit set to true), inserting all the fields of the new header into a packet. As with extraction, `emit` writes the fields in the order defined in the code [1].

Listing 5.5. Deparser definition

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
        packet.emit(hdr.tcp_opt);
        packet.emit(hdr.udp);
    }
}
```

5.3. Headers operations

There are very simple operations that can be performed on headers to influence their validity. The method `isValid()` verifies the value of the validity bit by returning its value. The methods `setValid()` and `setInvalid()`, as their names suggest, they set the header's validity bit to true and false respectively. These methods can only be applied to l-values.

In P4, "l-values are expressions that may appear on the left side of an assignment operation or as arguments corresponding to out and inout function parameters. An l-value represents a storage reference".

5.3.1. Encapsulation & Decapsulation

Encapsulating and decapsulation headers is very easy with these methods. As mentioned above, a header will only be "added" if its validity bit is set and removed if the validity bit equal 0 (invalid).

Once a header has been "added", the next step is to assign values to the header fields.

Warning: assigning values before setting the valdity bit will not work. If you want to add a header, be sure to set it before assigning values.

5.4. Counter & Meter

5.4.1. Counter

Counters provide a choice of whether to maintain only a packet count (`CounterType.packets`), only a byte count (`CounterType.bytes`), or both (`CounterType.packets_and_bytes`).

The direct counter is associated with at most one table. Every time the table is applied and a table entry is matched, the counter state associated with the matching entry is read, modified, and written back.

Note: counters can only be read from the control plane.

Listing 5.6. Counter and Direct Counter examples

```
counter(64, CounterType.packets_and_bytes) mycounter;
direct_counter(CounterType.packets_and_bytes) mydirectcounter
    ;

action monitoring() {
    mycounter.count((bit<32>) standard_metadata.ingress_port
        );
    mydirectcounter.count();
}
```

A more detailed demonstration can be found on Gitlab.

5.4.2. Meter

In P4, a metadata (often `meta.meta_tag` & `meta.direct_meta_tag`) now holds the color.

The regular meter allows you to specify the number of meter states, where the number of meter states is specified by the size parameter and the type of measurement; based on the number of packets, regardless of their size (`MeterType.packets`), or based upon the number of bytes the packets contain (`MeterType.bytes`).

The direct meter is associated with at most one table. It allows in this case to measure the rate of a flow of a table. Every time the table is applied and a table entry is matched, the meter state associated with the matching entry is read, modified, and written back.

We will set the rates of the CIR and PIR from the controller using P4Runtime [1] :

```
meter_array_set_rates <name> <rate_1>:<burst_1> <rate_2>:<
    burst_2>
```

If the meter type is `MeterType.bytes`, the *rate* unit is bytes/microsecond and *burst_size* in bytes. If the meter type is `MeterType.packets`, the *rate* unit is packets/microsecond and *burst_size* is the number of packets.

Listing 5.7. Meter and Direct Meter examples

```
meter(16384, MeterType.packets) acl_meter;
direct_meter<bit<32>>(MeterType.packets) direct_acl_meter;

action color_packets(bit<32> meter_index) {
    acl_meter.execute_meter<bit<32>>(meter_index, meta.
       meter_tag);
    direct_acl_meter.read(meta.direct_meter_tag);
}
```

A more detailed demonstration can be found on Gitlab.

5.5. Register & Hashes

At first glance, the registers don't seem obvious, or at least that was my impression. However, once understood, registers are very easy to use and very useful.

Registers are often used in conjunction with hashes, usually 5-tuple hashes, to identify sessions [1].

```
register<bit<REGISTER_ENTRIES_BIT_WIDTH>>(REGISTER_ENTRIES)
   my_register;
```

Hashes (CRC-16 and/or CRC-32) are generally used as indexes in the register. And it is to this index that we will associate a value.

Listing 5.8. 5-tuple hash

```
hash(meta.hash, HashAlgorithm.crc16, (bit<32>)0,
    {
        hdr.ipv4.srcAddr,
        hdr.ipv4.dstAddr,
        hdr.tcp.src_port,
        hdr.tcp.dst_port,
        hdr.ipv4.protocol
    },
    (bit<32>)REGISTER_ENTRIES);
```

The *metadata* `meta.hash` is where the hash will be stored. At the moment, the hash isn't in the register.
As explained, we usually use the hash as an index in the register. In the example below, we assign the value "1" to the index *meta.hash* :

```
    my_register.write(meta.hash, 1);
```

A second *metadata* `meta.value_associated_with_the_hash` will be needed to read the content of the register :

```
    my_register.read(meta.value_associated_with_the_hash, meta.
       hash);
```

Note: Reading the contents of an index that doesn't exist (or doesn't exist yet) will output the value 0. In this case, avoid assigning 0 for particular conditions.

5.6. Short summary

5.6.1. Counter

- Function: Counters are used to count and store statistics on the number of events or packets that have satisfied a specific condition. Note that counters can only be read from the control plane.
- Data Type: Counters can store packet count and byte count.
- Modifiability: Counter values can be incremented or decremented during packet processing.
- Typical Use: Counters are often used to gather statistics on specific events, such as the number of received packets, lost packets, etc.

5.6.2. Meter

- Function: Meters are used to measure and control packet flow rates. They are commonly employed for traffic control and rate limiting.

- Data Type: Meters store "rate" data (e.g., number of packets per unit of time).
- Modifiability: Meters measure packet flow rates, and based on defined rules, packets can be marked, allowed to pass, or dropped.
- Typical Use: Meters are used to implement traffic control policies, such as rate limiting for a specific service class.

5.6.3. Register

- Function: Registers are used to store information, indices, or maintain specific states during packet processing. The values associated with registers are kept from one packet to the next.
- Data Type: Registers can store data of various types depending on the specific declaration.
- Modifiability: Register values can be modified during packet processing.
- Typical Use: Registers are used to store information or states necessary for decision-making in packet processing.

5.6.4. Metadata

- Function: Metadata is used to store additional information associated with a packet, and this information can be shared among different control blocks and processing stages. The values associated with metadata are not kept from one packet to the next.
- Data Type: Metadata can store data of different types, including bits, integers, and other user-defined types.
- Modifiability: Metadata values can be modified during packet processing.
- Typical Use: Metadata is used to store information relevant to the context of packet processing, such as network state information, routing details, VLAN tags, etc.

5.7. Multicast & Cloning

5.7.1. Multicast

For packet mutlicasting, the simple standard metadata can be used to assign a group to the packet being processed. The multicasting configuration is then detailed in the control plane.

```
action multicast() {
    standard_metadata.mcast_grp = 1;
}
```

In the **control plane**, you first need to define a multicast group. As explained above, it is this group that will be assigned to the packet processed in the data plane.

```
mc_mgrp_create 1
```

Next, we configure the multicast node. This node takes the replication id (*rid*), which will be directly associated with the packet metadata, as well as the port numbers to be included in the group. It is from these ports that the packet will be multicast.

So, multicast packets can be identified by the standard metadata `egress_rid`.

```
mc_node_create 0 1 2 3 4 5
```

Finally, we simply associate the multicast group with the node.

```
mc_node_associate 1 0
```

5.7.2. Cloning

When you clone a packet with `clone` during ingress or egress processing, the packet will be cloned and sent to egress processing. A cloned packet cannot be modified during ingress processing. The original packet will continue its "normal" processing [1].

```
action clone_packet() {
    const bit<32> REPORT_MIRROR_SESSION_ID = 500;
    // Clone from ingress to egress pipeline
    clone(CloneType.I2E, REPORT_MIRROR_SESSION_ID);
}
```

The control plane has also have to be configured :

```
mirroring_add 500 1
```

The cloned packet will be identifiable from the original, as its standard metatada `instance_type` will be assigned the value 1.

5.8. Compute Checksums

The architecture provides functions for recalculating checksums in the case of modifications.

5.8.1. IPv4 Checksum

Depending on the protocol you want to recalculate the checkum, you need to check what it needs. IPv4 checkums require all header fields.

Listing 5.9. IPv4 Checksum calculation

```
update_checksum(
    hdr.ipv4.isValid(),
        { hdr.ipv4.version,
          hdr.ipv4.ihl,
          hdr.ipv4.diffserv,
          hdr.ipv4.totalLen,
          hdr.ipv4.identification,
          hdr.ipv4.flags,
          hdr.ipv4.fragOffset,
          hdr.ipv4.ttl,
          hdr.ipv4.protocol,
          hdr.ipv4.srcAddr,
          hdr.ipv4.dstAddr },
        hdr.ipv4.hdrChecksum,
        HashAlgorithm.csum16);
```

5.8.2. TCP Checksum

For the TCP protocol, calculating the checksum requires a little more information, but also requires the entire payload. This kind of information can be found in the RFC related to the protocol. The architecture also provides a function for including the payload.

Listing 5.10. TCP Checksum calculation

```
    update_checksum_with_payload(
        hdr.tcp.isValid(),
            {   hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr,
                8w0,
                hdr.ipv4.protocol,
                meta.tcpLength,
                hdr.tcp.src_port,
                hdr.tcp.dst_port,
                hdr.tcp.seq_number,
                hdr.tcp.ack_number,
                hdr.tcp.do,
                hdr.tcp.rsv,
                hdr.tcp.cwr,
                hdr.tcp.ece,
                hdr.tcp.urg,
                hdr.tcp.ack,
                hdr.tcp.psh,
                hdr.tcp.rst,
                hdr.tcp.syn,
                hdr.tcp.fin,
                hdr.tcp.window,
                hdr.tcp.urgPointer,
                hdr.tcp_opt.options },
            hdr.tcp.checksum,
            HashAlgorithm.csum16);
```

You can directly perform the TCP length computation during the IPv4 parsing state.

Listing 5.11. TCP Length field calculation

```
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        meta.tcpLength = hdr.ipv4.totalLen - (bit<16>)hdr.ipv4.
           ihl * 4;
        transition select(hdr.ipv4.protocol){
            TYPE_TCP: parse_tcp;
            TYPE_UDP: parse_udp;
            default: accept;
        }
```

5.9. Remarks

There are other functions and metadata (*enq_qdepth*, *enq_qdepth*, *ingress_global_timesta*
...) that I haven't covered here, or simply haven't had the opportunity to use. It's worth reading all the functions and metadata available in the V1Model architecture source code.

6. Learning Materials

In this section, you'll find a multitude of what I consider to be interesting links concerning the P4.

6.1. Cylab blog posts

- The P4 language - The basics of P4 language
- The P4 language - Overview of P4 programming

6.2. Documentation

- NSG-Ethz - Wiki
- NSG-Ethz - P4-Learning
- NSG-Ethz - BMv2 Simple Switch
- NSG-Ethz - Control Plane
- P4.org - Specifications
- Andy Fingerhut - P4 Guide

6.3. Slides

- NSG-Ethz - 2022 Slides
- $P4_{16}$ Programming for Intel® Tofino™ using Intel P4 Studio™

6.4. Exercices

- Cylab - Examples (V1Model)
- P4Lang - Tutorials (V1Model)
- NSG-Ethz - Exercices (V1Model)
- Zhaoyboo - P4 Tofino Examples (TNA)

6.5. Architectures

- V1Model Source code
- TNA Source code
- $P4_{16}$ Intel® Tofino™ Native Architecture - Public Version

Bibliography

[1] p4lang, *Education/gettingstarted.md at master · p4lang/education*. [Online]. Available: https://github.com/p4lang/education/blob/master/GettingStarted.md.

[2] *P416 language specification*. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.3.pdf.

[3] *Open networking foundation 2023*, Oct. 2023. [Online]. Available: https://opennetworking.org/p4/.

[4] *P4 programming language*. [Online]. Available: https://forum.p4.org/.

[5] F. Hauser, M. Häberle, D. Merling, *et al.*, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103 561, Mar. 2023. DOI: 10.1016/j.jnca.2022.103561.

[6] S. Laki, *Programmable Networks Lecture 2 – P4 basics & lookups*.