ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

MASTER'S DEGREE
IN
TELECOMMUNICATIONS ENGINEERING

DATA PLANE PROGRAMMABILITY FOR DYNAMIC SERVICE PROVISIONING

Master Thesis
in
Laboratory of Advanced Networking M

Supervisor
Prof. GIANLUCA DAVOLI

Candidate
ANNA VANTI

Co-supervisor
CHIARA GRASSELLI

SESSION III
ACADEMIC YEAR 2021/2022

Contents

Abstract **v**

1 Introduction **1**

1.1 Related Work 3

2 Overview **5**

2.1 Network Programmability 5

2.2 Data Plane Programmability 6

2.3 The P4 Programming Language 8

2.3.1 Specification History 9

2.3.2 $P4_{16}$ Design 10

2.3.3 $P4_{16}$ Data Plane Model 12

2.3.4 V1Model Architecture 15

2.3.5 $P4_{16}$ Data Types 17

2.3.6 Parser and Deparser in V1 Architecture 19

2.3.7 Match-Action Pipeline in V1 Architecture 24

2.3.8 Externs 30

2.4 Dynamic Service Provisioning 33

3 P4 Basics and Use Cases **35**

3.1 Automatic Procedure for VM configuration 36

3.2 Implementing Basic Forwarding 38

3.3 Implementing Basic Tunneling 46

3.4 Implementing a Control Plane using P4Runtime 55

3.5 Implementing Explicit Congestion Notification 64
3.6 Implementing Multi-Hop Route Inspection 71
3.7 Implementing A Basic Stateful Firewall 79

4 DoS Prevention Service Implementation 91
4.1 DoS prevention P4 Program 93
4.2 P4Runtime Controller 108
4.3 Interaction with the orchestration system 117

5 Performance Evaluation 119

6 Conclusion 133

Abstract

With a steady progression toward network softwarization, the new generation of communication infrastructures have undergone drastic transformation in recent years. Due to this, networks may now support higher levels of automation, pervasiveness, flexibility, and efficiency. The newest generation of network services must confront with the increasing needs of different application contexts. For instance, it is frequently necessary to provide an extremely tight latency (e.g. IoT and machine-to-machine communications). In contrast to cloud computing, fog computing proposes an intermediary layer between network infrastructure and end users, wherein clusters of fog nodes deliver services in response to service requests. Fog nodes, in addition to computational needs to the network, can offer a highly dynamic ecosystem by joining or removing themselves from the Fog cluster. To handle the many service models in this scenario and the associated resource monitoring and allocation challenges, a service orchestration system is required.

In this thesis, a DoS prevention service is designed and implemented through the P4 programming language and P4Runtime framework, and prepared for integration with a resource management and service provisioning layer placed between the service consumers and a Fog infrastructure. These tools enable the effective softwarization and programmability of the Data Plane of network devices. The complete system providing the DoS prevention service is described in detail in each of the implemented layers, and the results of a performance evaluation are presented.

Chapter 1

Introduction

In the last years, the new generation of communication infrastructures is radically transformed with a progressive evolution toward network softwarization. This brings the networks to enable an increasing level of flexibility, efficiency, pervasiveness, adaptability, and automation. Indeed, the different application contexts have different needs in terms of networking and computing performance, but what combines all of them is the growth of their critical needs. The newest generation of network services must confront with these increasing needs. In many cases, for example, a very tight latency experienced by the end-user or machine-to-machine application has to be ensured. Therefore, latency can be considered one of the most important parameters to take into account for the performance evaluation process of service providers. In particular, with the growth of the Internet of Things (IoT) and machine-to-machine communication, the devices require highly scalable approaches to massive processing and storage [1]. Cloud applications based on microservice design are widespread in softwarized networks for their high scalability and efficient lifecycle manageability. Recently, indeed, the Cloud Computing Paradigm is obtaining great popularity. This approach consists in delegating the major part of networking and computing functionalities to achieve the lowest possible latency and it is often done at the expense of computing power. Fog Computing, instead, proposes an intermediate layer between end-users and Cloud infrastructures, where clusters of Fog nodes provide services that are similar to those offered by its larger counterpart (i.e., Cloud Computing) but focusing on the needs of microservice-based and modular applications [1]. Fog Computing, in addition to a tight latency, allows for having a highly dynamic ecosystem.

Fog nodes, that offer computation power to the network, can join or remove themselves from the Fog cluster. In this way, the overall amount of resources in the different Fog clusters can vary in time according to the specific needs. The distributed and dynamic characteristics of the Fog Computing paradigm differ from the traditional characteristics of the Cloud Computing paradigm, however, the offered services are similar. Therefore the Everything-as-a-Service (XaaS) Cloud service model classification can be easily applied also to Fog Infrastructures keeping the dynamism and the flexibility typical of a Fog environment. In this scenario, a service orchestration system is needed to support the different service models and the related resource monitoring and allocation issues. In particular, the orchestrator should manage the services that are natively offered by a Fog node, and the services that can be configured and deployed, and made available on a Fog node. Considering that, the orchestrator should be able to handle service requests coming from end-users and machine-to-machine applications and to choose which node, if any, can be employed to provide the requested service [1]. Then, it will configure the chosen node according to the specific request, and inform the end-user about the availability of the requested service. In this thesis, a system for Fog ORCHestration (FORCH) is taken into account as a resource management and service provisioning layer placed between the service consumers and a Fog infrastructure [1]. Under this layer, the proposed service (a DoS prevention service) is programmed and implemented through the P4 programming language and P4Runtime framework. These tools enable the effective softwarization and programmability of network devices Data Planes. In the following Chapters, an overview of network programmability and the P4 language is provided, and the implementation of the DoS prevention service is presented in detail. In Chapter 2, after a brief overview of the Network Programmability concept, the P4 language is explained in all its aspects: specification history, data plane model, the specific V1Model Architecture, and some more implementation details. In Chapter 3, some of the P4 tutorials available at the GitHub P4 repository [2] are presented starting from the setup of the work environment to the solutions of the chosen exercises. In Chapter 4, the DoS prevention service is shown in each implementation layer and in Chapter 5 the results of a performance evaluation are presented through graphs and explanations.

1.1 Related Work

In this Section, a variety of works related to SDN and Fog computing are presented, paying close attention to works in which the P4 language is utilized. In [3], is considered the fact that, with the advances of SDN and the P4 language, there are new opportunities and challenges that next-generation SDN has for Fog computing. It is, indeed, proposed a new mechanism of deploying SDN control planes both locally and remotely to attend different challenges, and lower the response time for locally deployed applications (local control plane). A large part of past and current works focuses on the needs of IoT devices and on the services that they can offer. In [4] a detecting data telemetry based on P4 language and a preliminary Fog resource management based on the collected data telemetry is presented to offer latency critical computation and storage in the IoT scenario. Also, in [5], the technical challenges of the design of an efficient, virtualized, context-aware, self-configuring orchestration system for the Fog computing system are taken into account and a prototype of an intelligent self-managed orchestrator for IoT applications and services is proposed. In [6], the focus is on the P4 and P4Runtime to overcome the difficulties imposed by other protocols (such as OpenFlow protocol) in SDN Fog computing. Several works related to P4 language for Data Plane Programming, [7] [8] [9], describes how this programming language, by specifying how the switches should process packets, can reduce the overhead of the interaction between Data Plane and Control Plane and allows a more flexible solution for network monitoring. Moreover, intelligent resource monitoring and dynamic service provisioning are enabled. Finally, in [10], a use case related to DDoS attack prevention is proposed. This paper, indeed, presents microVNF, a Virtualized Network Function (VNF) written in the P4 programming language and running on a programmable Data Plane that aims to detect, prevent, and mitigate DoS attacks directed at the connected IoT devices.

Chapter 2

Overview

As summarized in Chapter 1, this thesis aims at presenting the P4 language for Data Plane Programmability and a simple use case for DoS prevention.
In this Chapter, after a brief presentation of network programmability, the P4 language is introduced with its history, benefits, abstract models, and implementation details. Also, an introduction to dynamic service provisioning achieved through P4 Data Plane Programmability is provided.

2.1 Network Programmability

The programmability of a device can be defined as the ability of the software or the hardware to execute a processing algorithm. Thus, the term network programmability is the ability to define the processing algorithm executed in a network and in the individual processing nodes, such as switches, routers, etc. Programming and reconfiguring the devices allows both the network equipment vendors and the users to build networks ideally suited to their needs. Active and programmable networks allow the creation, customization, deployment, and management of new services or applications that are configured (programmed) dynamically into network nodes. Users can thus utilize these programmable services to achieve their specific network requirements in terms of performance (e.g., throughput and latency), reliability (e.g., fast failover), flexibility and adaptability. The network efficiency is also improved due to the ability to reconfigure, rearrange and adapt the network to the current workload (e.g., by scaling out/in resources in a demand-aware manner) or failover mechanisms (e.g., in terms of fast failure detection, notification, and recovery).

Traditionally, the algorithms executed by network devices, are split into three distinct classes: the data plane, the control plane, and the management plane [11]. In this reference environment, a plane can be described as a group of algorithms that process different kinds of traffic, have different performance requirements, are designed using different methodologies, are implemented using different programming languages, and run on different hardware. The traditional network architecture consists, as introduced above, of three planes: the data plane, the control plane, and the management plane. Every network device must perform all three distinct activities: planning and regulating the forwarding process with the control plane protocols, processing the transit traffic and actual message forwarding in the data plane, and interacting with its owner (or NMS – Network Management System) through the management plane.
Traditional network devices are closed vertically integrated systems and their functionalities are rigidly programmed into the embedded software and hardware. A term commonly used nowadays in the industry is Network Operating System (NOS), an execution environment for programmatic control of the network [12]. Analogously to the read and write access interfaces to various resources provided by Computer Operating Systems, a Network Operating System provides the ability to observe and control a network. The Network Operating System does not manage the network itself; it provides programming interfaces with high-level abstractions of network resources that enable network application programs to perform complicated tasks safely and efficiently on a wide heterogeneity of networking technologies. In many modern high-speed devices, the Data Plane is not yet part of the NOS; having a fixed Data Plane with fixed devices for the different functionalities does not permit a flexible and extendible design of the Management and Control Planes. In order to program and manage the whole stack, the Network Operating System should consist of Control Plane, Management Plane, and Data Plane.

2.2 Data Plane Programmability

As mentioned in Section 2.1, the Data Plane should be part of the Network Operating System and, thus, can be defined and programmed by the users to encounter the specific needs of the network. The Data Plane algorithms are responsible for processing all the packets that pass through a telecommunication system, and they define the functionality, performance, and scalability of such

systems. The possibility to program the Data Plane allows the users to build custom network equipment without any compromise in terms of performance, scalability, speed, or energy consumption [11].
The Data Plane Programmability entails multiple benefits:

- Full control and customization. By programming the Data Plane it is possible to make the device behave exactly in the desired way and it is possible to inspect and monitor the programs to control the behavior of the device.
- Flexibility and ease of adding new features. Data plane programming introduces full flexibility to network packet processing. Algorithms, protocols, and features can be easily added, modified, or removed by the user, by simply changing, adding, removing, or modifying the programs.
- Reliability. With a programmable device, as mentioned above, it is possible to remove underused features in order to reduce the complexity of the network. Having a program that does only what is needed, without any unused functionality, reduces the risk to have bugs and network failures.
- Ease of deployment. Compared to long development circles of new silicon-based solutions, new algorithms can be programmed and deployed in a matter of days.
- Exclusivity and differentiation. There is no need to share information with the chip vendor, therefore it is possible to change the functionality of the network Data Plane without having to ask a third party. Network equipment designers and even users can experiment with new protocols and design unique applications; both are no longer dependent on vendors of specialized packet processing devices to implement custom algorithms. Data plane programming is also beneficial for network equipment developers that can easily create differentiated products despite using the same packet processing device keeping their know-how to themselves without the need to share the details with the chip vendor. Therefore, the possibility of not sharing ideas allows for obtaining a competitive advantage.
- Efficiency. With a fixed device, the features and the resource allocation are defined at design time. If it's used only a part of the implemented

functions, there are a lot of unused resources that are still consuming energy. A programmable device reduces energy consumption and expands scale by dedicating all the available resources just to the processes that are needed or easily turning off unused resources in order not to waste energy. In addition to improving the security and efficiency compared to multi-purpose appliances, including in the code only the needed components for a specific use allows for keeping the complexity low.

- Telemetry. The last advantage of using a programming language is the ability to look inside and inspect the device (e.g. visibility of intermediate results in the processing of packets) [12].

So far, modern Data Plane programs and programming languages have not yet achieved the degree of portability attained by general-purpose programming languages. However, expressing Data Plane algorithms in a high-level language has the potential to make telecommunication systems significantly more target-independent [11].
Moreover, Data Plane programming does not require but encourages full transparency. Sharing the source code, all definitions for protocols, functionalities, and behaviors can be viewed, analyzed, inspected, and tested so that Data Plane programs benefit from community development and review [11].

2.3 The P4 Programming Language

Industry and academia are converging on a new data plane programming language called P4 (Programming Protocol-Independent Packet Processors). The P4 specification is open and public [13]. The P4 language is developed by the p4.org consortium [14], which currently includes more than 60 companies in the area of interest of networking, cloud systems, network chip design, and academic institutions. The P4 Language Consortium is composed of academic and industry members, the scope is to produce an open-source, evolving, domain-specific language.
P4 is a high-level language for programming network devices that allows specifying how data plane devices (switches, NICs, routers, filters, etc.) process packets. Before the advent of the P4 language, chip vendors had total control over the functionalities supported in the network, and since networking silicon determined much of the possible behavior, silicon vendors controlled the rollout of new features, and rollouts could take years. The traditional design of a

network system follows a *bottom-up* approach that starts from a fixed-function chip and makes it difficult to add new features or modify the fixed functionalities. P4 and the Programmable Data Plane concept turn the traditional model upside down. Using a *top-down* approach it is possible to define at a high level exactly how the chip should process packets, which is done by the P4 program. Thus, application developers and network system designers can now use P4 language to implement specific behavior in the network, and changes can be made in minutes instead of years.
Among the several benefits of using the P4 language, one of the most important is, therefore, the ease of adding new features by changing the P4 program and recompiling. In the same way, it is always possible to remove underused features to reduce the complexity of the network or to replace old features with newer and more efficient ones. With P4, it becomes therefore very easy to reallocate resources after the deployment. Programmable devices guarantee greater visibility into the network and give the possibility to have new diagnostic techniques, telemetry techniques, etc.
For the moment being, the P4 Language Design Working Group (LDWG) of the P4 Language Consortium has standardized two distinct standards of P4: $P4_{14}$ and $P4_{16}$.

2.3.1 Specification History

In May 2013 was proposed the initial idea and the name P4, which stands for Programming Protocol-Independent Packet Processors, was coined. In July 2014 the first paper (SIGCOMM ACR) was published.
In August 2014 there was the first $P4_{14}$ draft specification (version 0.9.8) and very soon in September 2014 the $P4_{14}$ specification was released (version 1.0.0). Then, over the following years, many different versions of the P4 specification are released with minor specification revisions and amendments. The language was quite stable and it was supported on several different targets [13]. Around April 2016 the first P4 commits went into the public repositories. In December 2016 the first $P4_{16}$ draft specification for public review was released and in May 2017 the first $P4_{16}$ specification was published [13]. Once $P4_{16}$ started to take shape, the original P4 language became $P4_{14}$ language. The Table 2.1 and the Table 2.2 depict respectively the $P4_{14}$ specification history and the $P4_{16}$ specification history specifying the most important versions and their release dates.

In general, the programming languages that tend to have more independent implementations often use the year in the name, while the programming languages with one dominant implementation tend to use numerical version names. For P4 language, the version naming is based on the year of the original inception and then it takes about a year to finalize the specification.
The references to the language can have two kinds of spelling: official spelling `P4_16` on terminals and $P4_{16}$ in publications.

Table 2.1: Specification history of $P4_{14}$

$P4_{14}$ versions	
Version 1.0.1	01/2015
Version 1.0.2	03/2015
Version 1.0.3	11/2016
Version 1.0.4	05/2017
Version 1.0.5	11/2018

Table 2.2: Specification history of $P4_{16}$

$P4_{16}$ versions	
Version 1.0.0	05/2017
Version 1.1.0	11/2018
Version 1.2.0	10/2019
Version 1.2.1	06/2020
Version 1.2.2	05/2021

2.3.2 $P4_{16}$ Design

$P4_{16}$ is supposed to be the logical evolution of $P4_{14}$ and it has been introduced to address several $P4_{14}$ limitations that became apparent in the course of its use [12]. Those include the lack of means to describe various targets and architectures, weak typing and generally loose semantics, relatively low-level constructs, and weak support for program modularity [11]. The idea behind the $P4_{16}$ approach is basically to separate the notion of a *P4 Target* from the notion of a *P4 Architecture*. A target is the embodiment of a specific hardware implementation, while an architecture provides interfaces to program a

target via some sets of P4 programmable components, externs, and fixed components [12]. Externs are supplied by vendors and provide some functionality that is not directly implemented by the P4 language. They are "black boxes" providing an interface that can interact with or be invoked within P4 programs. A target can implement many different architectures. A *P4 platform* is the P4 architecture implemented on a given P4 target [12].
One of the most important goals of $P4_{16}$ is to be target independent, therefore to support a variety of different targets (ASICs, FPGAs, smart NICs, software targets, etc.), and architecture dependent. The community develops and standardizes the language itself and the core library, so the specialized constructs that every system should implement, and the vendors of the target devices provide the architecture definitions for their platforms and the libraries of externs they support (as shown in Figure 2.1).

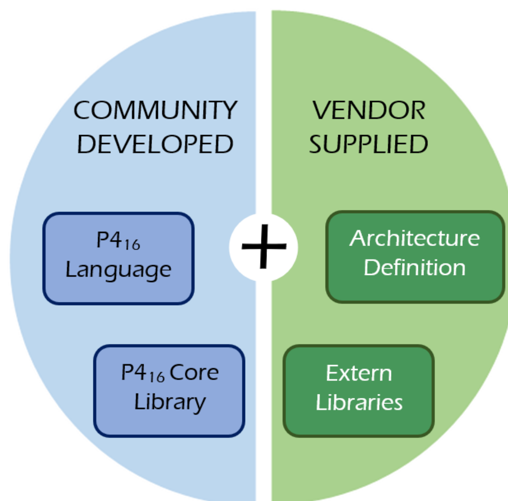


Figure 2.1: $P4_{16}$ Approach: community-developed and vendor-supplied components.

Therefore, support for multiple different targets is the major contribution of the $P4_{16}$ and is achieved by separating the core language from the specifics of a given architecture. The structure, the capabilities, and the interfaces of a specific pipeline are now encapsulated into an architecture description, and the target-specific functions are accessible through an architecture library, typically provided by the target vendor. The core components are further

structured into a small set of language constructs and a $P4_{16}$ core library that is useful for most P4 programs.
Compared to $P4_{14}$, $P4_{16}$ uses the same premises about how the packets are processed in high-speed pipelines and it uses the same or similar concepts and basic building blocks. $P4_{16}$ introduces strict typing, expressions, nested data structures, and several modularity mechanisms. This language is specifically designed to become much more expressive and convenient to use in both reading and writing programs. The semantics of the language is more formally and clearly defined and standardized.
The $P4_{16}$ model is designed to be very flexible in contrast to $P4_{14}$ which has mandatory aspects and implementations in the Data Plane. Moreover, the deployment of $P4_{16}$ doesn't involve the integration of new constructs and doesn't intend to make P4 become a general-purpose programming language.

2.3.3 $P4_{16}$ Data Plane Model

$P4_{16}$ Data Plane Model is based on the abstract model of a high-speed packet processing device, called PISA, which stands for Protocol-Independent Switch Architecture. The PISA model incorporates some basic components:

1. Programmable Parser;
2. Programmable Match-Action Pipeline;
3. Programmable Deparser.

Firstly, the programmable parser allows programmers to specify the format of the packet that should be processed. It determines which packet headers will be recognized by the data plane program and their order in the packets. It is designed as a simple, deterministic state machine, that processes packet data and identifies headers. Essentially, it converts the stream of bytes into the parsed representation, that is the set of headers. The transitions between different states are typically performed by looking at specific fields in the previously identified headers and, depending on the extracted value, the parser transitions to a specific state.
The central portion of the PISA architecture is a pipeline, consisting of a set of Match-Action stages. It is where the actual algorithms are executed. Programmers can define the tables and the exact processing algorithm through which some data are matched against a table containing several entries and

the corresponding action is executed.
At the end of the pipeline, there is a programmable deparser that performs the operation which is a reverse of the parsing. It re-assembles the packets back (e.g., by serializing the headers) into the stream of bytes. Programmers can, therefore, declare how the output packet will look at the end of the processing. These components are connected through the so-called Metadata Bus, which carries the intermediate results from one stage to the following one, as shown in Figure 2.2.

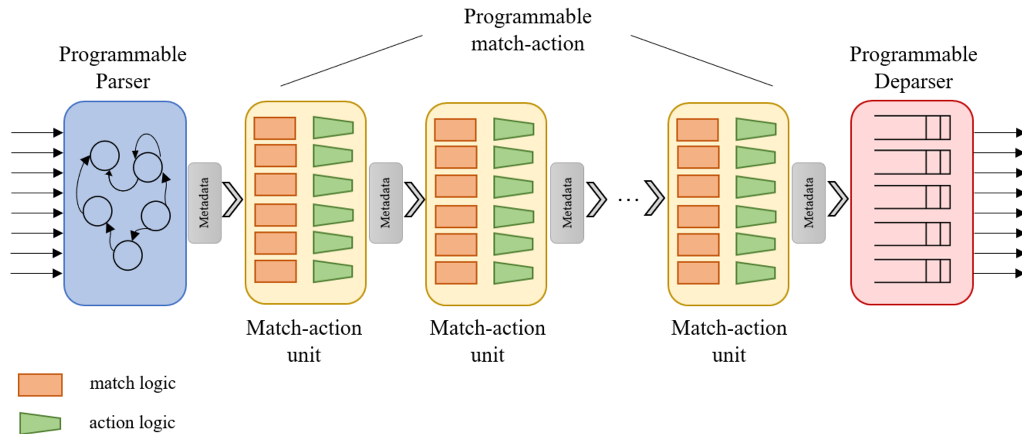


Figure 2.2: Protocol-Independent Switch Architecture (PISA).

These three components are analyzed more in detail in the following sections: the programmable parser and deparser in Section 2.3.6 and the programmable match-action pipeline in Section 2.3.7.
The way PISA operates is quite straightforwar, a basic processing scheme can be summarised in a few steps.

1. The packet first arrives at the parser and is split into individual headers, according to the parser program.

2. The extracted headers are passed to the programmable match-action pipeline that matches the headers against match-action tables and performs a variety of operations. For example, it can remove or add headers; it can freely move the data between the headers, the intermediate results, and the tables; it can perform both simple arithmetic operations or more specialized operations; etc.

3. Every time the match-action operation occurs, intermediate results are generated and they can be used for the matching and by the actions in the following stages.

4. This procedure goes on for as many stages as there are in the pipeline. Intermediate results and metadata are transformed after each stage and the output of the last stage of the match-action pipeline corresponds to the headers that will be present in the packet after the processing.

5. The deparser serializes the headers into the packet byte stream.

Several extended non-standard pipeline models can be implemented as the presented programmable components combined in different ways for different needs. For example, two parsers can be very useful for tunnel processing or other types of processing. Figure 2.3 shows an extended pipeline model with two different parsers; one of them is placed between two consecutive match-action stages and, therefore, it works on the intermediate metadata results.

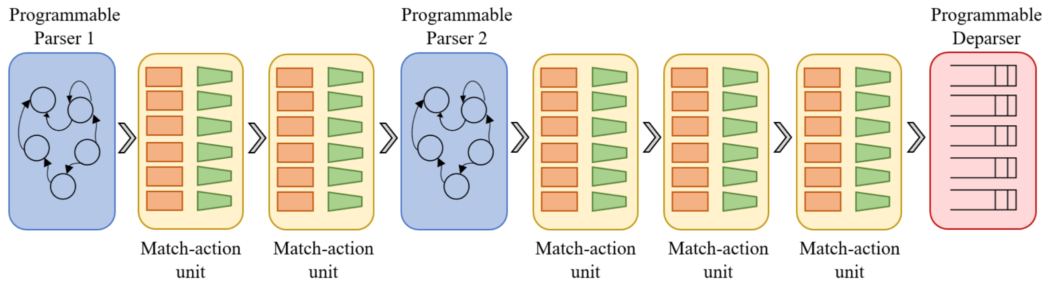


Figure 2.3: Extended Non-Standard Pipeline Model.

Moreover, specialized components are available to be added inside the stages of the pipeline to perform advanced functionalities. In Figure 2.4 these components are represented through several different shapes and different colors to make it easy to understand that they are architecture-specific (in contrast to the general pipeline model) and that each one of them is used for a specific functionality or a specific operation.

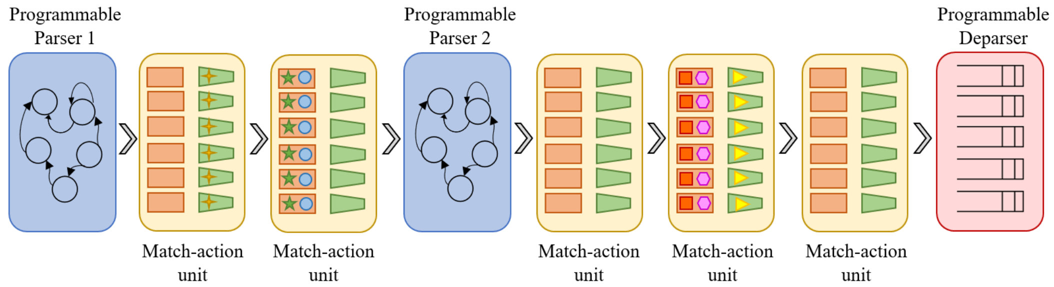


Figure 2.4: Extended Model with Specialized Components.

2.3.4 V1Model Architecture

The V1Model architecture is not the only architecture that can be supported by the BMv2 (Behavioral Model version 2) switch and does not exploit all the possible capabilities of $P4_{16}$. The V1Model architecture is included in the p4c compiler and was designed to be as much identical as possible to the $P4_{14}$ switch architecture, enabling ease of portability of P4 programs from $P4_{14}$ to $P4_{16}$. The presence of just a few differences between $P4_{14}$ and V1Model (e.g., names of metadata fields) makes this architecture compatible with the $P4_{14}$ architecture.
The V1Model architecture consists of the following components (also depicted in Figure 2.5):

- A programmable parser that extracts the packet headers;
- A programmable deparser that reconstructs the packet;
- Separate ingress and egress pipelines of match-action processing;
- A traffic manager that performs the task of packet queuing, scheduling, and replication between the input and the output.

$P4_{16}$ has some categories of language elements that allow programming flexible pipelines. A set of data types (bitstring, headers, structures, arrays, etc.) and expressions (basic operations and operators) allows describing how the data are transferred from one stage to the following one.
Parsers and Controls are the main programmable blocks that represent the actual data flow. Parsers are implemented as state machines and their purpose

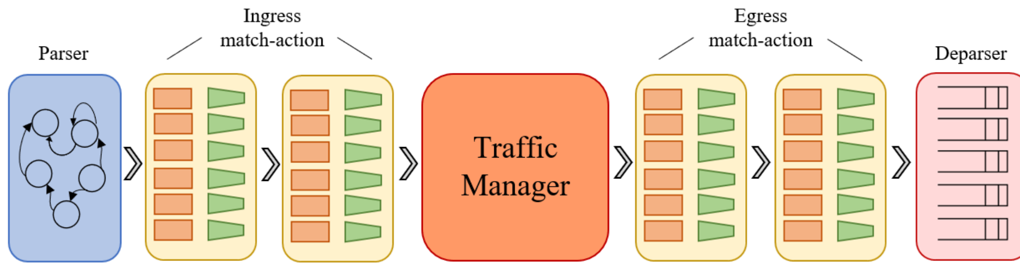


Figure 2.5: V1Model Architecture.

is to extract header fields from the packet in some user-defined way. Controls are responsible for the main match-action processing and they are the actual blocks where tables and actions are defined and applied to manipulate packet headers and metadata.
The architecture description and eventually the extern libraries are also part of these categories of language and are provided by the vendors of the P4 programmable targets. The architecture description specifies the specific configuration of programmable parsers, control blocks, and vendor-specific blocks. Extern libreries offer some functionality that is not implemented in P4 language and provide interfaces that can interact with or be invoked by P4 programs. The architecture description also includes a set of standard metadata fields that are used by the P4 program to direct the packet through the BMv2 switch. The code below shows the standard metadata defined in the V1Model Architecture [15]. The most commonly used fields are the `ingress_port` (that indicates the port on which the packets arrive), the `egress_spec` (that indicates the port to which the packet should be sent), the `egress_port` (that indicates the port that the packet will be sent out of and is only read in egress pipeline), etc.

```
/* V1Model standard metadata,
   defined in v1model.p4 */

struct standard_metadata_t {
   bit<9>   ingress_port;
   bit<9>   egress_spec;
   bit<9>   egress_port;
   bit<32>  clone_spec;
   bit<32>  instance_type;
   bit<1>   drop;
```

```
    bit<16> recirculate_port;
    bit<32> packet_length;
    bit<32> enq_timestamp;
    bit<19> enq_qdepth;
    bit<32> deq_timedelta;
    bit<19> deq_qdepth;
    bit<48> ingress_global_timestamp;
    bit<32> lf_field_list;
    bit<16> mcast_grp;
    bit<1>  resubmit_flag;
    bit<16> egress_rid;
    bit<1>  checksum_error;
}
```

2.3.5 $P4_{16}$ Data Types

$P4_{16}$ language, adopting the PISA approach, uses the concept of packet metadata. Packet metadata can be divided into packet headers, user-defined metadata, and intrinsic metadata.

- Packet headers correspond to the network protocol headers and are extracted during the parsing process.
- Intrinsic metadata are related to the fixed-function components and are used by P4 programmable components to control their behavior.
- User-defined metadata are a kind of temporary storage, similar to local variables in common programming languages. They are used by the developers to pass pieces of information through the different stages of the processing pipeline.

Figure 2.6 illustrates the information flow in the $P4_{16}$ processing pipeline that comprises different blocks. Packet metadata (packet headers, user-defined metadata, or intrinsic metadata) are used to pass the information between consecutive processing blocks.
$P4_{16}$ language supports a rich set of data types for data plane programming, that can be split into basic data types and derived data types [16]. Concerning the first typology, $P4_{16}$ includes common basic types such as boolean (`bool`), unsigned integer of arbitrary length (`bit<n>`), signed integer of arbitrary length (`int<n>`), etc. It is possible to perform many different operations on unsigned

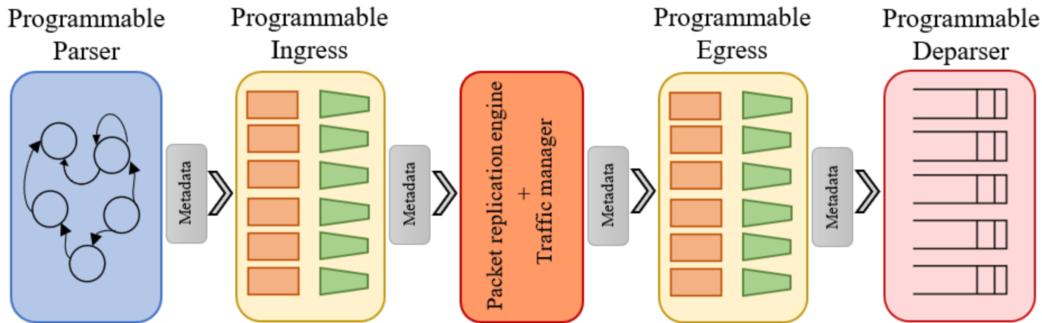


Figure 2.6: V1Model Architecture Information Flow.

integers such as addition, subtraction and concatenation while signed integers, simply called `int` in P4, support some, but not all, of the same basic operations. $P4_{16}$ includes also a bitstring type of variable length (`varbit<n>`, with `n` the maximum possible width of the bitstring) that is mostly used for those protocols that contain fields whose width is only known at runtime (e.g., the Options field in the IPv4 protocol). Finally, $P4_{16}$ also supports enumeration types that can be serializable or non-serializable, where the type representation is chosen by the compiler and hidden from the user.
Basic data types can be composed to construct derived data types. One of the most commonly used and more important derived data types is the header data type. It consists of an ordered collection of fields of the serializable types previously described and it should be byte-aligned (the total length should be a multiple of 8 bits). A header can be valid or invalid depending on whether the header is part of the parsed packet. The validity field of the header is accessible through methods such as `setValid()`, `setInvalid()`, and `isValid()`. Thus, P4 programs can add and remove headers by manipulating their validity bits. The code below shows an example of declaration of Ethernet and IPv4 headers in a P4 program [12]. The `typedef` statement can be used to make the code more readable by giving alternative names to some types. In the example is possible to see how the fields of the headers are represented through the different available data types.

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
```

```
    macAddr_t   dstAddr;
    macAddr_t   srcAddr;
    bit<16>     etherType;
}

header ipv4_t {
    bit<4>         version;
    bit<4>         ihl;
    bit<8>         diffserv;
    bit<16>        totalLen;
    bit<16>        identification;
    bit<3>         flags;
    bit<13>        flagOffset;
    bit<8>         ttl;
    bit<8>         protocol;
    bit<16>        hdrChecksum;
    ipv4Addr_t     srcAddr;
    ipv4Addr_t     dstAddr;
}
```

Another derived data type is the `struct`. Struct in $P4_{16}$ is an unordered collection of fields, it does not have alignment restrictions and it can contain any basic or derived data type including other structs, headers, and other types. Structs can be used as intrinsic metadata to pass a set of data (e.g., a collection of headers) from one component to another.
The last derived data type is the header stack which is an array of headers and is typically used to define repeating headers. It supports several kinds of operations to "push" headers onto the stack or to "pop" them from it.

2.3.6 Parser and Deparser in V1 Architecture

The basic idea of a parser is to extract from an input byte stream a set of header data and metadata. Parsers are defined as Finite State Machines (FSM) with three predefined states, an explicit *Start* state and two ending states (*Accept* and *Reject*), to which custom states are added in between. On this last point, P4 programmers can define custom non-mandatory states and it is common to define one state for each header type that the parser will extract.
The process of parsing always starts in the *Start* state and then transitions to other states, in which information from the packet is extracted according to the defined header structure, until it encounters either the *Accept* or *Reject* state. The state transitions may be either conditional, the transition depends

on the value of an extracted field, or unconditional. The *Reject* state can be reached explicitly or implicitly and it is defined by the architecture. Loops are acceptable within the parser.
Figure 2.7 shows a scheme of the Finite State Machine with *Start*, *Accept*, and *Reject* states and all the possible user-defined states in between.

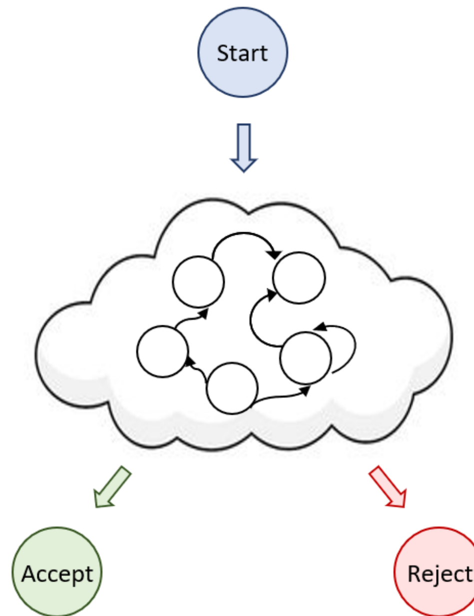


Figure 2.7: Parser: Finite State Machine Scheme.

In all the possible architectures, there are two mandatory arguments in the parser implementation: the input packet (`packet_in`) and the struct of the parsed headers (`hdr`). The first one is an input parameter, while the second one is an output parameter. The user metadata and standard metadata also pass through the parser and can be used if needed; in particular, the V1 Architecture allows having, as additional input passed to the parser, the `ingress_port`. Figure 2.8 shows a scheme of the Parser with specified all the input and output data while in the code below is presented an example of parser implementation [12]. In the example, the *Start* state is defined and it involves several different transitions based on the `etherType` field of the Ethernet header and the default transition to the *Accept* state. Moreover, a state relative to each transition should be defined (`parse_vlan_tag`, `parse_ipv4`,

parse_ipv6, etc.).

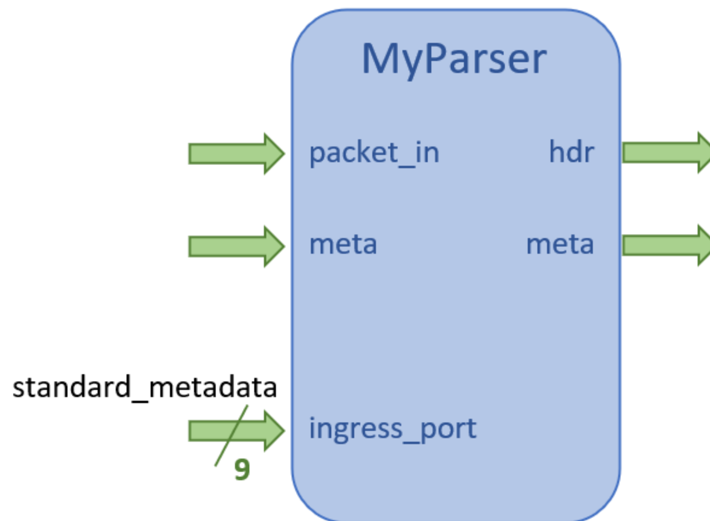


Figure 2.8: Parser Model.

```
parser MyParser (packet_in              packet,
                 out headers_t          hdr,
                 inout metadata_t       meta,
                 in standard_metadata   standard_metadata)  {

    /* states definition*/

    state start {
        packet.extract(hdr.ethernet);
        transition select (hdr.ethernet.etherType) {
        0x8100 &&& 0xEFFF : parse_vlan_tag;
        0x0800  :  parse_ipv4;
        0x86DD  :  parse_ipv6;
        0x0806  :  parse_arp;
        default :  accept;
    }
    ...

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

```
        transition select(hdr.ipv4.protocol) {
            6 : parse_tcp;
            17 : parse_udp;
            default : accept;
        }
    }

    state parse_ipv6 {   }

    ...
}
```

`Packet_in` is an extern defined in the core library (`core.p4`). The output parameter `hdr` is the headers structure made of all parsed headers, `metadata` is a parameter both in the input and in the output of the parser and, finally, the additional input `standard_metadata` is the mentioned above ingress port. The `extract` method is the simplest parsing method and allows the extraction of a fixed-size header from the packet. There are, anyway, other more advanced parsing methods. To design parsers with many states is often used the `select` statement, similar to a `case` statement in C or Java languages. Parsers usually branch based on the value of the just extracted header fields (e.g., the Ethernet `etherType` field represents the protocol encapsulated in the payload of the Ethernet frame and, therefore, it is possible to distinguish between the different parsing states depending on its value).
It is a good practice to implement in the parser a header verification algorithm that can detect if the header is incorrectly formed and doesn't follow the protocol rules and then transitions directly to the *Reject* state. In order to do that, in the core library, is defined a data type called `error` which indicates all the parser errors (a semplified example is shown in the code below [16]). Some standard conditions are defined in the core library, but P4 programmers are allowed to improve this type by adding new exceptions and error codes.

```
/* Standard errors, defined in core.p4 */

error {
    NoError,            // no error
    PacketTooShort,     // not enough bits in the packet for
                           extract
    NoMatch,            // match expression has no matches
    StackOutOfBounds,   // reference to an invalid element of
                           a header stack
    OverwritingHeader,  // one header is extracted twice
```

```
    HeaderTooShort,      // extracting too many bits in a varbit
                            field
    ParserTimeout        // parser execution time limit exceeded
}
```

Packet deparsing is a much simpler procedure. When the packet processing performed by the match-action pipeline is finished, the deparser serializes the packet. It reassembles the packet header and payload back into a byte stream, in this way the packet can be sent out via an egress port or stored in a buffer. Only the headers set to valid are added to the packet.
There are two mandatory arguments in the deparser implementation: the output packet (`packet_out`) and the struct of the valid headers (`hdr`). In the code below is presented an example of deparser implementation [12].

```
control MyDeparser (packet_out      packet,
                    in headers_t    hdr)
{
    apply {
        /* Layer 2 */
        packet.emit(hdr.ethernet);
        packet.emit(hdr.vlan_tag);

        /* Layer 2.5 */
        packet.emit(hdr.mpls);

        /* Layer 3 */
        packet.emit(hdr.arp);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.ipv6);

        /* Layer 4 */
        packet.emit(hdr.icmp);
        packet.emit(hdr.tcp);
        packet.emit(hdr.udp);
    }
}
```

`Packet_out` is an extern defined in the core library (`core.p4`) and it represents the byte stream that the deparser will assemble. The headers struct `hdr` in this case is an input parameter.
The `emit` method is defined in the core library and serializes the header only if it is valid, i.e. if the header is invalid or if it is not in the packet nothing will happen (e.g., in layer 3 only one between ARP, IPv4, and IPv6 header is

emitted).
The deparser is defined as a control block, a construct detailed in the next Subsection 2.3.7.
In $P4_{16}$ the parser and the deparser are completely decoupled. The output packet can be, therefore, assembled independently from how the parser extracted the packet headers.

2.3.7 Match-Action Pipeline in V1 Architecture

The match-action pipeline is executed after the successful parsing of a packet and it expresses the packet processing algorithm [11].
It consists of three elements:

- control blocks;
- actions;
- match-action tables.

In $P4_{16}$ control blocks, or just control, are similar to C functions but without loops and recursion. The algorithms of a control block, indeed, should be representable as Direct Acyclic Graph (DAG), i.e. are implemented through a sequential execution. Not only match-action pipelines but also deparsers and additional processing such as checksum updates are representable as DAGs [12].
Controls are interfaced with other blocks via either user-defined metadata or intrinsic architecture-defined metadata. Figure 2.9 represents the V1 Architecture control interfaces. Two parameters are both in input and output: headers and user-defined metadata. Then there are some input standard metadata:

- `ingress_port`,
- `ingress_timestamp`,
- `resubmit_flag`,

and there are standard metadata that serve as output:

- `egress_spec`,
- `egress_queue`,

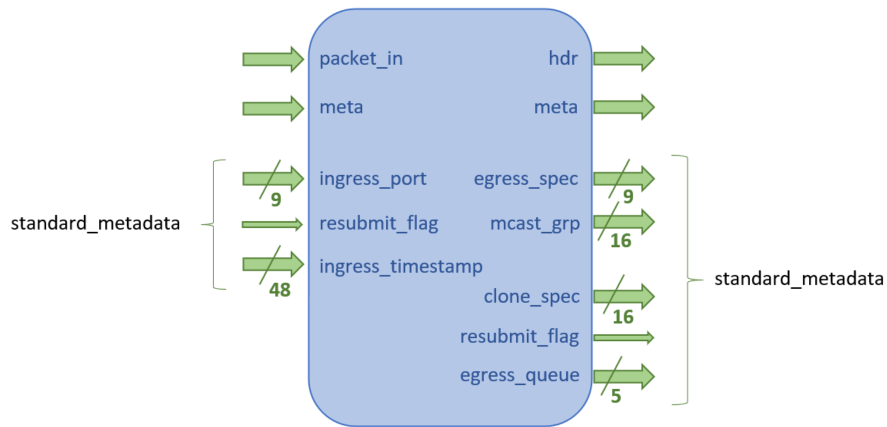


Figure 2.9: Control Block Scheme Interfaces.

- `mcast_grp` (the multicast group),
- `clone_spec`,
- `resubmit_flag`.

The body of a control block contains the definition of variables, tables, actions, and externs that will be used for processing. Then, they are called by an `apply()` method.

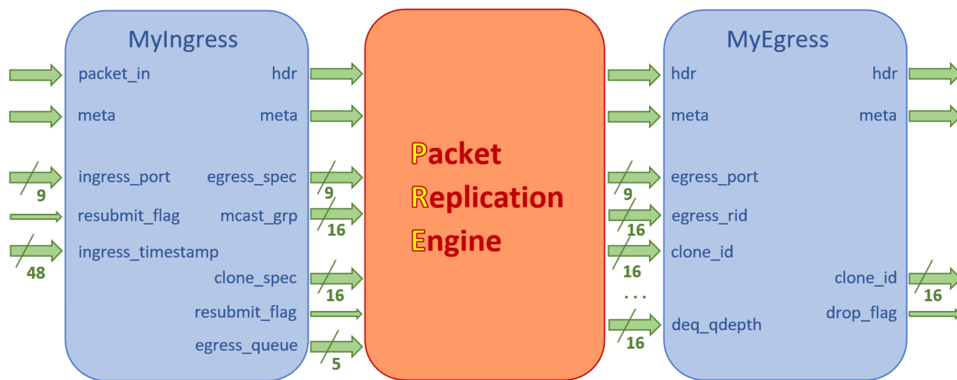


Figure 2.10: V1Model Architecture Pipeline Scheme.

In V1Model Architecture the pipeline can be represented as in Figure 2.10 with an ingress pipeline, an egress pipeline, and a Packet Replication Engine component in between.
The functionality of the PRE (Packet Replication Engine) component is not a matter of P4 implementation, but it is described in the documentation of the specific architecture and the interfaces of this component are standardized.
Actions are functions that can read and write packet headers and metadata. They are similar to functions in other general-purpose programming languages but they have no return value. Actions can be declared locally inside a control block or globally having, in this way, a different scope. In the V1Model Architecture library are already defined several actions (e.g., `mark_to_drop` is the action to drop packets).
Standard arithmetic and logical operations are supported:

- +, -, * ;
- &, |, >>, << ;
- ==, !=, >, >=, <, <= ;
- no division/modulo (in order not to have exceptions);
- bit manipulation operations
  - bit slicing;
  - bit concatenation.

In actions, the main objective is to operate on headers in several ways:

- header validity bit manipulation through the methods such as `setValid`, `setInvalid`, and `isValid` that, respectively, allows to add, remove or check a header;
- header assignment from tuples or another header
  `header =  f1, f2, ..., fn`
  `header1 = header2`
- several special operations on header stacks. For example, `push_front()` and `pop_front()` methods allow to push or pop a single label into the stack and shift all the others.

The match-action tables (MATs) are the fundamental units of the match-action pipeline and they are defined within control blocks. The structure of a match-action table is declared in the P4 program, while its entries are added by the control plane at runtime.
The declaration of a MAT specifies what to match on (match key and match kind), a list of possible actions, and additional properties such as the size of the table (e.g., the maximum number of entries that can be stored in the table) or the default action that will be executed if there is no match.
The match key consists of one or more header and metadata fields, each with its match kind specified. The P4 core library (`core.p4`) contains three standard match kind declarations: exact match, ternary match, and longest prefix matching (LPM). The `match_kind` is a special type in P4 (definition below) [16].

```
/* core.p4 */

match_kind {
    exact,
    ternary,
    lpm
}
```

Different architectures may support additional kind declarations into the `match_kind` type (in the architecture description files), and the V1Model Architecture (`v1model.p4`) extends the list of match types with the range and the selector types [12] as shown in the code below.

```
/* v1model.p4 */

match_kind {
    range,
    selector
}
```

The list of actions includes the IDs and names of all the actions that can be invoked by the table. Each table consists of one or more entries and each entry contains a specific key to match on, the action to be invoked, and optional action data that serve as parameters for the action invocation. Besides the entries, usually in tables are specified also the default action and the default action data that will be used in case no match is found in the table.
Figure 2.11 illustrates the match-action table structure and the match-action

processing. For each packet, specific fields from the chosen set of metadata

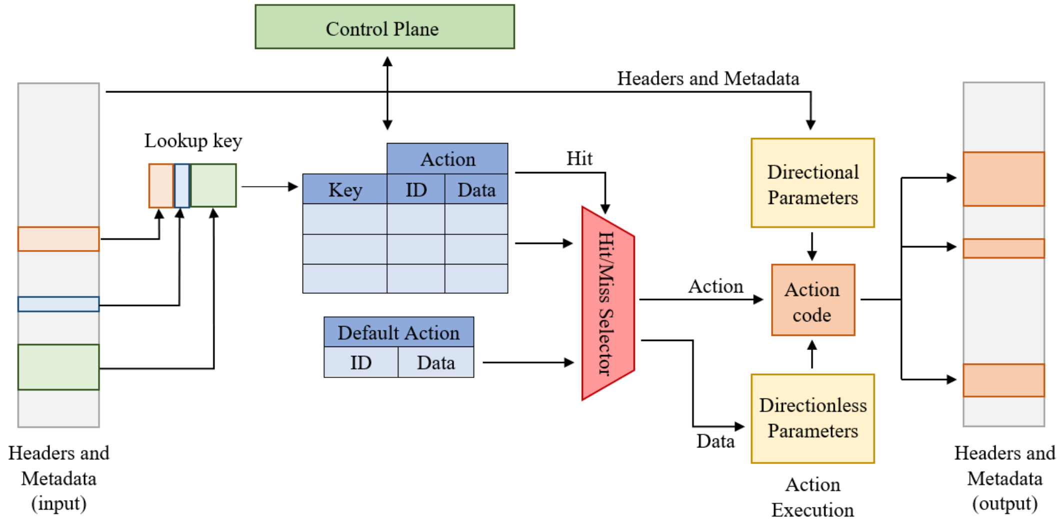


Figure 2.11: Match-Action Processing.

and headers are selected to construct the lookup key that is matched against all the entries of the match-action table. When the a match is found, the table is hit, the corresponding action is invoked and the action data are passed to the action as parameters to be used. If no match is found, the default action is applied. The action execution unit receives two types of data: directionless parameters that are provided directly by the table entries (therefore from the control plane) and directional parameters that consist of headers and metadata (therefore from the data plane). Finally, a new intermediate result is created and is passed to the following match-action stage of the pipeline [12].
In a control block, tables and actions are declared at the beginning, then there is the apply block that essentially performs the match-action processing through the tables `apply()` method. When the match-action is performed by this method, two results are generated: a *meta-result* that specifies if the table found the match or not (a boolean value representing the hit) and the ID of the chosen action.
In the following piece of code, it is possible to see an example of a Ingress Processing control block called `MyIngress` [12].

```
control MyIngress (inout headers_t     hdr,
                   inout metadata_t    meta,
```

```
                   inout standard_metadata_t standard_metadata)
{
    /* action and tables declaration */

    action l3_switch(   bit<9>  port,
                        bit<48> new_mac_da,
                        bit<48> new_mac_sa,
                        bit<12> new_vlan)    {

        /* Forward the packet to the specified port */
        standard_metadata.metadata.egress_spec = port;

        /* L2 Modifications */
        hdr.ethernet.dstAddr = new_mac_da;
        hdr.ethernet.srcAddr = mac_sa;
        hdr.vlan_tag[0].vlanid = new_vlan;

        /* IP header modification (TTL decrement) */
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    action l3_l2_switch(bit<9> port) {
        standard_metadata.metadata.egress_spec = port;
    }

    action l3_drop() {
        mark_to_drop();
    }

    ...

    table ipv4_host {
        key = { meta.ingress_metadata.vrf : exact;
                hdr.ipv4.dstAddr : exact:
        }
        actions = { l3_switch;  l3_l2_switch;
                    l3_drop;    noAction;
        }
        default_Action = noAction();
        size = 65536;
    }

    table ipv6_host {...}
    ...
```

```
    /* apply code */
    apply {
        if(hdr.ipv4.isValid()) {
            ipv4_host.apply();
        }
        ...
    }
}
```

2.3.8 Externs

As described in Section 2.3.4, $P4_{16}$ introduces the concept of P4 architectures as an intermediate layer between the core P4 language and the targets. P4 programs are developed for a specific P4 architecture and they can be deployed on all targets that implement the same P4 architecture [11].
Each architecture can provide additional functionalities that are not part of the P4 language core and with which the P4 programs can interact. Such functionalities are made available by using the `extern` construct, which describes the interfaces that are exposed to the data plane.
In $P4_{16}$, in contrast to $P4_{14}$, all the specialized objects and facilities are removed from the core language to have a unified mechanism to express them all through standardized interfaces. In $P4_{14}$, almost one-third of all the constructs were dedicated to specialized processing.
In P4 there are two types of objects: stateless and stateful objects. The first ones are reinitialized for each packet (e.g., variables, packet headers, packet_in, packet_out, etc.).

```
/* From core.p4 */

extern packet_in {
    void extract<T>(out T hdr);
    void extract<T>(out T variableSizeHeader,
                    in bit<32> variableFieldSizeInBits);
    T lookahead<T>();
    void advance(in bit<32> sizeInBits);
    bit<32> length();
}
```

The code above shows the definition of the `packet_in` type from the P4 core library. This type has several methods that are useful in writing the parser [16]. The externs belong to the second category, the stateful objects, along with the tables. Indeed, they keep their state between packets. Some examples of widely used externs are counters, meters, registers, selectors, etc.
In the following code is shown an example of a counter definition (V1 architecture) [12].

```
/* definition in v1model.p4 */

extern counter {
    counter(bit<32> instance_count, CounterType type);
    void count(in bit<32> index);
}

enum CounterType {
    packets,
    bytes,
    packets_and_bytes
}
```

The extern definition always contains the instantiation method, which has the same name as the extern and it can take some parameters, and other methods to access the extern. In the case of the counter, there is a single method, `count()`, without any return value.
The code below shows an example of a register definition (V1Model architecture) [12]. It includes, as explained before, the instantiation method and the `read` and `write` methods to access a value in the register at the specified index.

```
/* Definition in v1model.p4 */

extern register<T> {
    register(bit<32> instance_count);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}
```

Externs are part of the architecture definition, but they can be used even if their implementation is not known. Most of them must be explicitly instantiated in P4 programs and then other methods are provided to be invoked on the given extern instance.
In $P4_{16}$ tables can have architecture-specific properties and it is possible to

use specific externs, for example the `direct_counter`, to extend them. Using the extern `direct_counter`, quite similar to the previously described counter, it is possible to invoke the method `count()` within the actions related to the table to count packets or bytes of a given type.
In V1Model architecture, there are also a few stateless externs, such as hashes (`hash`) and random numbers (`random`), that can be used to perform some custom computations. The code below shows the description oh the extern `hash` and the enumeration type `HashAlgorithm` that lists the possible available hash algorithms.

```
/* Definition in v1model.p4 */

enum HashAlgorithm {
    crc32,
    crc32_custom,
    crc16,
    crc16_custom,
    random,
    identity
}

extern void hash<T, D>( out T result,
                        in HashAlgorithm algo,
                        in T base,
                        in D data,
                        in T max);
```

This extern allows to calculate a hash function of the value specified by the `data` parameter. The value written into the output parameter named `result` must be in the range `[base, base + max-1]` inclusive.
Finally the example below shows the definition of the extern `random` in the V1Model Architecture. It allows geterating a random number in the range `[lo, hi]` inclusive and writing it in the `result` parameter.

```
/* Definition in v1model.p4 */

extern void random<T>(out T result, in T lo, in T hi);
```

2.4 Dynamic Service Provisioning

Fog Computing, with respect to typical Cloud solutions, can facilitate the adoption of the Xaas (Everything-as-a-Service) paradigm extending it towards a more dynamic and adaptable scenario. The Network Programmability and the Data Plane Programmability play an important role in enhancing the flexibility of the network. Indeed, with the advances of SDN (Software Defined Networking) and the P4 language, there are new challenges and opportunities that the new generation SDN has for Fog computing. In particular, P4 is a language to define the Data Plane behavior for forwarding devices, and therefore it is possible to define how packets are parsed, how they are processed (customized matching criteria and actions), and how they are sent again into the network. The possibility to develop new pipelines allows the network devices to change their behavior and extend their functionalities, and the control-to-data plane programming protocols (e.g. P4Runtime) allows to perform these changes in a dynamic way depending on the needed services. P4 language can be a good option in a Fog scenario for the following reasons:

- Customized protocols can be utilized. The P4 language allows the definition of both standardized and custom protocols. The definition of the headers, tables, and actions, indeed, provides the means to build custom protocols.
- Protocol translation. As a consequence of being able to program which headers are parsed and which actions are executed, the pipeline can also enable and disable headers on demand [3].
- Monitoring. Having a custom data plane enables new telemetry protocols (e.g. Inband telemetry (INT) [17]) to monitor the behavior and performance of packets within P4-programmable devices [3]. Moreover, to monitor the data plane, information via metadata are available at each unit of the programmable pipeline. This can be a beneficial use case to monitor packets at the Fog nodes, exporting information to Fog managers.
- Data plane partition. Being able to create tables (define maximum entry size, keys to match, actions and parameters, etc.) also brings new ways to distribute data plane functionality [3]. For example, one table can be used to prevent possible attacks while other tables can perform L2/L3

forwarding. This concept is tightly related to the use case presented in Chapter 4.

A system that consists of a Fog infrastructure, a Fog Orcheatration System able to act as a resource management and service provisioning layer placed in between the service consumers and the Fog infrastructure [1], and a programmable Data Plane that can be reconfigured and managed in a simple and dynamic way, should be therefore capable to perform a dynamic service provisioning.

Chapter 3

P4 Basics and Use Cases

Reference implementations for compilers, simulation, and debugging tools are available at the GitHub P4 repository [2]. The public repository contains a rich set of tutorials and exercises. These tutorials are created as hands-on laboratories and contain everything needed to experiment in P4: an overview of the language, sets of exercises that include different aspects of P4 data plane programmability including lecture slides and references to useful documentation, and a Virtual Machine with all the needed software.
A set of exercises is proposed to get started with P4 programming. These exercises are organized into several modules that increase in complexity and some of them are described in the following sections.

1. Introduction and Language Basics
   - Basic Forwarding [Section 3.2];
   - Basic Tunneling [Section 3.3];
2. P4Runtime and the Control Plane
   - P4Runtime [Section 3.4];
3. Monitoring and Debugging
   - Explicit Congestion Notification [Section 3.5];
   - Multi-Hop Route Inspection [Section 3.6];
4. Advanced Behavior
   - Source Routing;

- Calculator;
- Load Balancing;
- Quality of Service;

5. Stateful Packet Processing

    - Firewall [Section 3.7];
    - Link Monitoring;

Several pre-installed tools are already present in the available Virtual Machine and are used while performing the tutorials: the BMv2 software switch, the p4c reference P4 compiler, and the network emulation environment Mininet. The BMv2 (Behavioral Model version 2) framework is a software switch developed by the P4 Language Consortium that allows network developers to implement, test and debug several P4 data plane and control plane programs [18]. The p4c is a reference compiler for the P4 language, it supports both $P4_{14}$ and $P4_{16}$. It provides a standard frontend and "mid-end" which can be combined with a target-specific backend that can be easily added or removed at the need [19]. Mininet is a network emulator through which it is possible to create a network of virtual hosts, switches, controllers, and links. The Mininet CLI (and API) allows to easily interact with the network, therefore Mininet is a useful tool for the development, testing, and debugging in a complete virtual experimental network. Mininet is also a great way to develop, share, and experiment with Software-Defined Networking (SDN) systems using OpenFlow and P4 [20].
In the tutorials described in the following sections, it is used a specific architecture which is implemented on top of the BMv2 switch targets, the V1Model Architecture (explained in detail in Section 2.3.4). The available documentation on the repository, indeed, includes the commented file `v1model.p4`.

3.1 Automatic Procedure for VM configuration

To perform the P4 exercises, a Virtual Machine with all the useful software is needed. The Github P4 repository [2] contains all the tools that allow performing an automatic configuration of the P4 Virtual Machine using Git, Vagrant, and VirtualBox.

- *Git* is an open-source Distributed Version Control System (DVCS) used to track changes and modifications of the public repository. It allows downloading a local copy of the last version of the code.

- *Vagrant* is an open-source software that allows for building, maintaining, and managing development environments. The focus is on automation trying to simplify the configuration of reproducible and portable work environments. All the software requirements are written in the so called *Vagrantfile* and, through the command `vagrant up`, all the steps needed to create a development-ready machine are executed.
  In this case, the *Vagrantfile* contains all the configurations that are necessary to automatically build the P4 Virtual Machine. It runs some scripts, contained in the repository, to load the whole environment. In particular, after configuring the Virtual Machine through the box `bento/ubuntu-20.04`, the script `root-common-bootstrap.sh` creates the two users and the script `user-common-bootstrap.sh` installs Mininet, downloads the repository with all tutorial files and some other applications that can be useful during the development of all the exercises.

- *VirtualBox* is a virtualization tool that runs on Windows, Linux, and several other hosts and supports a large number of guest operating systems. Each guest OS can be managed independently within its Virtual Machine.

The first step, after the installation of Vagrant and VirtualBox, is cloning the repository simply using the following command:

```
$ git clone https://github.com/p4lang/tutorials
```

Then, the building and the configuration of the effective Virtual Machine are performed by running in the right directory the command:

```
$ vagrant up
```

This command creates a Virtual Machine that includes P4 software installed from pre-compiled packages and all the necessary dependencies. In this way, it is possible to directly perform all the exercises.

3.2 Implementing Basic Forwarding

The first exercise aims to write a P4 program that implements basic forwarding. For simplicity, just the IPv4 forwarding is taken into consideration [21].
Figure 3.1 represents the topology used in the exercise. It is a single pod of a fat-tree topology and it is described in the file `topology.json`.

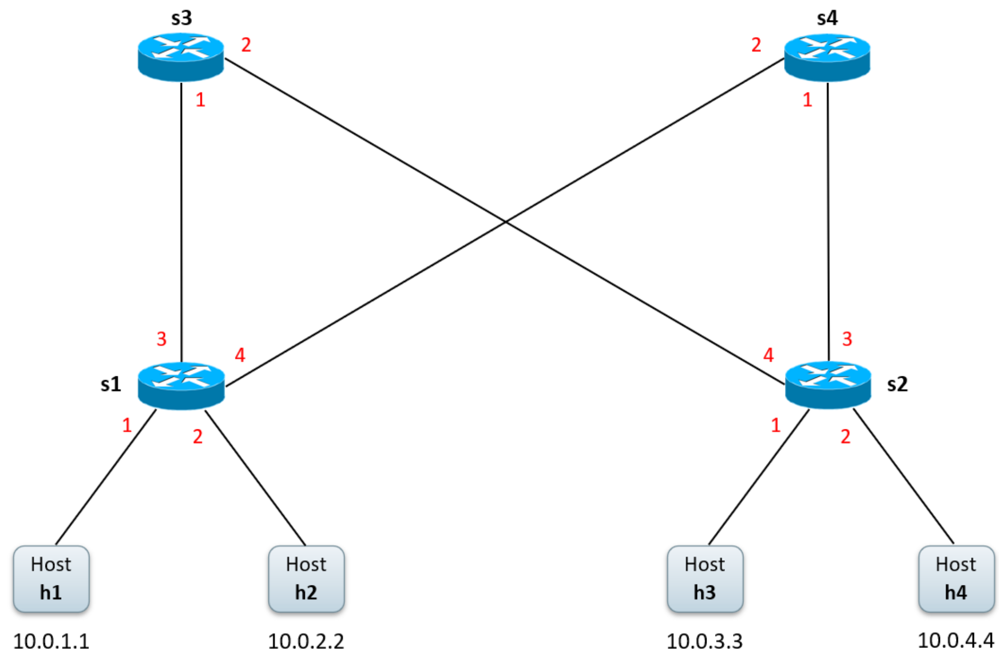


Figure 3.1: Network topology of the tutorial *Implementing Basic Forwarding*.

Concerning IPv4 forwarding, the switch has to perform the following actions for each incoming packet:

- updating the source and destination MAC addresses in the IP header of the packet;
- decrementing the tome-to-live (TTL) field in the IP header of the packet;
- forwarding the packet through the appropriate output port.

The data plane and control plane work together to perform IP routing. The P4 data plane defines the format of the routing table, it specifies fields to match on and the possible actions to invoke. It performs the table lookup and it executes the chosen action. The control plane is responsible for populating the table with entries.
To build this simple IP forwarding, a table must be defined in each switch. The control plane populates the table with static rules and each rule maps an IP address to the MAC address and the output port for the next hop. The control plane rules are defined in the files `sX-runtime.json` (where `X` corresponds to each switch number. `X`=1, 2, 3, 4).
A skeleton P4 program, `basic.p4`, is available to be extended and modified to implement the data plane logic of forwarding IPv4 packets. The initial behavior implemented in the predisposed file is to drop all packets. It is possible to compile the incomplete `basic.p4` file and bring up the topology of Figure 3.1 with Mininet and test this initial behavior.
The command:

```
$ make run
```

compiles the `basic.p4` file and starts the Mininet topology configuring all the switches with the P4 program installed on and all the hosts with the appropriate network configurations. The command also installs the packet-processing static rules in the tables of each switch.
Through the Mininet CLI, it is possible to make some tests.

```
mininet > h1 ping h2
mininet > pingall
```

Given that the test is executed with the not-yet-complete `basic.p4` file, the ping fails because each switch is programmed according to the initial simple behavior which drops all packets on arrival.
The complete `basic.p4` file contains the following components:

- header type definitions for Ethernet (`ethernet_t`) and IPv4 (`ipv4_t`);
- parsers for Ethernet and IPv4 that populate `ethernet_t` and `ipv4_t` fields;
- an action (`drop`) to drop a packet. It can be done using `mark_to_drop()`, a primitive action defined in the file `v1model.p4` that modifies the field `standard_metadata.egress_spec` to an implementation-specific value

that causes the packet to be dropped at the end of ingress or egress processing;

- an action (`ipv4_forward`) that sets the egress port for the next hop, updates the Ethernet destination address with the address of the next hop, updates the Ethernet source address with the address of the switch, and decrements the time-to-live;
- a control that defines a table (`ipv4_lpm`) that matches on an IPv4 destination address and invokes either `drop` or `ipv4_forward`, and an `apply` block that applies the table;
- a deparser that selects the order of the fields inserted into the outgoing packet [21].

The Ethernet and IPv4 headers (`ethernet_t` and `ipv4_t`) are defined at the beginning of the P4 program and they are added to the headers struct.

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

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;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```

The implementation of the checksum computation block is provided in the starter code, so the parser, the ingress control logic, and the deparser need to be implemented. This example does not need any egress control block processing.
Parsers must always start in the *Start* state, then a state for the Ethernet header parsing, as well as a state for the IPv4 header parsing (`parse_ethernet` and `parse_ipv4`), have to be defined. The incoming packets are parsed starting from the Ethernet header, therefore the first transition is from the *Start* state to the `parse_ethernet` state. In this second state (`parse_ethernet`), the Ethernet header is extracted and the transition branches to different states based on the `etherType` field of the Ethernet header using the select statement. If the `etherType` field is equal to the `IPV4_TYPE` value defined at the top of the file, then it transitions to the `parse_ipv4` state, otherwise, the packet does not contain an IPv4 header so it transitions to the *Accept* state (default). In the `parse_IPv4` state, the IPv4 header is extracted and, finally, the transition to the *Accept* state is performed.
The code below shows the parser implementation.

```
const bit<16> TYPE_IPV4 = 0x800;

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;
    }
}
```

Figure 3.2 represents the scheme of the parser in this exercise as a Finite State Machine (FSM).

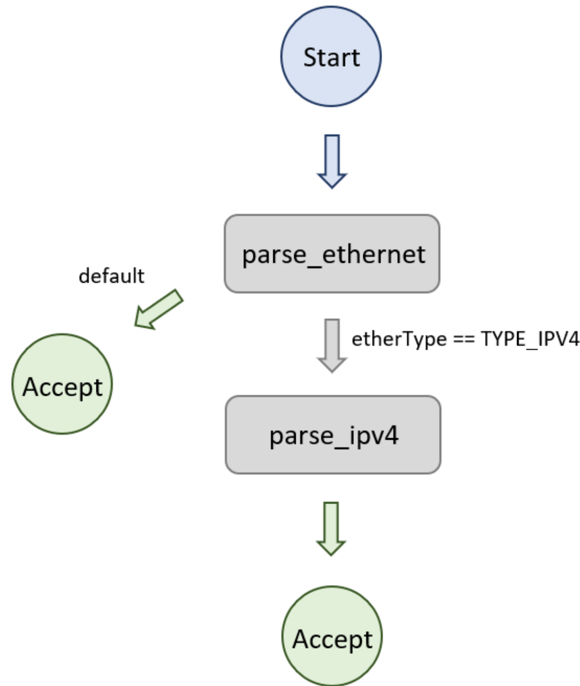


Figure 3.2: Parser state diagram.

In the Ingress Processing control block, the table `ipv4_lpm` is defined. The match key is the IPv4 destination address and the match type is lpm (longest prefix match). The second table property is the list of all possible actions that can be invoked. In this case, the list includes three possible actions: `ipv4_forward`, `drop`, and `NoAction`. `NoAction` is defined in `core.p4`. In the table implementation the table size is also specified and the `default_action` should be `NoAction`.
The code below is the implementation of the `ipv4_lpm` table.

```
table ipv4_lpm {
    key = {
        hdr.ipv4.dstAddr: lpm;
    }
    actions = {
        ipv4_forward;
```

```
        drop;
        NoAction;
    }
    size = 1024;
    Default_action = NoAction();
}
```

For example, the table of the switch s1, once the static rules are installed, can be represented as follows (Figure 3.3).

	Action	
Key	Action	Action Data
10.0.1.1/32	ipv4_forward	dstAddr = 08:00:00:00:01:11 port = 1
10.0.2.2/32	ipv4_forward	dstAddr = 08:00:00:00:02:22 port = 2
10.0.3.3/32	ipv4_forward	dstAddr = 08:00:00:00:03:00 port = 3
10.0.4.4/32	ipv4_forward	dstAddr = 08:00:00:00:04:00 port = 4
	Default Action	
	NoAction	

Figure 3.3: Forwarding table of switch s1.

The `ipv4_forward` action must perform the following operations:

- setting the egress port to the port number provided by the control plane. In the exercise, this is done by setting the `egress_spec` field of the `standard_metadata`;

```
standard_metadata.egress_spec = port;
```

- updating the packet's source MAC address with the MAC address of the switch (packet's current destination MAC address);

```
hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
```

- updating the packet's destination MAC address with the address of the next hop, so the address provided by the control plane;

  ```
  hdr.ethernet.dstAddr = dstAddr;
  ```

- decrementing the TTL field of the ipv4 header.

  ```
  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
  ```

The code below shows the `ipv4_forward` action complete implementation.

```
action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

At the end of the Ingress Processing control block the validity of the IPv4 header is verified and the table `ipv4_lpm` is applied.

```
apply {
    if (hdr.ipv4.isValid()) {
        ipv4_lpm.apply();
    }
}
```

For the implementation of the deparsing logic, the only thing to do is to emit the Ethernet header and the IPv4 header into the packet in the right order as shown in the code below.

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

By completing the `basic.p4` file, a fully functional data plane for basic IPv4 routing is correctly implemented.

The simple IP router is, indeed, able to perform the following functions:

- parsing the Ethernet and IPv4 headers from the packet;
- matching the destination address in the IPv4 routing table;
- setting the egress port;
- updating the source and destination MAC addresses;
- decrementing the TTL;
- deparsing the headers to construct the final packet.

Running the final `basic.p4` file, it is possible to successfully perform ping between any two hosts in the topology. It is also possible to use the two available python scripts `receive.py` and `send.py` that allow sending a message from one host to each other host showing all connection details (addresses, TTL, etc.).

3.3 Implementing Basic Tunneling

Tunneling, also known as "port forwarding", is a communication protocol that allows for the secure movement of data from one private network to another and it allows private network communications to be sent across a public network, such as the Internet, through a process called *encapsulation*. Tunneling is done by encapsulating the private data and protocol information within the public network transmission units, in this way the exclusive information appears to the public network simply as data. Therefore, through the encapsulation process, the private data packets can appear as though they are public and they can pass through a public network unnoticed.
In the second tutorial, the support for a basic tunneling protocol should be added to the IP router implemented in the previous assignment. The basic switch forwards packets based on the destination IP address but, in this case, a new protocol is added and a new header type is defined (`myTunnel`) to encapsulate the IP packets and modify the switch behavior. Figure 3.4 shows

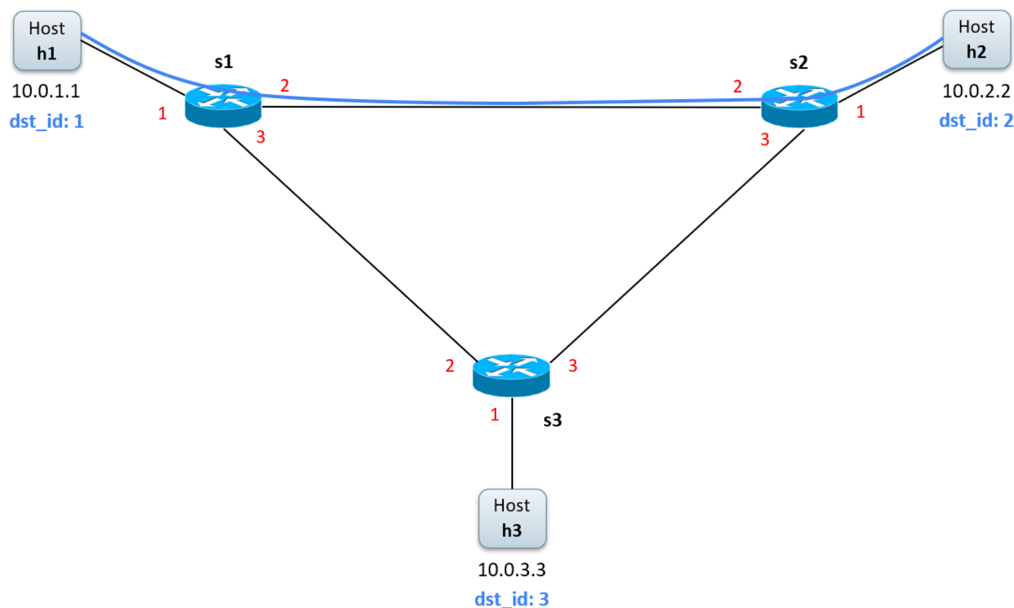


Figure 3.4: Network topology of the tutorial *Implementing Basic Tunneling*.

the topology set up by Mininet in this exercise. It consists of three switches connected in a triangle topology and one host attached to each switch.
The `myTunnel` header should include two fields:

- the `proto_id` field is used to determine the type of packet being encapsulated. In this exercise, only IPv4 packets are taken into account;
- the `dst_id` field is the ID of the destination host.

The new behavior of the switch should be the following: if an encapsulated IP packet arrives, it should perform forwarding using the destination ID field in the myTunnel header; if a normal unencapsulated IP packet arrives, it should perform IP routing as usual.
The available `basic_tunnel.p4` file contains the implementation of the basic IP router while the new functionalities should be added. The complete implementation of the `basic_tunnel.p4` switch should be able to forward based on the content of the custom encapsulation header as well as perform normal IP forwarding if the encapsulation header does not exist in the packet [22].
More in detail, the implementation should contain:

- a new header type, called `myTunnel_t`, that contains two 16-bit fields: `proto_id` and `dst_id`. The myTunnel header should then be added to the headers struct.
- an updated parser able to extract either the myTunnel header or IPv4 header based on the `etherType` field in the Ethernet header. The `etherType` corresponding to the myTunnel header is `0x1212` and it is defined at the beginning of the file. If the `proto_id` field of the mytunnel header is equal to the `TYPE_IPV4` value, the parser should also extract the IPv4 header.
- a new action `myTunnel_forward` that sets the egress port to the port number provided by the control plane.
- a new table `myTunnel_exact` that performs an exact match on the `dst_id` field of the myTunnel header. This table should invoke either the `myTunnel_forward` action if there is a match in the table or the `drop` action otherwise.

- an updated apply statement in the Ingress Processing control block that applies the newly defined `myTunnel_exact` table if the myTunnel header is valid, otherwise, it invokes the `ipv4_lpm` table if the IPv4 header is valid.
- An updated deparser that emits the Ethernet, myTunnel, and IPv4 headers.
- the static rules for the new table so that the switches can correctly forward packets for each possible value of `dst_id` [22].

The new type (`TYPE_MYTUNNEL`) is defined at the beginning of the P4 file together with the definition of the IPv4 type (`TYPE_IPV4`) and the new header type (`myTunnel_t`) is defined and it is added to the headers struct.

```
const bit<16> TYPE_MYTUNNEL = 0x1212;
const bit<16> TYPE_IPV4 = 0x800;

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

struct headers {
    ethernet_t   ethernet;
    myTunnel_t   myTunnel;
    ipv4_t       ipv4;
}
```

The parser should be able to recognize, besides the Ethernet and the IPv4 headers, the mytunnel header if it's present in the incoming packet. An extra case to the select statement in the `parse_ethernet` state is added: if the `etherType` field is equal to `TYPE_MYTUNNEL`, the transition should be to the state `parse_myTunnel`. In the code below, the `parse_ethernet` state is shown.

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

The new state called `parse_myTunnel` is defined and, in this state, the myTunnel header is extracted. After extracting the mytunnel header, the select statement is used to transition to either the `parse_ipv4` state or the *Accept* state based on the value of the `proto_id` field. In the code below, the `parse_myTunnel` state is shown.

```
state parse_myTunnel {
    packet.extract(hdr.myTunnel);
    transition select(hdr.myTunnel.proto_id) {
        TYPE_IPV4 : parse_ipv4;
        default : accept;
    }
}
```

Figure 3.5 represents the scheme of the parser in this exercise as a finite state machine (FSM).

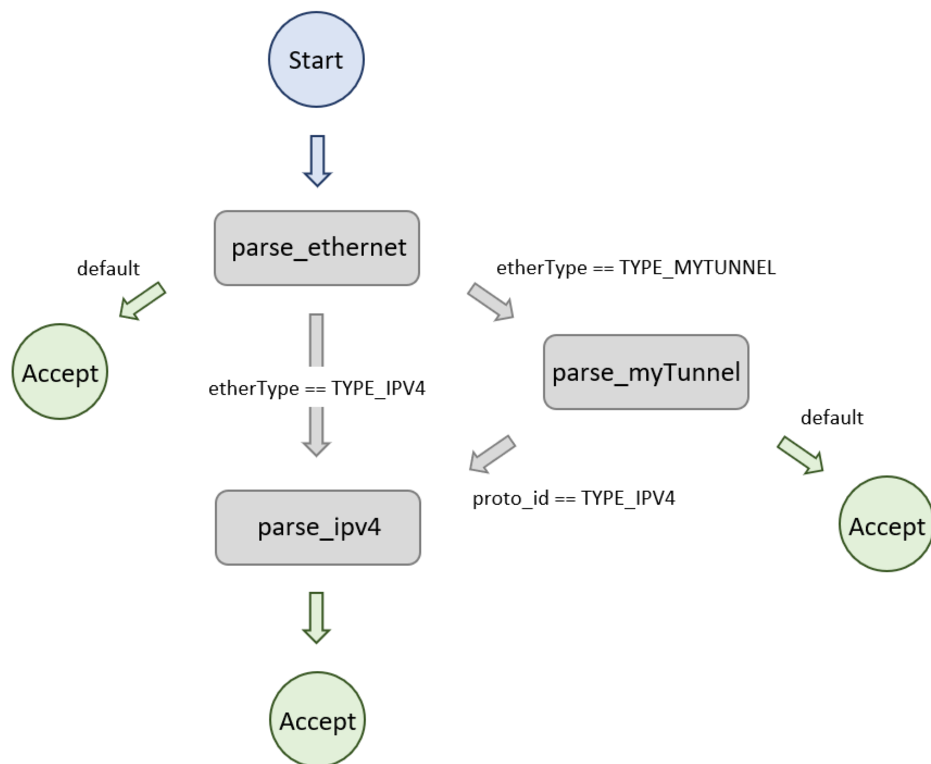


Figure 3.5: Parser state diagram.

The Ingress Processing control block is updated by adding the action `myTunnel_forward` and the table `myTunnel_exact`. The new table `myTunnel_exact` matches on the `dst_id` field of the myTunnel header with an exact match. The list of possible actions includes the new action `myTunnel_forward`, the `drop` action, and `NoAction`. The `myTunnel_forward` action looks very similar to the `ipv4_forward` action but it is simpler. It simply receives an egress port number from the control plane and set the standard metadata `egress_spec` field to be equal to the provided port number. In the code below both the `myTunnel_forward` action and the `myTunnel_exact` table are shown.

```
action myTunnel_forward(egressSpec_t port) {
    standard_metadata.egress_spec = port;
}

table myTunnel_exact {
    key = {
        hdr.myTunnel.dst_id: exact;
    }
    actions = {
        myTunnel_forward;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = drop();
}
```

Also, the control flow is updated: the `ipv4_lpm` table is applied if the IPv4 header is valid and the myTunnel header is not valid (processing only the non-tunneled IPv4 packets) and the new table is applied whenever the myTunnel header is valid (processing only the tunneled packets).

```
apply {
    if (hdr.ipv4.isValid() && !hdr.myTunnel.isValid()) {
        ipv4_lpm.apply();
    }
    if (hdr.myTunnel.isValid()) {
        myTunnel_exact.apply();
    }
}
```

The updated deparser emits the myTunnel header into the packet after the Ethernet header and before the ipv4 header.

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

Some static rules are added in the files `sX-runtime.json` (one for each switch, `X`=1, 2, 3) where also the static forwarding rules of the previous exercise are defined. These new static rules are then inserted in the new tables by the control plane so that the switches can correctly forward packets for each possible value of `dst_id`. Figure 3.6 shows the IPv4 forwarding table and tunneling table of switch s1 and the code below describes the static rules that are written in the file `s1-runtime.json` and that are added to these tables: the first three are forwarding rules and the other three are tunneling rules.

```
{
  "target": "bmv2",
  "p4info": "build/basic_tunnel.p4.p4info.txt",
  "bmv2_json": "build/basic_tunnel.json",
  "table_entries": [
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:01:11",
        "port": 1
      }
    },
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.2.2", 32]
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:02:00",
        "port": 2
      }
```

```
    },
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.3.3", 32]
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:03:00",
        "port": 3
      }
    },
    {
      "table": "MyIngress.myTunnel_exact",
      "match": {
        "hdr.myTunnel.dst_id": [1]
      },
      "action_name": "MyIngress.myTunnel_forward",
      "action_params": {
        "port": 1
      }
    },
    {
      "table": "MyIngress.myTunnel_exact",
      "match": {
        "hdr.myTunnel.dst_id": [2]
      },
      "action_name": "MyIngress.myTunnel_forward",
      "action_params": {
        "port": 2
      }
    },
    {
      "table": "MyIngress.myTunnel_exact",
      "match": {
        "hdr.myTunnel.dst_id": [3]
      },
      "action_name": "MyIngress.myTunnel_forward",
      "action_params": {
        "port": 3
      }
    }
  ]
}
```

ipv4_lpm table

Key	Action	Action Data
10.0.1.1/32	ipv4_forward	dstAddr = 08:00:00:00:01:11 port = 1
10.0.2.2/32	ipv4_forward	dstAddr = 08:00:00:00:02:00 port = 2
10.0.3.3/32	ipv4_forward	dstAddr = 08:00:00:00:03:00 port = 3
	Default Action	
	NoAction	

myTunnel_exact table

Key (dst_id)	Action	Action Data
1	myTunnel_forward	port = 1
2	myTunnel_forward	port = 2
3	myTunnel_forward	port = 3
	Default Action	
	drop	

Figure 3.6: Forwarding table and Tunneling table of switch s1.

Also for this second exercise, a couple of python scripts are provided for the testing. In this case, the `send.py` script involves the use of the tag `--dst_id` (in addition to the parameters like the IP address and the message to be sent) to specify the `dst_id` field of the myTunnel header.
It is possible to test the normal IP routing in the same way as for the previous exercise with ping or with the python scripts `send.py` and `receive.py` through which all the transmission details are displayed (e.g., the TTL is decremented by one for each crossed switch). Moreover, it is possible to test the actual tunneling by using the two previously described scripts: changing the destination IP in any IP address and adding the tag `--dst_id` with the right ID the

switch forwards the packet considering the port ID and not the IP address. It is possible to see the fields of the MyTunnel header (`proto_id` and `dst_id`) displayed during the transmission. For example, launching the command:

```
./receive.py
```

from the CLI of the host h2 and the command:

```
./send.py 10.0.3.3 "msg" --dst_id 2
```

from the CLI of the host h1, the packet is received at the host h2, even though the IP address is the address of the host h3. Due to the presence of the myTunnel header, the switch is no longer using the IP header for routing but it uses the `dst_id` MyTunnel header field.

3.4 Implementing a Control Plane using P4Runtime

In the first two exercises, a static controller is used to install static flow rules that are specified in a *runtime.json* file. More in detail, the p4c compiler compiles the P4 program and generates a target-specific (BMv2) JSON configuration file that is loaded into the Data Plane and a special file called p4info that is needed to install the table entries. The static rules, in the previous tutorials, are specified in the `sX-runtime.json` files and are installed while bringing up the Mininet instance with the `make run` command.
There are a few approaches to achieving runtime control:

- The P4 compiler auto-generated runtime APIs. These are C++ APIs that provide functionality such as add/remove table entries etc. This approach is program-dependent: if the P4 program changes, the APIs should change too and the controller should be restarted.

- The BMv2 CLI (used in the previous exercises). This approach is program independent but target-specific (BMv2), therefore the control plane is not portable between different targets.

- OpenFlow. It is target-independent but protocol-dependent. Indeed, the supported set of protocols is baked-in the specification and it's not trivial to extend it.

- OCP (Open Compute Project) SAI (Switch Abstraction Interface). As OpenFlow, this approach is target-independent and protocol-dependent.

- P4Runtime. It provides a target-independent and protocol-independent interface between the control plane and the data plane [15].

P4Runtime is a framework for runtime control of P4 targets, it is an open-source project that includes the API definitions and the server implementation. P4Runtime is one of the most commonly used data plane APIs and is standardized in the P4 API WG (Working Group) [23] of the P4 Language Consortium [14]. As shown in Table 3.1, P4Runtime is target-independent and protocol-independent and it allows to push new P4 programs without recompiling the deployed switches.

Table 3.1: Runtime control APIs.

API	Target-Independent	Protocol-Independent
P4 compiler auto-generated	✓	✗
BMv2 CLI	✗	✓
OpenFlow	✓	✗
SAI	✓	✗
P4Runtime	✓	✓

Figure 3.7 depicts the P4Runtime operating principle using a P4 program called `test.p4`. P4 targets always include a gRPC server and controllers implement a gRPC client for the connection between the Control Plane and the Data Plane. The P4Runtime server interacts with the P4-programmable

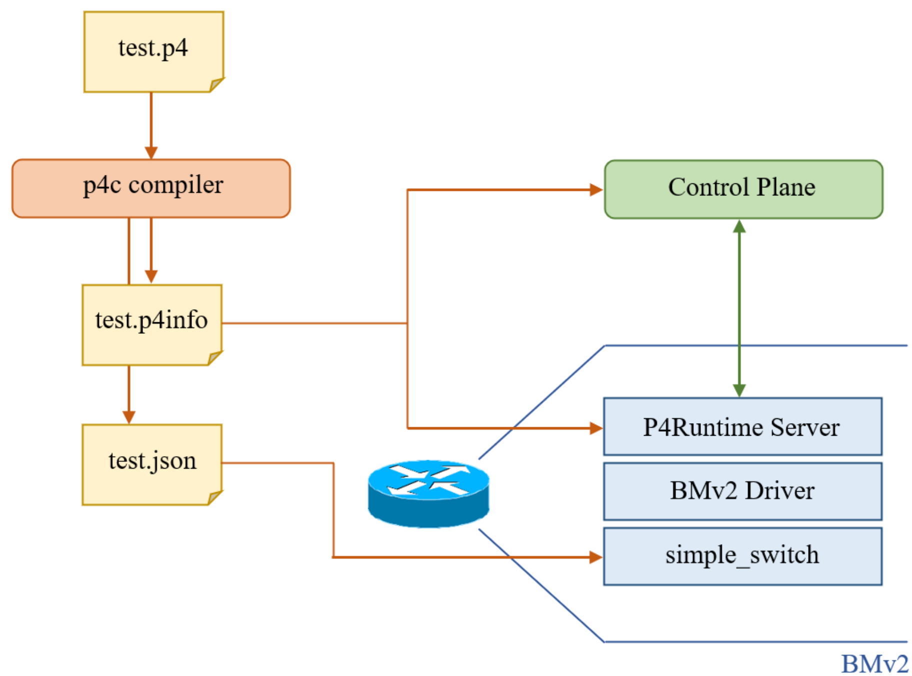


Figure 3.7: Semplified P4Runtime workflow.

components via the BMv2 driver that has access to all the P4 entities (Match-Action Tables, externs, etc.).

The P4Runtime APIs define the messages and semantics of the interface between the client and the server. The API structure of P4Runtime is described within the *p4runtime.proto* Protobuf file which is available on GitHub as part of the standard [24] [25]. The controller can access the P4 entities which are declared in the P4Info metadata. The P4Info structure is defined by *p4info.proto*, another Protobuf file available as part of the standard [24]. This file is part of the P4Runtime but it can be extended to use custom data structures, e.g., to implement interaction with target-specific externs. For each P4 entity, read and write accesses are provided to the controller at runtime.
In the represented workflow, a P4 source program is compiled to produce both a P4 device configuration file and P4Info metadata. Metadata in the P4Info describes, as introduced before, both the overall program itself as well as all entity instances derived from the P4 program. Each entity instance has an associated numeric ID assigned by the P4 compiler which serves as a concise "handle" used in API calls [24]. The target-specific configuration is directly loaded onto the P4 target so that the P4 entities can be accessed. The *P4Info* file, generated by the p4c compiler, is shared between the Control Plane and the Data Plane. It captures P4 program attributes needed at runtime:

- IDs for tables, actions, parameters, etc;
- Table structure, action parameters, etc. [15].

These *P4Info* metadata specify the P4 entities which can be accessed via P4Runtime APIs and, therefore, they have a one-to-one correspondence with instantiated objects in the P4 source code [24].
The *P4Info* schema is designed to be target and architecture-independent, although the specific contents are likely to be architecture-dependent [24]. The *P4Info* file, in any case, is target-independent. Therefore, the compiler, given the same P4 program, generates the same file for all possible different targets.
In this tutorial, P4Runtime is used to send flow entries to the switches instead of installing static table entries by using the switches' CLI. To define the packet-processing pipeline, the same P4 program of the previous exercise is built, renamed to `advanced_tunnel.p4`, and modified by adding new functionalities. More in detail, two counters are defined in the Ingress Processing control block (`ingressTunnelCounter` and `egressTunnelCounter`), the two actions `myTunnel_ingress` and `myTunnel_egress` are implemented and the `myTunnel_exact` table is modified by adding these actions.

The `myTunnel_ingress` action validates the myTunnel header, sets the `dst_id` and `proto_id` parameters and updates the relative `ingressTunnelCounter` value. The `myTunnel_egress` action sets the output port, destination MAC address, and the `etherType` field of the Ethernet header; then it invalidates the myTunnel header and increments the `egressTunnelCounter` value relative to the `dst_id` parameter. The `myTunnel_exact` table is simply modified by adding the above-mentioned actions to the list of possible actions to be invoked.

```
const bit<32> MAX_TUNNEL_ID = 1 << 16;
counter(MAX_TUNNEL_ID, CounterType.packets_and_bytes)
                                        ingressTunnelCounter;
counter(MAX_TUNNEL_ID, CounterType.packets_and_bytes)
                                        egressTunnelCounter;

action myTunnel_ingress(bit<16> dst_id) {
    hdr.myTunnel.setValid();
    hdr.myTunnel.dst_id = dst_id;
    hdr.myTunnel.proto_id = hdr.ethernet.etherType;
    hdr.ethernet.etherType = TYPE_MYTUNNEL;
    ingressTunnelCounter.count((bit<32>) hdr.myTunnel.dst_id);
}

action myTunnel_egress(macAddr_t dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ethernet.etherType = hdr.myTunnel.proto_id;
    hdr.myTunnel.setInvalid();
    egressTunnelCounter.count((bit<32>) hdr.myTunnel.dst_id);
}

table myTunnel_exact {
    key = {
        hdr.myTunnel.dst_id: exact;
    }
    actions = {
        myTunnel_forward;
        myTunnel_egress;
        drop;
    }
    size = 1024;
    default_action = drop();
}
```

Figure 3.8 shows the topology (the same as the previous tutorial *Implementing Basic Tunneling*, Section 3.3) with three switches (s1, s2, s3) configured in a triangle, each connected to one host (h1, h2, h3).

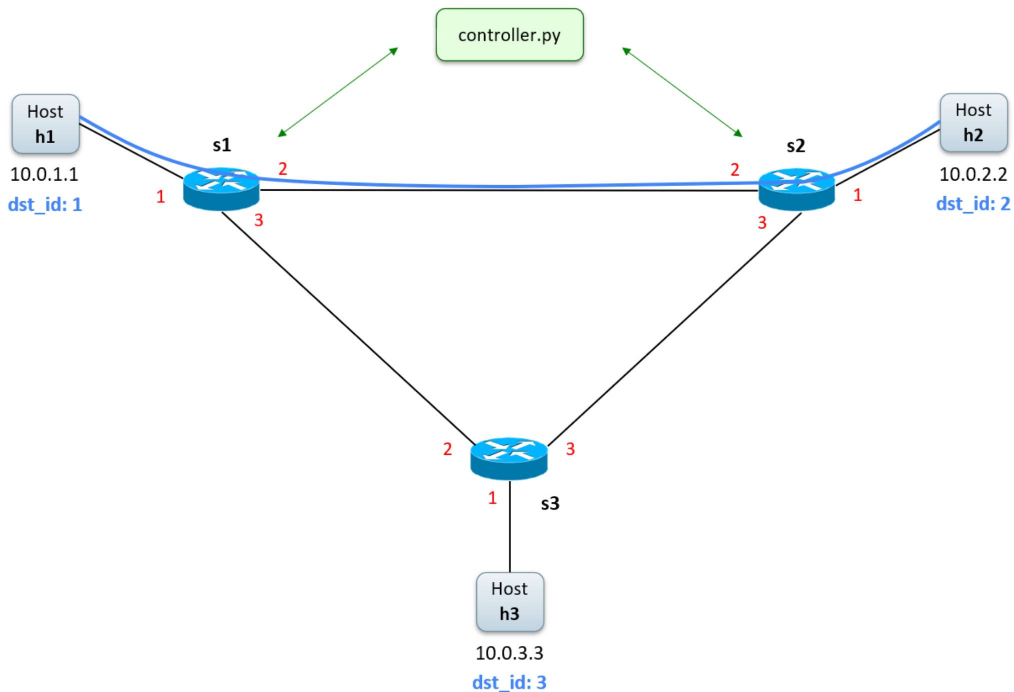


Figure 3.8: Network topology of the tutorial *Implementing a Control Plane*.

The control plane, in this exercise represented by the script `mycontroller.py`, has the following responsibilities:

- establishing a gRPC (Remote Procedure Call) connection to the switches s1 and s2 for the P4Runtime service;
- pushing the P4 program to each switch;
- writing the tunnel forwarding rules:
  - `myTunnel_ingress` rule on the ingress switch in the `ipv4_lpm` table to encapsulate packets into a tunnel with the specified ID;

  - `myTunnel_forward` rule, a transit rule, on the ingress switch to forward packets based on the specified ID;
  - `myTunnel_egress` rule on the egress switch to decapsulate packets with the specified ID and send them to the host.

- reading the tunnel ingress and egress counters (`ingressTunnelCounter` and `egressTunnelCounter`) every two seconds [26].

A few helper libraries are present in the `p4runtime_lib` directory:

- `helper.py`
  - contains the `P4InfoHelper` class which is used to parse the `p4info` files;
  - provides translation methods from entity names to and from ID numbers;
  - builds P4 program-dependent sections of P4Runtime table entries.
- `switch.py`
  - contains the `SwitchConnection` class which grabs the gRPC client stub, and establishes connections to the switches;
  - provides methods that construct the P4Runtime protocol buffer messages and makes the P4Runtime gRPC service calls.
- `bmv2.py`
  - contains `Bmv2SwitchConnection` class which extends the `SwitchConnections` class and provides the BMv2-specific device payload to load the P4 program.
- `convert.py`
  - provides methods to encode and decode from friendly strings and numbers to the byte strings required for the protocol buffer messages;
  - is used by `helper.py` [26].

The P4 compiler generates the interface between the switch pipeline and the control plane. This interface is defined in the `advanced_tunnel.p4info` file. The table entries built in `mycontroller.py` refer to specific tables, keys, and actions by name, and through the `P4InfoHelper` class, it is possible to convert

the names into the IDs that are required for P4Runtime.
The incomplete starter code `mycontroller.py` installs only some of the rules needed to tunnel the traffic between two hosts. In the `writeTunnelRules` function, there are the tunnel ingress and egress rules.

```
SWITCH_TO_HOST_PORT = 1

def writeTunnelRules(   p4info_helper, ingress_sw, egress_sw,
                        tunnel_id, dst_eth_addr, dst_ip_addr):

    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.ipv4_lpm",
        match_fields={
            "hdr.ipv4.dstAddr": (dst_ip_addr, 32)
        },
        action_name="MyIngress.myTunnel_ingress",
        action_params={
            "dst_id": tunnel_id,
        })

    ingress_sw.WriteTableEntry(table_entry)
    print("Installed ingress tunnel rule on %s"
                                    % ingress_sw.name)

    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.myTunnel_exact",
        match_fields={
            "hdr.myTunnel.dst_id": tunnel_id
        },
        action_name="MyIngress.myTunnel_egress",
        action_params={
            "dstAddr": dst_eth_addr,

            "port": SWITCH_TO_HOST_PORT
        })

    egress_sw.WriteTableEntry(table_entry)
    print("Installed egress tunnel rule on %s"
                                    % egress_sw.name)
```

In the main function of the controller, after the switch connection creation and the installation of the P4 program on the switches, two tunnels are created through the `writeTunnelRules` function: one tunnel is from h1 to h2 (`tunnel_id = 100`), with s1 the ingress switch and s2 the egress switch, and

one tunnel from h2 to h1 (`tunnel_id=200`), with s2 the ingress switch and s1 the egress switch.

```
# Write the rules that tunnel traffic from h1 to h2
writeTunnelRules(p4info_helper, ingress_sw=s1, egress_sw=s2,
            tunnel_id=100, dst_eth_addr="08:00:00:00:02:22",
            dst_ip_addr="10.0.2.2")

# Write the rules that tunnel traffic from h2 to h1
writeTunnelRules(p4info_helper, ingress_sw=s2, egress_sw=s1,
            tunnel_id=200, dst_eth_addr="08:00:00:00:01:11",
            dst_ip_addr="10.0.1.1")
```

It's possible to test the initial behavior. Two terminal windows are needed: one for the data plane network and one for the controller program.
In the first one, the command

```
make
```

starts the Mininet instance with the topology described in Figure 3.8. Starting a ping between the hosts h1 and h2, no packets arrive at the receiver because no rules are installed on the switches yet.
In the second terminal window, the command

```
./mycontroller.py
```

installs the `advanced_tunnel.p4` program on the switches and pushes the initial rules. The program prints the counters every two seconds, therefore it is possible to inspect the `ingressTunnelCounter` value. In this case, starting a ping between the hosts h1 and h2 it is possible to see the ingress tunnel counter for the switch s1 increasing and the other counters remaining at zero.
In this first test, each switch is mapping traffic into tunnels based on the IP address, the complete `mycontroller.py` file should contain also the transit rule that allows the switches to forward the traffic between the switches based on the tunnel ID. The tunnel transit rule should be written in the `writeTunnelRules` function, it should be added to the `myTunnel_exact` table, and it should match on tunnel ID (`hdr.myTunnel.dst_id`). In this way, the traffic is forwarded using the `myTunnel_forward` action on the right port.
In the simple topology taken into account, the switches s1 and s2 are connected using a link attached to port 2 on both switches. Therefore a variable `SWITCH_TO_SWITCH_PORT` is defined at the top of the file and it is used as the port parameter for the action.

```
SWITCH_TO_SWITCH_PORT = 2

def writeTunnelRules(p4info_helper, ingress_sw, egress_sw,
                     tunnel_id, dst_eth_addr, dst_ip_addr):

...

table_entry = p4info_helper.buildTableEntry(
    table_name="MyIngress.myTunnel_exact",
    match_fields={
        "hdr.myTunnel.dst_id": tunnel_id
    },
    action_name="MyIngress.myTunnel_forward",
    action_params={
        "port": SWITCH_TO_SWITCH_PORT
    })

ingress_sw.WriteTableEntry(table_entry)
print("Installed transit tunnel rule on %s" % ingress_sw.name)
```

Using the described topology, a single transit rule and a fixed port are sufficient, but in general, a transit rule for each switch in the path is needed and the port should be selected dynamically along the path. While the Mininet network is running, by simply following the same procedure as before, the controller adds the new rule to the switches table. In this case, starting a ping between the hosts h1 and h2, it is possible to see that the values of all counters start to increment.

3.5 Implementing Explicit Congestion Notification

This tutorial aims to extend the basic L3 forwarding with the implementation of Explicit Congestion Notification (ECN). ECN is an extension to the Internet Protocol and Transmission Control Protocol and allows end-to-end notification of network congestion without dropping packets. Explicit Congestion Notification, indeed, is an optional feature that can be used between two ECN-enabled endpoints when the underlying network supports it: if an end-host supports ECN, it sets consequently the value of the `ipv4.ecn` field and each switch can modify this value accordingly to the size of the queue with respect to a threshold. The receiver, at this point, can copy the final value and send it back to the sender to make it lowering the transmission rate.

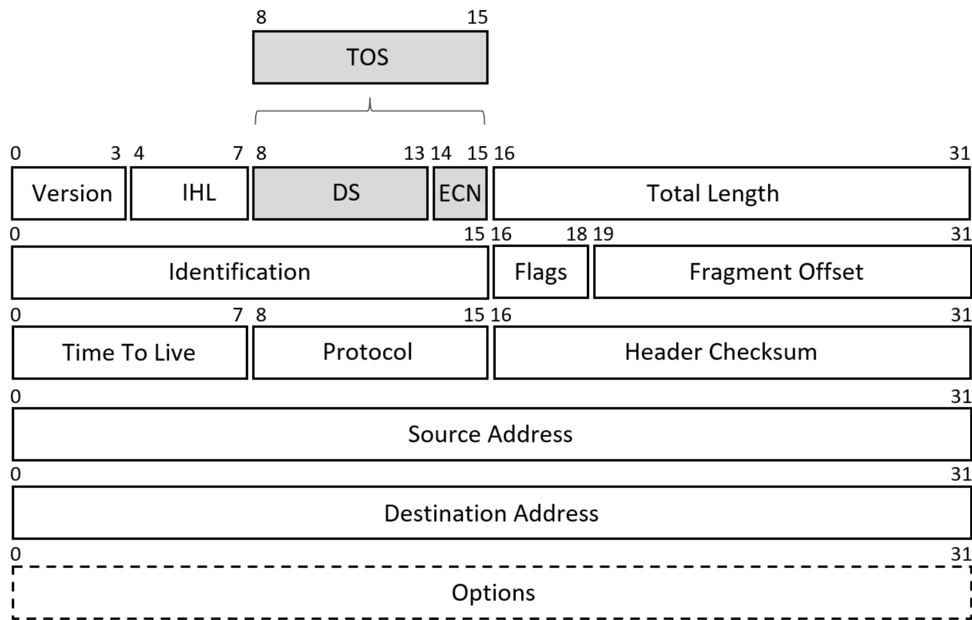


Figure 3.9: IPv4 Header Structure.

Figure 3.9 describes the IPv4 header emphasizing the TOS (Type Of Service) 8 bits field. The first 6 bits represent the DiffServ field and the last 2 bits represent the ECN field.

The ECN field can have four different values that correspond to different states:

- `00` (0): Non ECN-Capable Transport, *Non-ECT*;
- `01` (1): ECN Capable Transport, *ECT*;
- `10` (2): ECN Capable Transport, *ECT*;
- `11` (3): Congestion Encountered, *CE*.

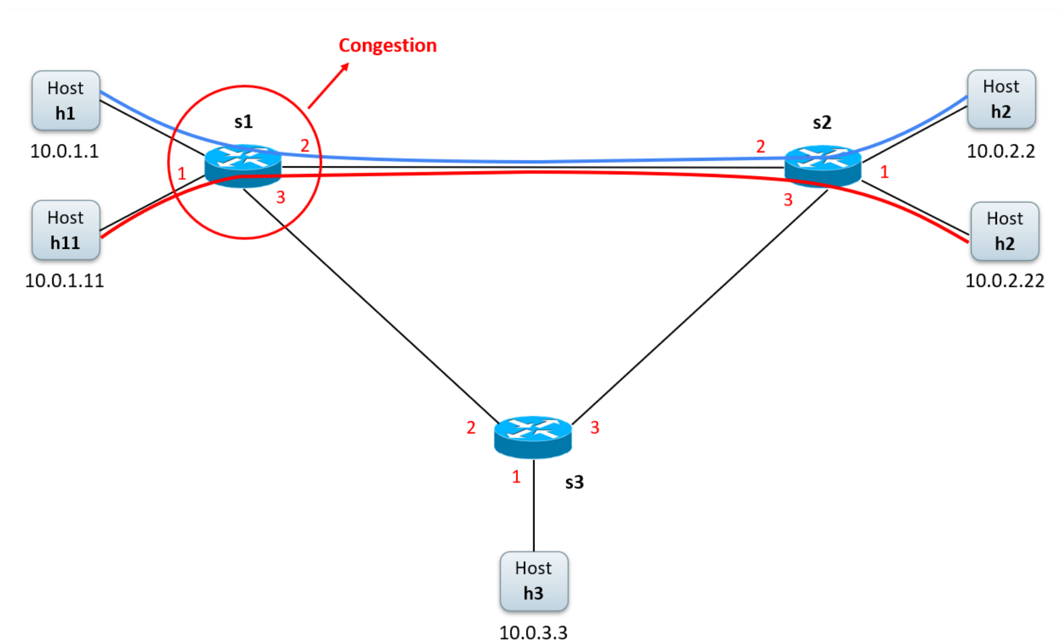


Figure 3.10: Network topology of the tutorial *Implementing ECN*.

The provided skeleton P4 program `ecn.p4` implements L3 forwarding and should be extended to properly set the ECN bits and implement the Explicit Congestion Notification. It is possible to compile the incomplete P4 program. The command

```
make
```

starts a Mininet instance with three switches (s1, s2, and s3) configured in a triangle and five hosts (h1, h11, h2, h22, and h3). As it is possible to see in Figure 3.10, the hosts h1 and h11 are connected to s1, the hosts h2 and h22 are connected to s2 and the host h3 is connected to s3. The previous command also installs the routing static rules based on the configuration files `sX-runtime.json` (where `X` corresponds to each switch number, `X=1, 2, 3`). To test the initial behavior, low-rate traffic is sent from h1 to h2 (through the scripts `send.py` and `receive.py`) and high-rate traffic is sent from h11 to h22 (through the `iperf` tool, which creates a constant bit rate UDP stream). In Figure 3.10, the two kinds of traffic are represented respectively with the blue and the red lines. The link between s1 and s2 is common between the two flows and it can represent a bottleneck.
Four terminals are needed respectively for h1, h11, h2, and h22. The command below in the Mininet CLI allows to open the needed terminals.

```
mininet> xterm h1 h11 h2 h22
```

In h2 terminal, the `receive.py` script should be started to capture incoming packets and store the output in the `h2.log` file.

```
./receive.py > h2.log
```

In h22 terminal, the `iperf` UDP server should be started through the following command.

```
iperf -s -u
```

In h1 terminal, the script `send.py` sends one packet per second for 30 seconds and the message should be received at h2 terminal.

```
./send.py 10.0.2.2 "msg" 30
```

Finally in h11 terminal, the `iperf` client should be started, through the command below, to send packets for 15 seconds.

```
iperf -c 10.0.2.22 -t 15 -u
```

In this first test, the ECN logic in the switches is not implemented yet therefore the `ipv4.tos` field of the IPv4 header is always set to 1 as shown in Figure 3.11.

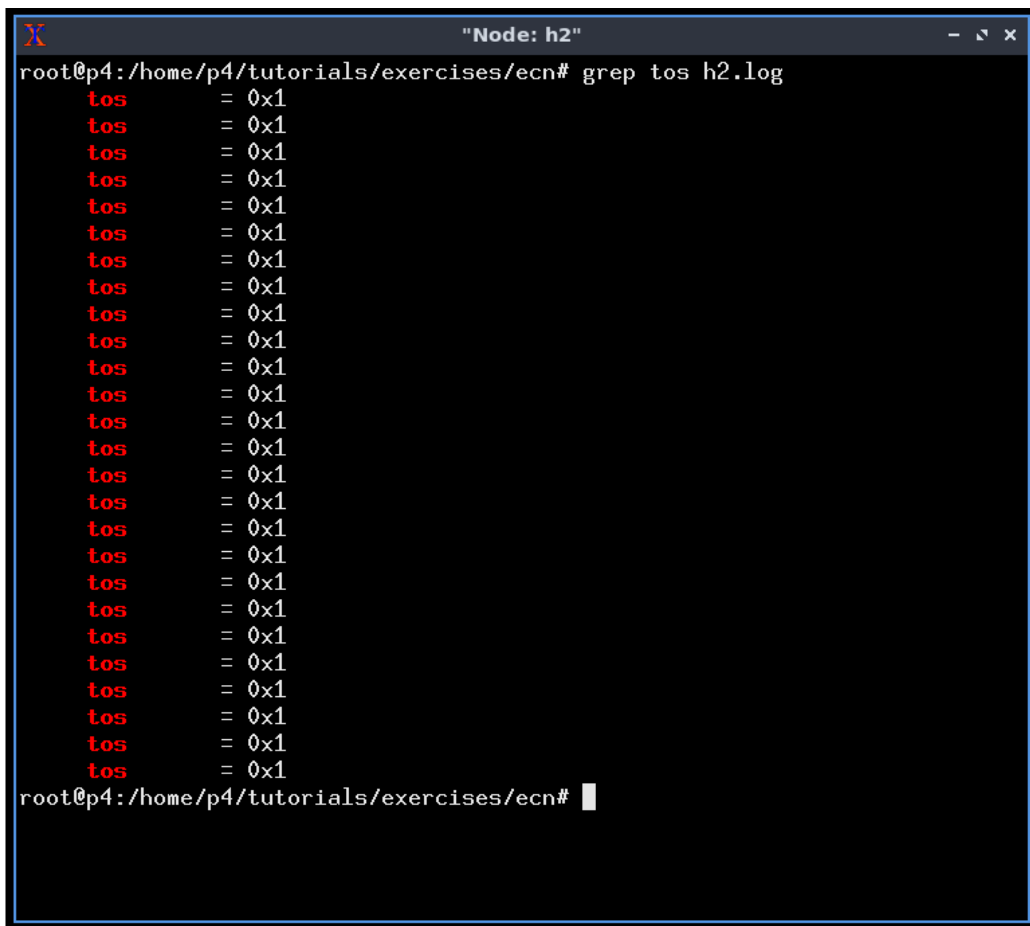


Figure 3.11: Screenshot of the TOS field during the test connection.

A complete `ecn.p4` program implements the ECN logic and is able to properly set the ECN flag. It should contain the following components:

- header type definitions for Ethernet (`ethernet_t`) and IPv4 (`ipv4_t`);
- parsers for Ethernet and IPv4 that populate `ethernet_t` and `ipv4_t` fields;
- an action to drop a packet (`drop`) and an action to forward packets (`ipv4_forward`);

- an ingress control block that performs the lookup of the table `ipv4_lpm` and invokes either drop or `ipv4_forward`.
- an egress control block that checks the ECN and, based on the presence or not of congestion, properly sets the `hdr.ipv4.ecn`;
- a deparser that selects the order in which fields are inserted into the outgoing packet [27].

The IPv4 header should be modified by splitting the TOS field into DiffServ and ECN fields to obtain `hdr.ipv4.diffserv` and `hdr.ipv4.ecn` fields. The checksum block must be updated accordingly.

```
header ipv4_t {
    bit<4>     version;
    bit<4>     ihl;
    bit<6>     diffserv;
    bit<2>     ecn;
    bit<16>    totalLen;
    bit<16>    identification;
    bit<3>     flags;
    bit<13>    fragOffset;
    bit<8>     ttl;
    bit<8>     protocol;
    bit<16>    hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

control MyComputeChecksum(  inout headers hdr,
                            inout metadata meta) {
     apply {
        update_checksum(
            hdr.ipv4.isValid(),
            { hdr.ipv4.version,
              hdr.ipv4.ihl,
              hdr.ipv4.diffserv,
              hdr.ipv4.ecn,
              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);
    }
}
```

In this way, in the P4 program, it is possible to read and write the ECN bits through the `hdr.ipv4.ecn` field of the IPv4 header.
The parsing logic, the deparsing logic, and the Ingress Processing control block are the same implemented in the first tutorial *Implementation Basic Forwarding* [section 3.2].
In the Egress Processing control block, the queue length should be compared with a threshold value defined at the top of the P4 file (`ECN_THRESHOLD`). The logic should be the following: if the ECN value in the packet header is equal to 1 or 2 (*ECT*), the switch should compare the queue length with the threshold and if it is larger than the threshold, the ECN flag is set to 3 (*CE*).
The standard metadata for the V1 Architecture includes the queue depth field that is measured by the Traffic Manager and made available in the Egress Match-Action Pipeline (architecture in Figure 2.5, standard metadata of V1Model Architecture in Section 2.3.4):
The code below shows the definition of the threshold value and the implementation of the Egress Processing control block.

```
const bit<19> ECN_THRESHOLD = 10;

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

    action mark_ecn() {
        hdr.ipv4.ecn = 3;
    }

    apply {
        if (hdr.ipv4.ecn == 1 || hdr.ipv4.ecn == 2){
            if (standard_metadata.enq_qdepth >= ECN_THRESHOLD){
                mark_ecn();
            }
        }
    }
}
```

When the same test described before is performed, the behavior is different this time. With the command

```
./receive.py > h2.log
```

in the h2 terminal, it is possible to track the tos value from the log file. In particular, with the complete P4 program `ecn.p4`, the value of TOS changes from 1 to 3 as the queue builds up and changes back to 1 when `iperf` finishes and the queue depletes [27] as shown in Figure 3.15.

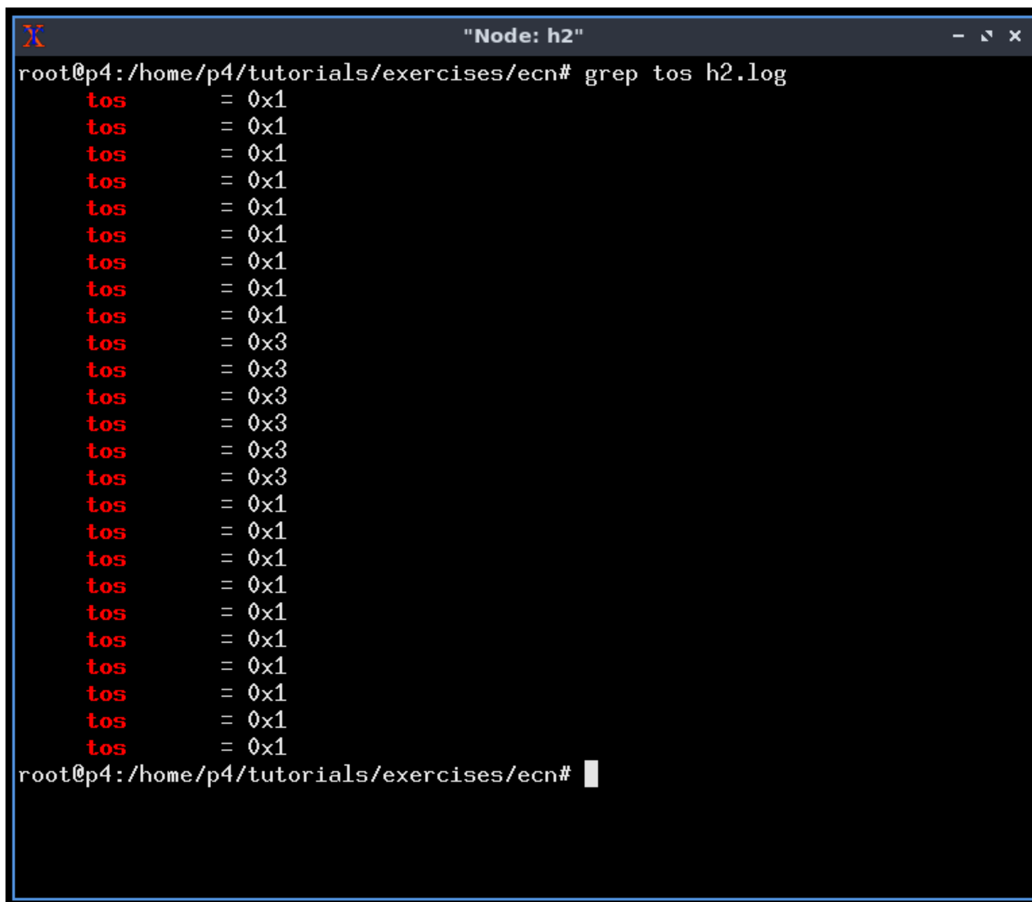


Figure 3.12: Screenshot of the TOS field during the test connection.

3.6 Implementing Multi-Hop Route Inspection

Explicit Congestion Notification (Section 3.5) only shows that there is congestion somewhere in the network without specifying exactly where it is and the extent of the congestion. An optimization of this logic can include the possibility for the switches to modify the packet header to let the network know what happens when it passed through the switch itself. For example, each switch can include its switch ID and queue depth in the packet header or any other statistics that can be useful in network monitoring. In this way, the end host sees network information from several switches and knows if there is congestion, which switch is congested, and its queue depth. The objective of this tutorial is to implement this kind of logic, called Multi-Hop Route Inspection (MRI). In order to track the path and the queue length of switches that every packet travels through, a P4 program that appends an ID and queue length to the header stack of each packet is needed. At the destination host, the sequence of switch IDs corresponds to the path and each ID is followed by the queue length of the switch port [28].
The IPv4 header includes two parts: a fixed part and a variable part. The variable part comprises the Option field that can be a maximum of 40 bytes and it is used for network monitoring and testing.

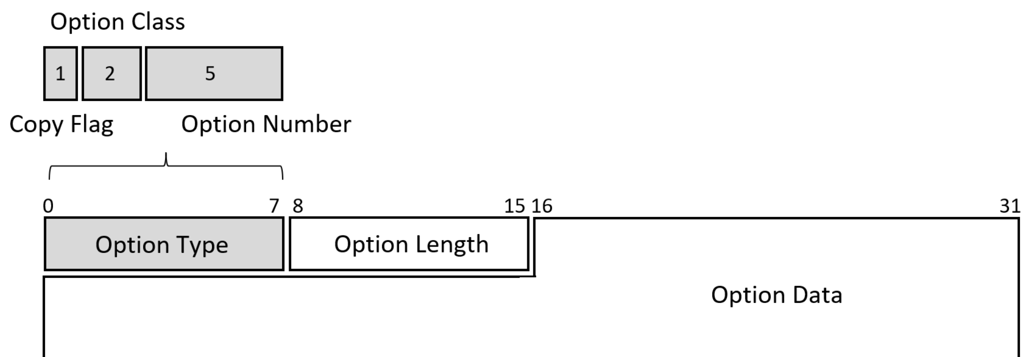


Figure 3.13: IPv4 Option format.

Figure 3.13 shows the format of the Option field. The Option Type field is divided into three subfields: the Copy Flag field is set to 1 if the option

is intended to be copied in all fragments when a datagram is fragmented; the Option Class field specifies the general purpose of the option; the Option Number field defines the type of option. The Option Length field defines the total length of the option including the type field and the length field itself. Finally, there are the data to be sent as part of the option.
In this tutorial, the Option field of the IPv4 header is used to carry the MRI information. Two new header types are defined and added at the end of the header after the Option Type and Option Length fields. The `ipv4_option_t`, `mri_t`, and `switch_t` headers are defined as shown in the following code and are added to the headers struct.

```
header ipv4_option_t {
    bit<1> copyFlag;
    bit<2> optClass;
    bit<5> option;
    bit<8> optionLength;
}

header mri_t {
    bit<16>  count;
}

header switch_t {
    switchID_t  swid;
    qdepth_t    qdepth;
}

struct headers {
    ethernet_t         ethernet;
    ipv4_t             ipv4;
    ipv4_option_t      ipv4_option;
    mri_t              mri;
    switch_t[MAX_HOPS] swtraces;
}
```

The `mri_t` header has just one field that shows how many switches the packet has encountered. If this header is not present in the packet, the switch simply forwards it. The `switch_t` header contains the actual statistic records of each switch hop the packet goes through (the switch ID and the queue depth). The header struct can not have a dynamic size, therefore it must be specified a maximum value for the size of the stack of `switch_t` headers.
A new type of error is defined (`IPHeaderTooShort`) to be thrown if the header

is not correctly formed. Through the `verify` function, the parser can check the length of the header and behave consequently: it transitions to the *Reject* state if it is too short, to the *Accept* state if the length is compatible with the absence of the Option field, and to the `parse_ipv4_option` if it is longer. This is done by looking at the `hdr.ipv4.ihl` field, which specifies the number of 32-bit words in the header (the minimum value is 5).

```
error { IPHeaderTooShort }

state parse_ipv4 {
    packet.extract(hdr.ipv4);
    verify(hdr.ipv4.ihl >= 5, error.IPHeaderTooShort);
    transition select(hdr.ipv4.ihl) {
        5             : accept;
        default       : parse_ipv4_option;
    }
}
```

The initial behavior of the provided skeleton P4 program `mri.p4` is simply performing L3 forwarding. It should be extended to properly manage the MRI custom headers.
The command

```
make
```

compiles the incomplete `mri.p4` program, starts a Mininet instance, and installs packet-processing rules in the tables of each switch (defined in the `sX-runtime.json` files, where `X` corresponds to the switch number). The topology is the same as the previous tutorial and the first test can be done by following the same steps (section 3.5). The received packets do not contain any information about the path they followed.
The complete `mri.p4` program contains the following components:

- header type definitions for Ethernet (`ethernet_t`), IPv4 (`ipv4_t`), IP Options (`ipv4_option_t`), MRI (`mri_t`), and Switch (`switch_t`);
- parsers for Ethernet, IPv4, IP Options, MRI, and Switch that will populate `ethernet_t`, `ipv4_t`, `ipv4_option_t`, `mri_t`, and `switch_t`;
- an action to drop a packet (`drop`) and an action to forward packets (`ipv4_forward`);

- an ingress control that performs the lookup of the table `ipv4_lpm` and invokes either `drop` or `ipv4_forward`;
- an action (called `add_swtrace`) that adds the switch ID and queue depth values to the header;
- an egress control that applies the table `swtrace` to store the switch ID and queue depth, and calls the action `add_swtrace`;
- a deparser that selects the order in which fields are inserted into the outgoing packet [28].

One of the biggest challenges in implementing MRI is handling the recursive logic for parsing the new headers. A `parser_metadata` field, called `remaining`, is used to keep track of how many `switch_t` headers we need to parse [28]. This field should be set to the value of `hdr.mri.count` in the `parse_mri` state, while it should be decremented in the `parse_swtrace` state. In this way, it is possible for the parser to transition from the `parse_swtrace` state to itself until `remaining` is equal to 0. Below is shown the parser code.

```
const bit<5>  IPV4_OPTION_MRI = 31;

state parse_ipv4_option {
    packet.extract(hdr.ipv4_option);
    transition select(hdr.ipv4_option.option) {
        IPV4_OPTION_MRI: parse_mri;
        default: accept;
    }
}

state parse_mri {
    packet.extract(hdr.mri);
    meta.parser_metadata.remaining = hdr.mri.count;
    transition select(meta.parser_metadata.remaining) {
        0 : accept;
        default: parse_swtrace;
    }
}

state parse_swtrace {
    packet.extract(hdr.swtraces.next);
    meta.parser_metadata.remaining =
                        meta.parser_metadata.remaining  - 1;
```

```
    transition select(meta.parser_metadata.remaining) {
        0 : accept;
        default: parse_swtrace;
    }
}
```

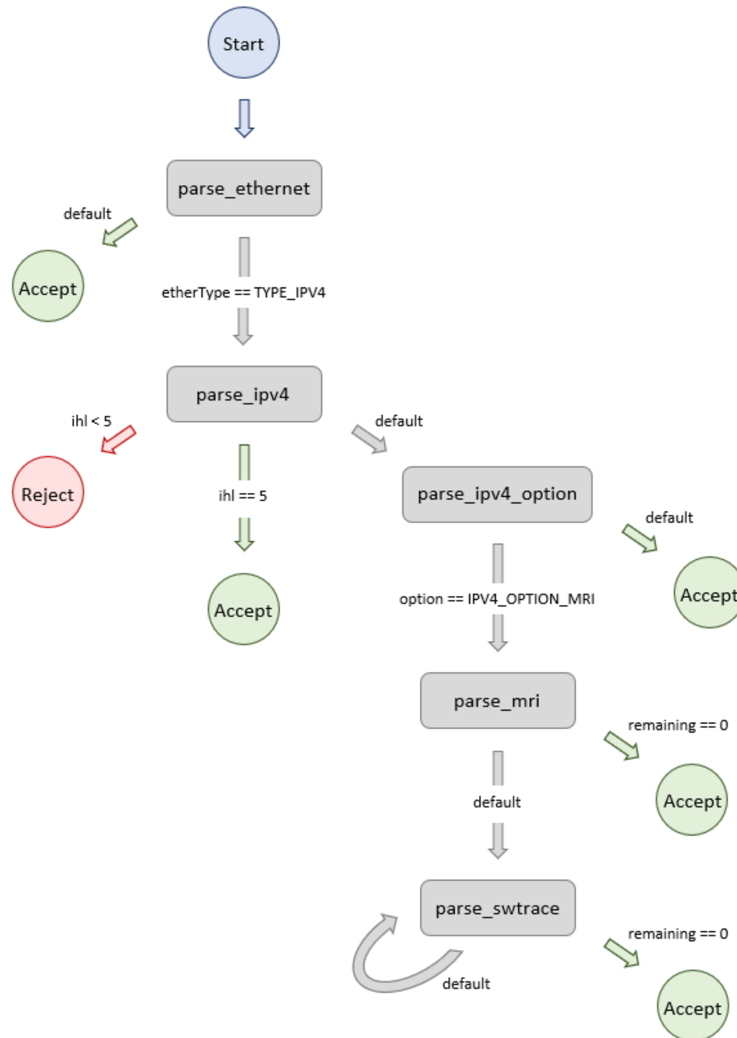


Figure 3.14: Parser state diagram.

Figure 3.14 depicts the parser states (`parse_ethernet`, `parse_ipv4`, `parse_ipv4_options`, `parse_mri`, and `parse_swtrace`) and all the transitions.

In the Egress Processing control block the table `swtrace` and the action `add_swtrace` should be added to increment the `hdr.mri.count` field and append a `switch_t` header. More in detail, as shown in the code below, the `push_front()` method is used on the `swtrace` header stack and, then, it is validated and the ID and queue depth values are added. Moreover, the total length and the option length should be modified according to the addition of the just mentioned data. The `apply` statement should simply check if the `mri` header is present and apply the `swtrace` table.

```
action add_swtrace(switchID_t swid) {
    hdr.mri.count = hdr.mri.count + 1;
    hdr.swtraces.push_front(1);
    hdr.swtraces[0].setValid();
    hdr.swtraces[0].swid = swid;
    hdr.swtraces[0].qdepth =
                        (qdepth_t)standard_metadata.deq_qdepth;

    hdr.ipv4.ihl = hdr.ipv4.ihl + 2;
    hdr.ipv4_option.optionLength =
                        hdr.ipv4_option.optionLength + 8;
    hdr.ipv4.totalLen = hdr.ipv4.totalLen + 8;
 }

table swtrace {
    actions = {
        add_swtrace;
        NoAction;
    }
    default_action = NoAction();
}

apply {
    if (hdr.mri.isValid()) {
        swtrace.apply();
    }
}
```

Static rules are added ar compile time in the table `swtrace` of each switch through the `sX-run-`
`time.json` files. For example, the code below shows the static rule added in the switch s1. This entry is added to the table `swtrace` of swith s1, the action to be invoked is the `add_swtrace` and the parameter that is passed to the action is the ID of the switch itself (equal to 1).

```
"table_entries": [
    {
        "table": "MyEgress.swtrace",
        "default_action": true,
        "action_name": "MyEgress.add_swtrace",
        "action_params": {
            "swid": 1
        }
    },

...

]
```

The deparser, finally, should emit the `ipv4_option`, `mri`, and `swtraces` headers after the Ethernet and IPv4 ones.

```
control MyDeparser(packet_out packet, in headers hdr) {

    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.ipv4_option);
        packet.emit(hdr.mri);
        packet.emit(hdr.swtraces);
    }
}
```

Running the same steps as before, using the modified `mri.p4` program, it is possible to see the sequence of switches through which the packet has traveled and the corresponding statistics. Therefore, when a message is delivered from h1 to h2 through the `send.py` and `receive.py` scripts, all the MRI details are displayed and it is possible to see that the queue length at the common link (from switch s1 to switch s2) is higher when the high rate traffic is traveling [28]. Figure 3.15 shows the details of a packet sent from h1 to h2 in these conditions.

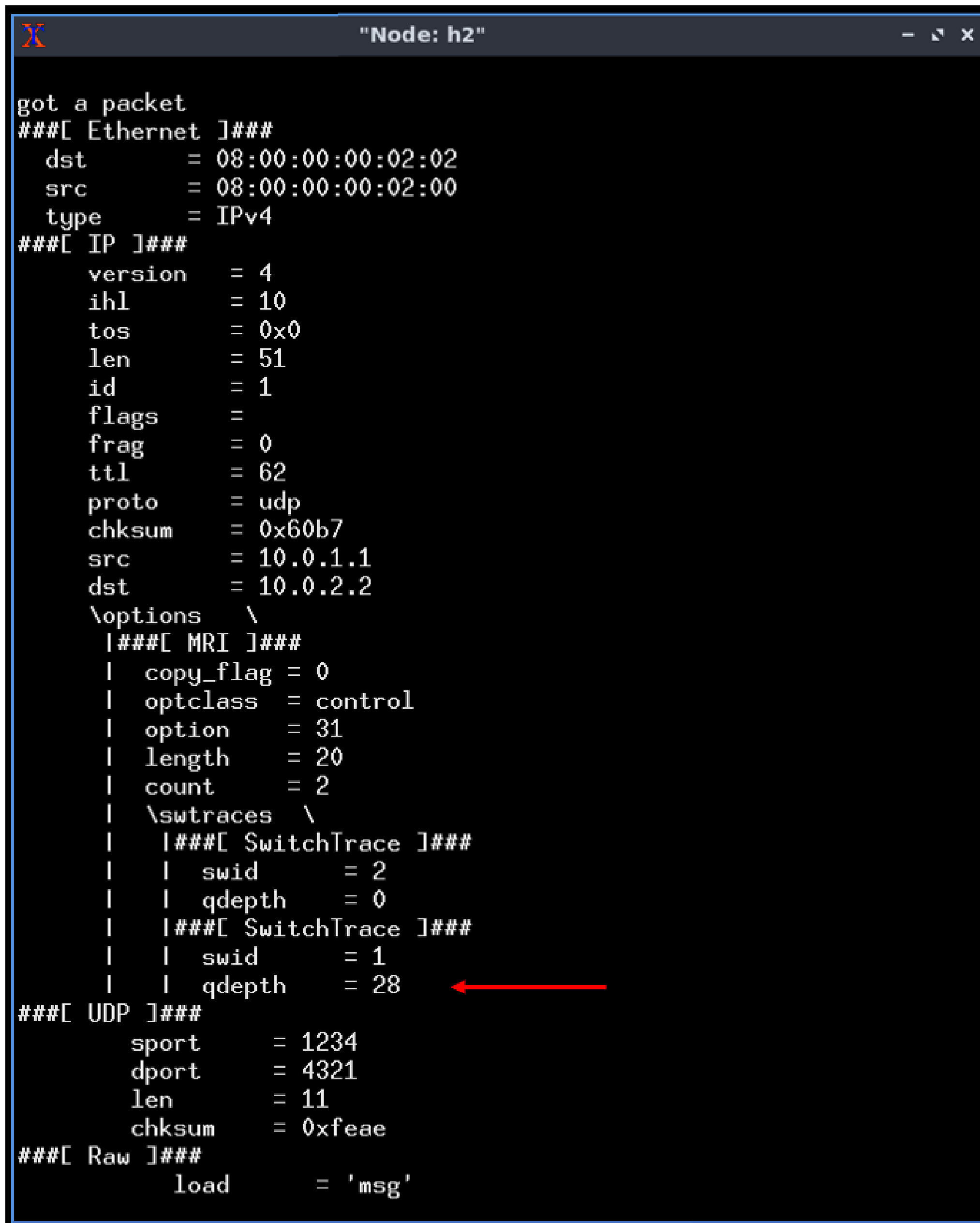


Figure 3.15: Example of connection details of the tutorial *Implementing MRI*.

3.7 Implementing A Basic Stateful Firewall

This tutorial involves the implementation of a basic stateful firewall. The topology utilized is the one in Figure 3.16 which consists of four hosts connected to four switches. In the switches s2, s3, and s4 the basic IPv4 router program (`basic.p4`, explained in section 3.2) is running. The switch s1, instead, is configured with a P4 program that implements the firewall (`firewall.p4`).

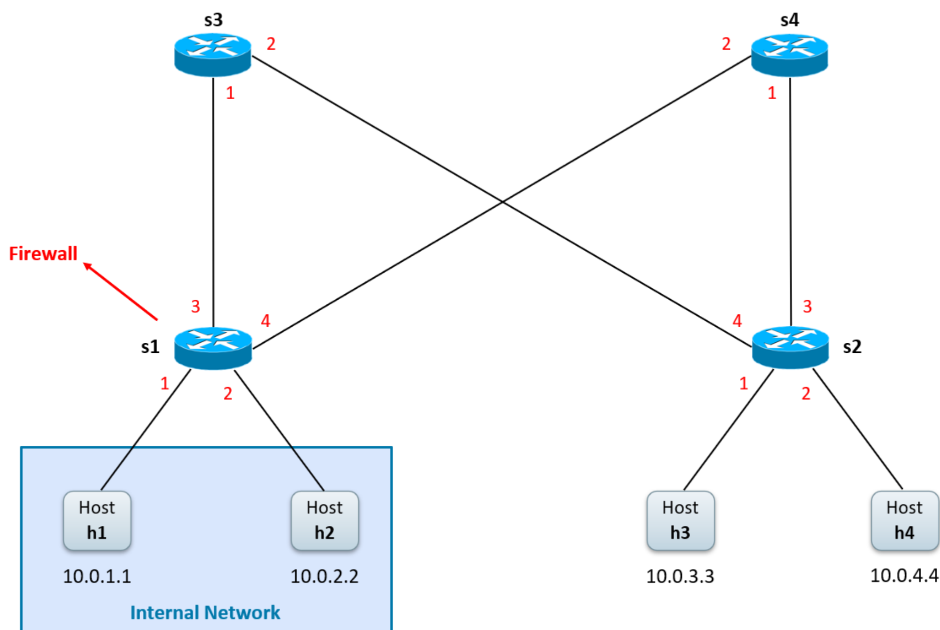


Figure 3.16: Network topology of the tutorial *Implementing a Firewall*.

The `firewall.p4` program should have the following functionalities:

- hosts h1 and h2 are on the Internal Network and can always connect one to the other;
- hosts h1 and h2 can freely connect to h3 and h4 on the external network;

- hosts h3 and h4 can only reply to connections once they have been established from either h1 or h2, but they cannot initiate new connections to the hosts in the Internal Network [29].

As for previous tutorials, a skeleton P4 program is present in the directory and it is possible to compile it and test its initial behavior.
The command

```
make run
```

compiles the `firewall.p4` file, starts the topology in Mininet, and correctly configures all hosts and switches.
In the Mininet CLI, it is possible to try some `iperf` TCP flows between the hosts to see the actual flow traffic between the different hosts.

```
mininet> iperf h1 h2
mininet> iperf h2 h1
```

Through the commands shown above, it can be seen that TCP flows between the hosts h1 and h2 (inside the Internal Network) work.

```
mininet> iperf h1 h3
mininet> iperf h1 h4
mininet> iperf h2 h3
mininet> iperf h2 h4
```

In the same way, TCP flows from hosts in the Internal Network (h1 and h2) to the outside hosts (h3 and h4) also work.

```
mininet> iperf h3 h1
mininet> iperf h3 h1
mininet> iperf h4 h2
mininet> iperf h4 h2
```

TCP flows from outside hosts (h3 and h4) to hosts inside the Internal Network (h1 and h2) should not work, but since the firewall is not implemented yet, in this first test they work [29].
To implement the basic stateful firewall in the data plane, a bloom filter is used. This filter has two functionalities: checking if a packet is coming into the Internal Network from the outside and verifying if it is part of an already established TCP connection. It is implemented through two registers: `bloom_filter_1` and `bloom_filter_2`.

The complete `firewall.p4` program should contain the following components:

- header type definitions for Ethernet (`ethernet_t`), IPv4 (`ipv4_t`) and TCP (`tcp_t`);
- parsers for Ethernet, IPv4 and TCP that populate `ethernet_t`, `ipv4_t` and `tcp_t` fields;
- an action to drop a packet, using `mark_to_drop()`;
- an action (called `compute_hashes`) to compute the bloom filter's two hashes using hash algorithms *crc16* and *crc32*. The hashes are computed on the packet 5-tuple consisting of IPv4 source and destination addresses, source and destination port numbers, and the IPv4 protocol type;
- an action (`ipv4_forward`) and a table (`ipv4_lpm`) that performs basic IPv4 forwarding (section 3.2);
- an action (called `set_direction`) that simply sets a one-bit direction variable;
- a table (called `check_ports`) that reads the ingress and egress port of a packet (after IPv4 forwarding) and invokes `set_direction`. The direction is set to 1 if the packet is incoming into the internal network. Otherwise, the direction is set to 0. To achieve this, the file `s1-runtime.json` contains the appropriate entries for the `check_ports` table.
- a control that:
  - applies the table `ipv4_lpm` if the packet has a valid IPv4 header;
  - if the TCP header is valid, applies the `check_ports` table to determine the direction;
  - applies the `compute_hashes` action to compute the two hash values which are the bit positions in the two register arrays of the bloom filter (`reg_pos_one` and `reg_pos_two`). When the direction is 1, i.e. the packet is incoming into the Internal Network, `compute_hashes` is invoked by swapping the source and destination IPv4 addresses and the source and destination ports. This is to have an unique value representing the couple sender-receiver and to check against

the bloom filter's set bits when the TCP connection was initially made from the internal network;
  - if the TCP packet is going out of the internal network and it is a SYN packet, sets to 1 both the bloom filter registers at the computed bit positions (`reg_pos_one` and `reg_pos_two`);
  - if the TCP packet is entering the internal network, read both the bloom filter registers at the computed bit positions and drop the packet if at least one is not set to 1;

- a deparser that emits the Ethernet, IPv4, and TCP headers in the right order [29].

In this exercise, the parser scheme is the one shown in Figure 3.17. Apart from the standard states, the three custom states `parse_ethernet`, `parse_ipv4`, and `tcp` are implemented to extract respectively the ethernet, IPv4, and TCP headers.
In the following code, the `tcp_t` header and the `tcp` parser state are shown.

```
header tcp_t{
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4>  dataOffset;
    bit<4>  res;
    bit<1>  cwr;
    bit<1>  ece;
    bit<1>  urg;
    bit<1>  ack;
    bit<1>  psh;
    bit<1>  rst;
    bit<1>  syn;
    bit<1>  fin;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}

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

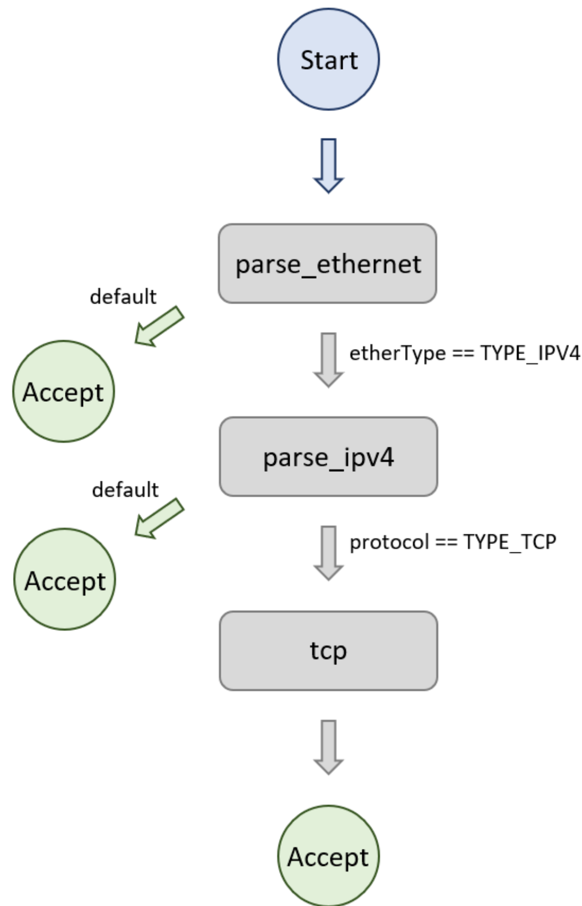


Figure 3.17: Parser state diagram.

At the beginning of the Ingress Processing control block the two registers are defined, as well as the following variables: positions variables that are used as register indices (`reg_pos_one` and `reg_pos_two`), values to be write on and read from the registers (`reg_val_one` and `reg_val_two`) and the one-bit value representing the direction (`direction`).

```
#define BLOOM_FILTER_ENTRIES 4096
#define BLOOM_FILTER_BIT_WIDTH 1

register<bit<BLOOM_FILTER_BIT_WIDTH>>(BLOOM_FILTER_ENTRIES)
                                                bloom_filter_1;
register<bit<BLOOM_FILTER_BIT_WIDTH>>(BLOOM_FILTER_ENTRIES)
```

```
                                                   bloom_filter_2;
bit<32> reg_pos_one; bit<32> reg_pos_two;
bit<1> reg_val_one; bit<1> reg_val_two;
bit<1> direction;
```

The `compute_hashes` action exploits the extern `hash`, defined in the V1Model Architecture file `v1model.p4`. This action, shown in the code below, computes two hash values with two different hash algorithms: *crc16* and *crc32*, and stores them respectively in the `res_pos_one` and `reg_pos_two` variables. Later they are used as unique indices in the two registers and they represent the specific connection. Indeed, these values are computed starting from the packet 5-tuple made of IPv4 source and destination addresses, source and destination ports, and the IPv4 protocol.

```
action compute_hashes(ip4Addr_t ipAddr1, ip4Addr_t ipAddr2,
                      bit<16> port1, bit<16> port2) {

   //Get register position

   hash(reg_pos_one, HashAlgorithm.crc16, (bit<32>)0, {ipAddr1,
                ipAddr2, port1, port2, hdr.ipv4.protocol},
                (bit<32>)BLOOM_FILTER_ENTRIES);

   hash(reg_pos_two, HashAlgorithm.crc32, (bit<32>)0, {ipAddr1,
                ipAddr2, port1, port2, hdr.ipv4.protocol},
                (bit<32>)BLOOM_FILTER_ENTRIES);
}
```

The table `check_ports` matches in the egress and ingress ports (exact match) and calls the `set_direction` action that simply set the one-bit value passed as an action parameter. In the following piece of code, these implementations are shown.

```
action set_direction(bit<1> dir) {
    direction = dir;
}

table check_ports {
    key = {
        standard_metadata.ingress_port: exact;
        standard_metadata.egress_spec: exact;
    }
    actions = {
        set_direction;
```

```
        NoAction;
    }
    size = 1024;
    default_action = NoAction();
}
```

	Action	
Key	Action	Action Data
ingress_port == 1 egress_spec == 3	set_direction	dir = 0
ingress_port == 1 egress_spec == 4	set_direction	dir = 0
ingress_port == 2 egress_spec == 3	set_direction	dir = 0
ingress_port == 2 egress_spec == 4	set_direction	dir = 0
ingress_port == 3 egress_spec == 1	set_direction	dir = 1
ingress_port == 3 egress_spec == 2	set_direction	dir = 1
ingress_port == 4 egress_spec == 1	set_direction	dir = 1
ingress_port == 4 egress_spec == 2	set_direction	dir = 1
	Default Action	
	NoAction	

Figure 3.18: `check_port` table of switch s1.

In Figure 3.18, an example of `check_ports` table entries of switch s1 is represented, while in the following code are shown the corresponding entries, written in the `s1-runtime.json` file.
As it is possible to see from the graphical table representation and from the implemented rules described by the code, the direction is set to 1 when the packet is incoming into the internal network, while it is set to 0 if the packet

is directed from the internal network to the external hosts.

```
{
  "target": "bmv2",
  "p4info": "build/firewall.p4.p4info.txt",
  "bmv2_json": "build/firewall.json",
  "table_entries": [
    {
      "table": "MyIngress.check_ports",
      "match": {
        "standard_metadata.ingress_port": 1,
        "standard_metadata.egress_spec": 3
      },
      "action_name": "MyIngress.set_direction",
      "action_params": {
        "dir": 0
      }
    },
    {
      "table": "MyIngress.check_ports",
      "match": {
        "standard_metadata.ingress_port": 1,
        "standard_metadata.egress_spec": 4
      },
      "action_name": "MyIngress.set_direction",
      "action_params": {
        "dir": 0
      }
    },
    {
      "table": "MyIngress.check_ports",
      "match": {
        "standard_metadata.ingress_port": 2,
        "standard_metadata.egress_spec": 3
      },
      "action_name": "MyIngress.set_direction",
      "action_params": {
        "dir": 0
      }
    },
    {
      "table": "MyIngress.check_ports",
      "match": {
        "standard_metadata.ingress_port": 2,
        "standard_metadata.egress_spec": 4
```

```
    },
    "action_name": "MyIngress.set_direction",
    "action_params": {
      "dir": 0
    }
  },
  {
    "table": "MyIngress.check_ports",
    "match": {
      "standard_metadata.ingress_port": 3,
      "standard_metadata.egress_spec": 1
    },
    "action_name": "MyIngress.set_direction",
    "action_params": {
      "dir": 1
    }
  },
  {
    "table": "MyIngress.check_ports",
    "match": {
      "standard_metadata.ingress_port": 3,
      "standard_metadata.egress_spec": 2
    },
    "action_name": "MyIngress.set_direction",
    "action_params": {
      "dir": 1
    }
  },
  {
    "table": "MyIngress.check_ports",
    "match": {
      "standard_metadata.ingress_port": 4,
      "standard_metadata.egress_spec": 1
    },
    "action_name": "MyIngress.set_direction",
    "action_params": {
      "dir": 1
    }
  },
  {
    "table": "MyIngress.check_ports",
    "match": {
      "standard_metadata.ingress_port": 4,
      "standard_metadata.egress_spec": 2
    },
```

```
      "action_name": "MyIngress.set_direction",
      "action_params": {
        "dir": 1
      }
    }
  ]
}
```


Finally, the `apply` block of the Ingress Processing control block implements the actual firewall behavior. After the verification of the headers, the `check_ports` table is applied and, depending on the direction, the action `compute_hashes` takes as input parameters the addresses and port numbers in the right order: source and then destination addresses and ports if the packet is going outside the internal network (if `direction` is equal to 0) and the opposite order if the packet is coming into the internal network (if `direction` is equal to 1).
When a packet comes from the internal network, if the `SYN` flag is set to 1, the registers are updated and in the respective positions, `reg_pos_one` for `bloom_filter_1` and `reg_pos_two` for `bloom_filter_2`, the one-bit value is set to 1. In this way, when one host in the internal network starts a connection, the relative register value is set to 1 and it is readable while processing the successive packets of the same connection that, therefore, are not discarded.
When a packet comes from outside, the values are read from the registers at the relative indices and, if both values are equal to 1, the flow is allowed to pass, otherwise, the packet is dropped.

```
apply {
    if (hdr.ipv4.isValid()){
        ipv4_lpm.apply();
        if (hdr.tcp.isValid()){
            direction = 0; // default
            if (check_ports.apply().hit) {
                if (direction == 0) {
                    compute_hashes(hdr.ipv4.srcAddr,
                                   hdr.ipv4.dstAddr,
                                   hdr.tcp.srcPort,
                                   hdr.tcp.dstPort);
                }
                else {
                    compute_hashes(hdr.ipv4.dstAddr,
                                   hdr.ipv4.srcAddr,
                                   hdr.tcp.dstPort,
                                   hdr.tcp.srcPort);
```

```
                }
                if (direction == 0){
                    if (hdr.tcp.syn == 1){
                        bloom_filter_1.write(reg_pos_one, 1);
                        bloom_filter_2.write(reg_pos_two, 1);
                    }
                }
                else if (direction == 1){
                    bloom_filter_1.read(reg_val_one,
                                        reg_pos_one);
                    bloom_filter_2.read(reg_val_two,
                                        reg_pos_two);
                    if (reg_val_one != 1 || reg_val_two != 1){
                        drop();
                    }
                }
            }
        }
    }
}
```

After having extended the `firewall.p4` file, it is possible to test the correct behavior by following the same steps as before. This time, by launching the commands

```
mininet> iperf h3 h1
mininet> iperf h3 h1
mininet> iperf h4 h2
mininet> iperf h4 h2
```

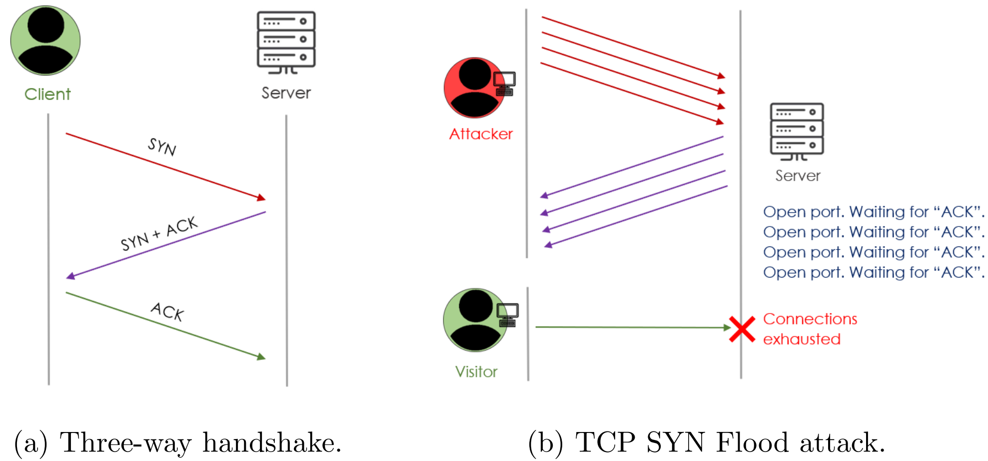
the TCP flows from outside hosts (h3 and h4) to hosts inside the internal network (h1 and h2) should be blocked by the firewall.

Chapter 4

DoS Prevention Service Implementation

In this Chapter, after a brief introduction about DOS attacks and how to prevent them, the implemented service for DoS prevention is presented. TCP SYN Flood is a common type of Denial-of-Service (DoS) or Distributed-Denial-of-Service (DDoS) attack that can target any system connected to the Internet and that provides Transmission Control Protocol (TCP) services (for example web servers, email servers, etc). This form of DoS attack exploits part of the normal TCP three-way handshake to consume all available server resources and, therefore, to make the target server unresponsive and unavailable to legitimate traffic. Essentially, the attacker sends TCP connection requests faster than the target server can process them, causing network saturation. When a client and server establish a normal TCP collection, the so-called “three-way handshake” process is performed, and the messages exchange looks like this:

- the client requests a connection by sending a SYN (synchronize) message to the server, therefore a TCP message with the SYN flag set and the aim to start a new connection;
- the server acknowledges by sending a SYN-ACK (synchronize-acknowledge) message back to the client, therefore a TCP message with both the SYN flag and the ACK flag set;
- the client finally responds with an ACK (acknowledge) message, therefore a TCP message with the ACK flag set, and the connection is established.



(a) Three-way handshake. (b) TCP SYN Flood attack.

Figure 4.1: TCP messages.

The listed steps are graphically represented in Figure 4.1a, while Figure 4.1b represents the case in which an attacker exploits the fact that after the initial SYN packet has been received, the server responds back with one or more SYN/ACK packets and it waits for the final step in the handshake. In the second case, the message exchange looks like this:

- the attacker sends a high volume of SYN messages to the target server;
- the server responds to each connection request and keeps the port open in order to receive the client's response;
- the attacker never sends any responses and the server keeps waiting for the final ACK packets.
- in this way, while waiting for the arrival of each new SYN packet, the server is maintaining a new open port connection, and, when a client tries to connect to the server, the server is unable to correctly provide the requested service.

In the next Sections, a simple P4 program and a P4 controller are proposed to

be integrated into the FORCH (Fog Orchestration) System and dynamically provide a service for DoS Prevention.

4.1 DoS prevention P4 Program

In this first Section of Chapter 4, the P4 program is presented. The network topology utilized for testing this program is the one shown in Figure 4.2.

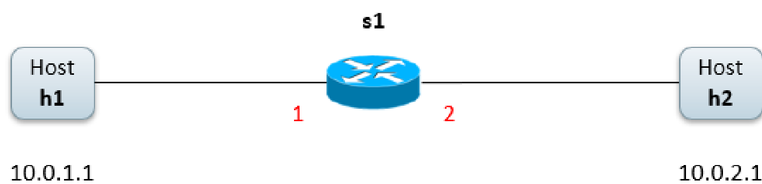


Figure 4.2: Network Topology.

It includes one switch s1 and two hosts, h1 and h2, connected to it. In particular, the host h1 represents the one that needs to be protected during a DoS attack performed by the host h2. Therefore, in switch s1, the P4 program `antiDOS.p4` is running and the proper flow rules are installed in order to protect host h1. The code below is written in the `topology.json` file, used to build up the topology of Figure 4.2 with Mininet. The JSON file includes the configuration of both the hosts, the switch, and the links connecting these three components.

```
{
  "hosts": {
      "h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
         "commands":["route add default gw 10.0.1.10 dev eth0",
                "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00"]},
      "h2": {"ip": "10.0.2.1/24", "mac": "08:00:00:00:02:22",
         "commands":["route add default gw 10.0.2.20 dev eth0",
                "arp -i eth0 -s 10.0.2.20 08:00:00:00:02:00"]}
  },
  "switches": {
      "s1": { "runtime_json" : "pod-topo/s1-runtime.json" }
```

```
  },
  "links": [
      ["h1", "s1-p1"], ["h2", "s1-p2"]
  ]
}
```

The P4 program `antiDOS.p4` should contain the following components:

- header type definitions for Ethernet (`ethernet_t`), IPv4 (`ipv4_t`) and TCP (`tcp_t`);
- parsers for Ethernet, IPv4 and TCP that populate `ethernet_t`, `ipv4_t` and `tcp_t`;
- an action to drop a packet (`drop`) that uses the primitive `mark_to_drop()` action, defined in the file `v1model.p4`;
- an action (`ipv4_forward`) and a table (`ipv4_lpm`) that performs basic IPv4 forwarding;
- an action, called `calc_hash`, that calculates a unique value starting from the 4-tuple consisting of destination IP address, destination port number, source IP address, and source port number of the considered packet;
- a table, called `protected_server`, that matches on the destination IP address and destination port number, invokes one action among NoAction and `calc_hash`, does not includes any action parameters, and which default action is NoAction;
- an ingress control that:
  - if the packet has a valid IPv4 header, performs the lookup of the table `ipv4_lpm` and invokes either the `drop` or the `ipv4_forward` action;
  - if the packet has a valid TCP header, performs the lookup of the table `protected_server` and invokes the action `calc_hash`;
  - if the `protected_server` table is hit, increments a connection specific counter, implemented through a P4 register, with the aim to keep track of the number of connections established from the packet source to the packet destination;

    - when the counter reaches a predefined threshold, blocks the successive attempts to establish a new connection from the same source IP address and port number;
    - after a predefined amount of time from the locking of new connections, resets the counter in order to allow again the establishment of connections from the IP address and port number;

- a deparser that selects the order in which the headers are inserted into the outgoing packet.

The headers used by this P4 program are the Ethernet header, the IPv4 header, and the TCP header. They are defined as follows and are added to the headers struct.

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

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;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

header tcp_t{
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4>  dataOffset;
    bit<4>  res;
    bit<1>  cwr;
    bit<1>  ece;
```

```
    bit<1>  urg;
    bit<1>  ack;
    bit<1>  psh;
    bit<1>  rst;
    bit<1>  syn;
    bit<1>  fin;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
    tcp_t        tcp;
}
```

Through the defined headers, it is possible to access all the specific fields of each header during the processing of packets. The parser, in addition to the three standard states (*Start*, *Accept*, and *Reject*), is composed of the following states: `parse_ethernet`, `parse_ipv4`, and `parse_tcp`. From the *Start* state, the parser transitions to the `parse_ethernet` state where, if the field `hdr.ethernet.etherType` corresponds to the `TYPE_IPV4` value, it transitions to the `parse_ipv4` state. Here, the `hdr.ipv4.protocol` field of the IPv4 header is inspected and, if it is equal to the `TYPE_TCP` value, it transitions to the `parse_tcp state`. The final transition is made to the *Accept* state. The transitions to the *Reject* state are implicit and they happen each time a packet is not well-formed therefore packet processing is not feasible. The values `TYPE_IPV4` and `TYPE_TCP` are defined at the beginning of the P4 program and they correspond to two specific values defined in the Ethernet and IPv4 protocol specifications. In particular, the `etherType` field (16 bits) of the Ethernet header is used to indicate which protocol is encapsulated in the payload of the Ethernet frame and, if it is equal to `0x800` (hexadecimal value), the payload encapsulates the IPv4 protocol. The `protocol` field (8 bits) of the IPv4 header defines the protocol used in the data portion of the IP datagram and, if it is equal to `0x06` (hexadecimal value, corresponding to the protocol number `6`), the encapsulated protocol corresponds to the Transmission Control Protocol (TCP).
The code below shows the parser implementation.

```
const bit<16> TYPE_IPV4 = 0x800;
const bit<8>  TYPE_TCP  = 6;

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 select(hdr.ipv4.protocol){
            TYPE_TCP: parse_tcp;
            default: accept;
        }
    }

    state parse_tcp {
       packet.extract(hdr.tcp);
       transition accept;
    }
}
```

Figure 4.3 represents the scheme of the parser as a finite state machine (FSM) with the explicit states and transitions.
In the Ingress Processing control block, the tables and the actions are implemented. For the computations included in this control block, some user-defined metadata are needed: `pkt_hash` is a variable used to store the unique value calculated through the `hash` function that is used as an index register later on; `count` is a variable used to store the incrementing number of attempts to establish new connections. The other two fields of the struct, `timemaxsyn` and

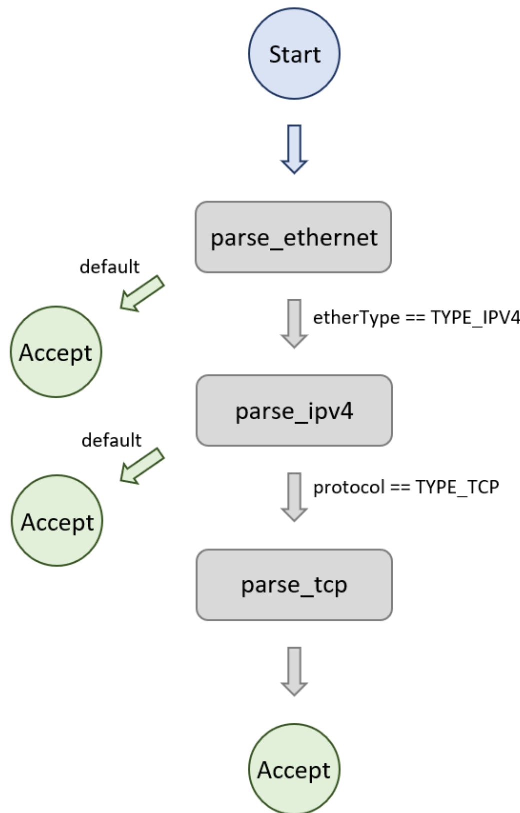


Figure 4.3: Parser state diagram.

`interval`, are exploited in the `apply` block and, therefore, are explained later this Section.

```
struct metadata {
    bit<32>     pkt_hash;
    bit<32>     count;
    bit<48>     timemaxsyn;
    bit<48>     interval;
}
```

The `drop` action is implemented simply by exploiting the `mark_to_drop` function. This primitive function is defined in the file `v1model.p4` and it modifies the field `standard metadata.egress_spec` to an implementation-specific value that causes the packet to be dropped at the end of ingress processing.

The `ipv4_forward` action performs the following operations: setting the egress port to the port number provided by the control plane, updating the packet's source MAC address with the MAC address of the switch (packet's current destination MAC address), updating the packet's destination MAC address with the address of the next hop provided by the control plane, and decrementing the TTL field of the IPv4 header.
The `ipv4_lpm` Match-Action Table is the actual forwarding table. It matches on the destination IP address with an LPM (Longest Prefix Match) type match. In particular, the table entries specify the IP addresses with the relative subnet masks and the match is found with the longest prefix among all the table entries. The list of actions that can be invoked includes the described `ipv4_forward` action, `drop` action, and `NoAction`. The `drop` action is also the default action, therefore it is invoked any time there is no hit.
The action `calc_hash` uses the V1Model Architecture predefined function `hash()`. This function is defined in the `v1model.p4` file and it is a mathematical function capable of converting an input, possibly made of more than one parameter, into a unique output of fixed length. It takes as input parameters the variable into which the result is stored (`meta.pkt_hash`), the algorithm utilized to calculate the hash value (`HashAlgorithm.crc32`), the first and last values of the range of possible results and the 4-tuple that has to be converted into a single value. Moreover, this action reads from the register `mycounter` the value corresponding to the index just calculated. This value, stored in the `meta.count` variable, represent the number of attempts of establishing a new connection counted until this moment.
The `protected_server` Match-Action Table is the one responsible for the DoS prevention logic and it is applied to the inserted target hosts. The declaration `protected_server` MAT specifies what to match on (destination IP address and destination port number), a list of possible actions (the previously described `calc_hash` and `NoAction`), and additional properties such as the size of the table and the default action that will be executed if there is no match. When an entry is added to this MAT, the relative host is protected from DoS attacks in the way specified in the P4 program.
The following code shows the implementation of the `ipv4_forward` action, the `ipv4_lpm` Match-Action Table, the `calc_hash` action, and the `protected_server` Match-Action Table.

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

    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port){
        standard_metadata.egress_spec = port;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    action calc_hash() {
        hash(meta.pkt_hash, HashAlgorithm.crc32, (bit<32>)0,
                      {hdr.ipv4.dstAddr,
                       hdr.tcp.dstPort,
                       hdr.ipv4.srcAddr,
                       hdr.tcp.srcPort}, (bit<32>)8192);
        mycounter.read(meta.count, (bit<32>)meta.pkt_hash);
    }

    table protected_server {
        key = {
            hdr.ipv4.dstAddr: exact;
            hdr.tcp.dstPort : exact;
        }
        actions = {
            calc_hash;
            NoAction;
```

```
        }
        size = 1024;
        default_action = NoAction;
    }

    ...
}
```

At the end of the Ingress Processing control block, after the definition of all the tables and actions, the `apply` block performs the actual program logic. After the verification of IPv4 and TCP headers validity, the SYN flag of the TCP header is inspected. This flag, pointed out in Figure 4.4, represents the

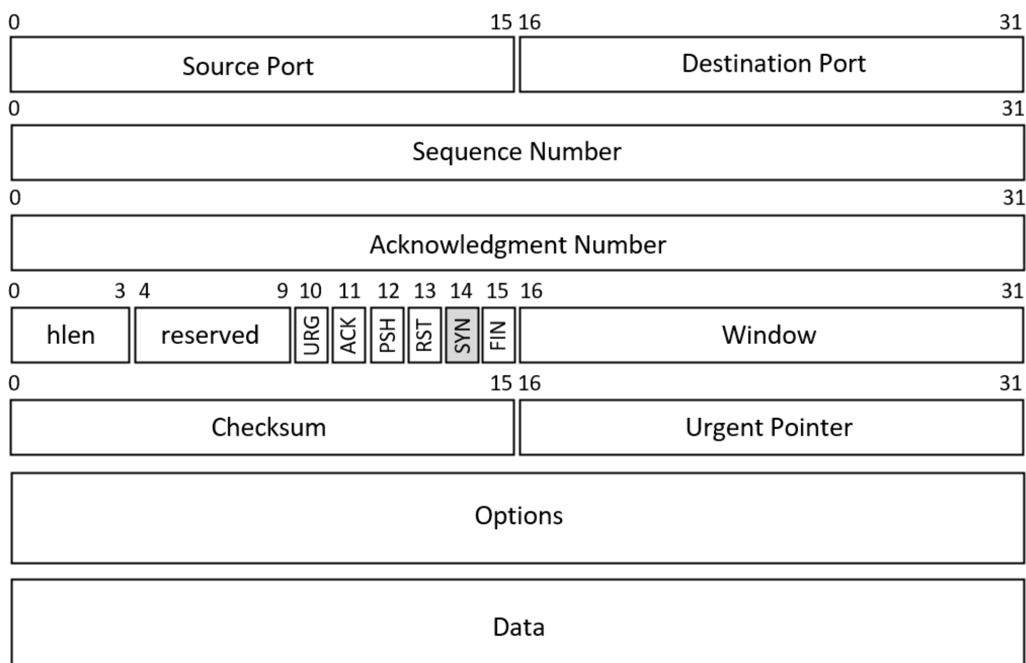


Figure 4.4: TCP Header Format.

synchronization flag and it is used to establish a three-way handshake between two hosts. Only the first packet from both the sender and receiver should have this flag set. Therefore, when the SYN flag is set to 1, the source host is trying to establish a new connection with the destination host. As explained in the initial part of this Chapter, the TCP SYN flood (or SYN flood) is a

type of Denial of Service (DoS) attack that exploits part of the normal TCP three-way handshake to consume resources on the targeted server and make it unavailable. The P4 program `antiDOS.p4` has the aim to prevent these types of attacks. If the SYN flag is set to 1, the lookup is performed on the `protected_server` MAT. If a match is found, the processed SYN packet is directed towards a protected server and consequently the `calc_hash` action is invoked. The hash index is computed and the relative value is read from the register `mycounter`. This value is, then, compared with a threshold (`MAX_SYN`) that, for the sake of simplicity, is set to 3. When the counter value, incremented for each connection, reaches the threshold, the packets are directly dropped. In this way, they never arrive at the end host and there isn't a waste of resources. Moreover, when the counter value corresponds to the threshold value, the `ingress_global_timestamp` standard metadata field is stored in the `meta.timemaxsyn` variable and it is written in a `time_last_maxsynpkt` register in order to be compared with the timestamps of the following packets. The V1Model Architecture `standard_metadata` have already been presented in Chapter 2. The `ingress_global_timestamp` field represents the timestamp relative to the moment in which the switch starts processing the packets, and it is measured in microseconds. For each following arriving packet, the time interval between the current timestamp and the timestamp stored in the register at the right index is computed and when this interval (`meta.interval`) reaches the predefined time threshold (`RESET_INTERVAL`), the counter value at the index specific for the couple sender-receiver is reset. By resetting the counter, the source host can try to establish a new connection with the destination host without any blocking policy. The `time_last_maxsynpkt` register has the purpose of keeping track of the ingress timestamp of the last packet corresponding to the achievement of the `MAX_SYN` threshold for a specific couple sender-receiver.
In the following code is implemented the just described `apply` block.

```
apply {
    if (hdr.ipv4.isValid()) {
        ipv4_lpm.apply();
    }
    if(hdr.tcp.isValid()) {
        if(hdr.tcp.syn == 1) {
            if(protected_server.apply().hit) {
                if (meta.count == MAX_SYN) {
                    meta.count = meta.count + 1;
```

```
                    mycounter.write((bit<32>)meta.pkt_hash,
                                     meta.count);
                    meta.timemaxsyn = standard_metadata.
                                    ingress_global_timestamp;
                    time_last_maxsynpkt.write((bit<32>)
                            meta.pkt_hash, meta.timemaxsyn);
                    drop();
                } else if (meta.count < MAX_SYN) {
                    meta.count = meta.count + 1;
                    mycounter.write((bit<32>)meta.pkt_hash,
                                    meta.count);
                    time_last_maxsynpkt.write((bit<32>)
                                    meta.pkt_hash,
                                    standard_metadata.
                                    ingress_global_timestamp);
                } else if (meta.count > MAX_SYN) {
                    time_last_maxsynpkt.read(meta.timemaxsyn,
                                    (bit<32>)meta.pkt_hash);
                    meta.interval = standard_metadata.
                                    ingress_global_timestamp -
                                    meta.timemaxsyn;
                    if (meta.interval > RESET_INTERVAL) {
                        meta.count = 0;
                        mycounter.write((bit<32>)meta.pkt_hash,
                                        meta.count);
                        meta.interval = 0;
                        drop();
                    }
                    else {
                        meta.count = meta.count + 1;
                        mycounter.write((bit<32>)meta.pkt_hash,
                                        meta.count);
                        drop();
                    }
                }
            }
        }
    }
}
```

The deparser, finally, should emit the Ethernet header, the IPv4 header, and the TCP header into the outgoing packet. In the code below the deparser, called `MyDeparser` is shown.

```
control MyDeparser(packet_out packet, in headers hdr) {
```

```
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
    }
}
```

For an initial test, some static rules are added at compile time in the `ipv4_lpm` table and the `protected_server` table. In the `s1-runtime.json` the following rules are written: the first ones are inserted in the `ipv4_lpm` table while the last one is inserted in the `protected_server` table and represents the host to be protected from DoS attacks (host h1).

ipv4_lpm table

Key	Action	Action Data
10.0.1.1/32	ipv4_forward	dstAddr = 08:00:00:00:01:11 port = 1
10.0.2.1/32	ipv4_forward	dstAddr = 08:00:00:00:02:22 port = 2
Default Action		
	NoAction	

protected_server table

Key	Action	Action Data
dstAddr ==10.0.1.1 dstPort == 65000	calc_hash	
Default Action		
	NoAction	

Figure 4.5: Match-Action Tables entries.

Figure 4.5 depicts the tables described above as well as the code below, that is written in the `s1-runtime.json` file.

```
{
  "target": "bmv2",
  "p4info": "build/antiDOS.p4.p4info.txt",
  "bmv2_json": "build/antiDOS.json",
  "table_entries": [

    {
      "table": "MyIngress.ipv4_lpm",
      "default_action": true,
      "action_name": "MyIngress.drop",
      "action_params": { }
    },

    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:01:11",
        "port": 1
      }
    },

    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.2.1", 32]
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:02:22",
        "port": 2
      }
    },

    {
      "table": "MyIngress.protected_server",
      "match": {
        "hdr.ipv4.dstAddr": ["10.0.1.1"],
        "hdr.tcp.dstPort": [65000]
      },
      "action_name": "MyIngress.calc_hash",
```

```
      "action_params": { }
    }
  ]
}
```

A simple way to test the behavior of the `antiDOS.p4` program inside the network topology represented in Figure 4.2 is the network tool `hping3`. `hping3` is a network tool able to send custom TCP/IP packets and display target replies like `ping` program does with ICMP replies. It allows simply performing monitoring and testing of firewalls and networks by generating and analyzing manipulated packets for the TCP/IP protocol.
To install `hping3` on Debian and its based Linux distributions including Ubuntu, it is possible to use the apt packages manager as shown below.

```
sudo apt install hping3 -y
```

A simple DoS attack can be performed by launching the following command in the h2 terminal:

```
hping3 -S --flood -s 65001 -k -p 65000 10.0.1.1
```

This command includes several useful options:

- `-S` : it is part of the TCP-related options and it is used to set the SYN flag to 1;
- `--flood` : it allows to send packets as fast as possible and replies will be ignored;
- `-s source_port` : it represents the port number of the host from which the traffic is sent;
- `-k` : usually the shown command is used with a *baseport* that is increased by one for each packet, while this option allows keeping the same source port for each sent packet;
- `-p destination_port` : it represents the port number of the destination host towards which the traffic is directed;
- the destination IP address.

The output does not show replies because they were ignored, but the `s1.log` file contains trace messages describing how the switch processes each packet. By inspecting the `s1.log` file it is, therefore, possible to verify the correct behavior of the switch s1. Figure 4.6 depicts the flowchart of the `antiDOS.p4` P4 program.

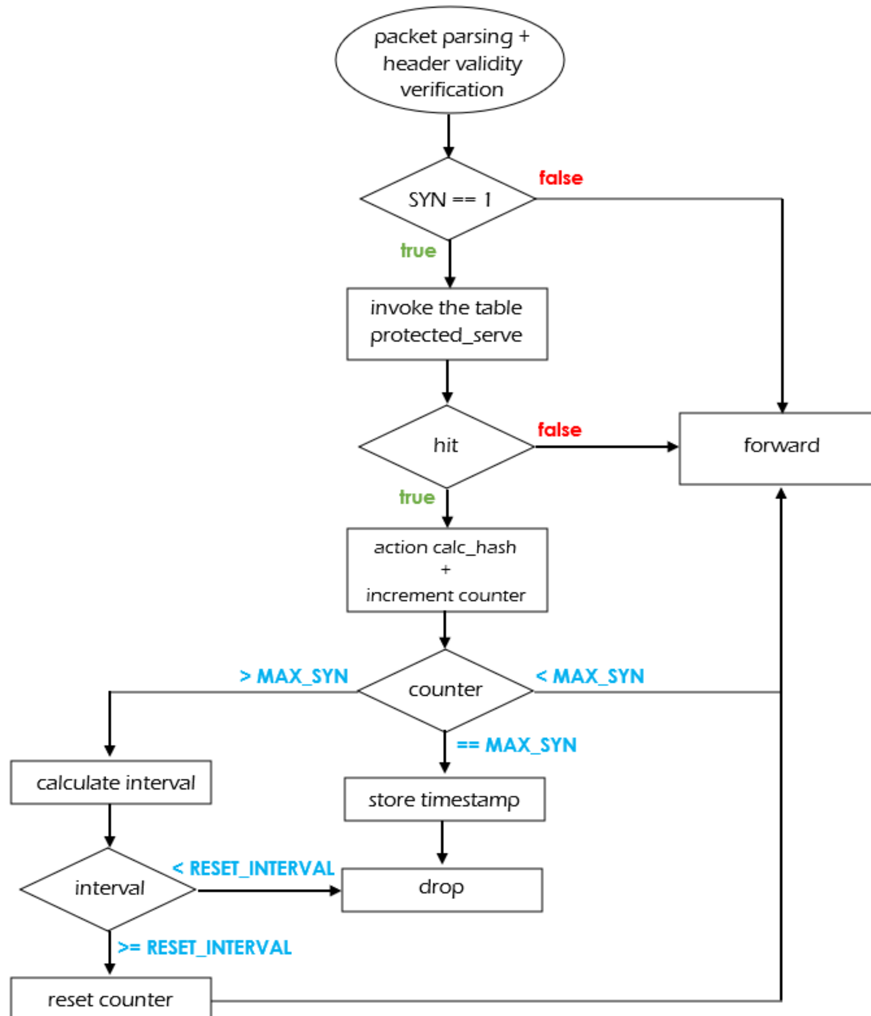


Figure 4.6: Flowchart.

4.2 P4Runtime Controller

The P4Runtime framework, already introduced in Section 3.4, is a control protocol for P4-defined data planes. Therefore, it includes control plane specifications for controlling the data plane elements of devices defined or described by a P4 program [24]. Figure 3.7 is proposed again in this Section with more details. Figure 4.7, indeed, represents a more detailed P4Runtime Reference Architecture. The device or target to be controlled is at the bottom, and one or more controllers are at the top of the schematized architecture. In the use case presented in this thesis, a single controller is implemented therefore Figure 4.7 shows only one entity with the role of controlling the target device. The P4Runtime APIs are target-independent, so the platform drivers and the

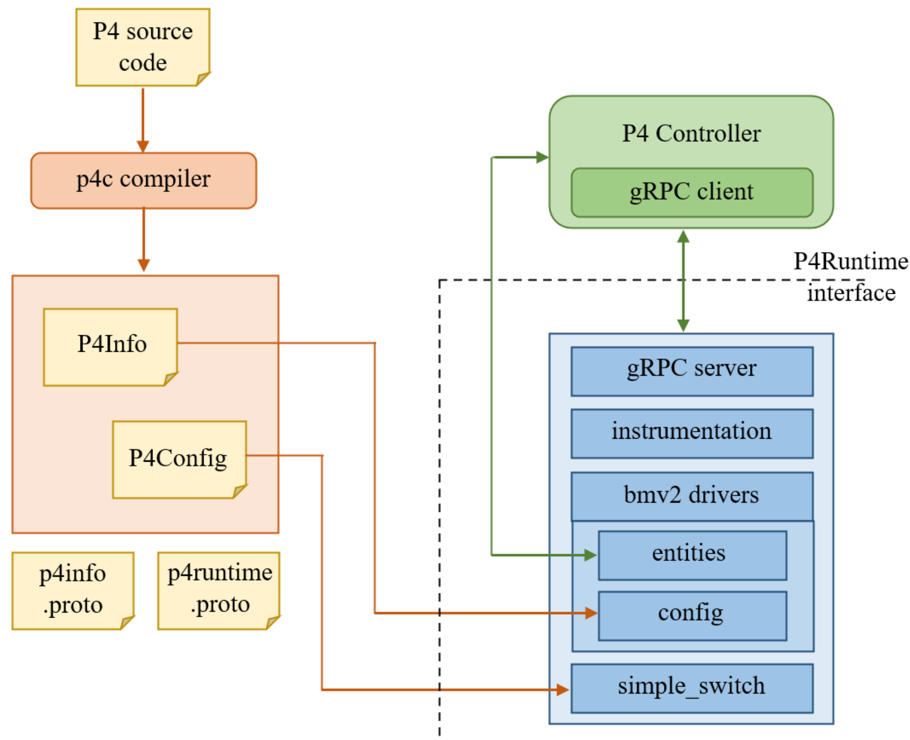


Figure 4.7: P4Runtime Reference Architecture.

specific target on which the controller is running are irrelevant from the point of view of the controller itself. For example, the messages utilized for inserting new table entries in the Data Plane are the same across multiple different

targets as long as the P4 program is the same.
By running the P4 compiler, a P4Info file and a JSON file are generated. The second one is specifically generated in the case of a BMv2 target, therefore it is target-specific, and it is used to configure BMv2 at runtime. The P4Info file, instead, is a target-independent compiler output that only depends on the P4 program. The P4Info file consists of a set of metadata that specifies the P4 entities which can be accessed via P4Runtime. These entities (Match-Action Tables, Actions, Externs, etc.) have a one-for-one correspondence with the instantiated objects in the P4 source code, and the P4Info file shares all their details and properties between the Data Plane and the Control Plane.
Figure 4.8, shows the piece of the `antiDOS.p4.p4info` file related to the `calc_hash` action and the `protected_server` table while in Figure 4.9 is shown the piece related to the registers `mycounter` and `time_last_maxsynpkt`.

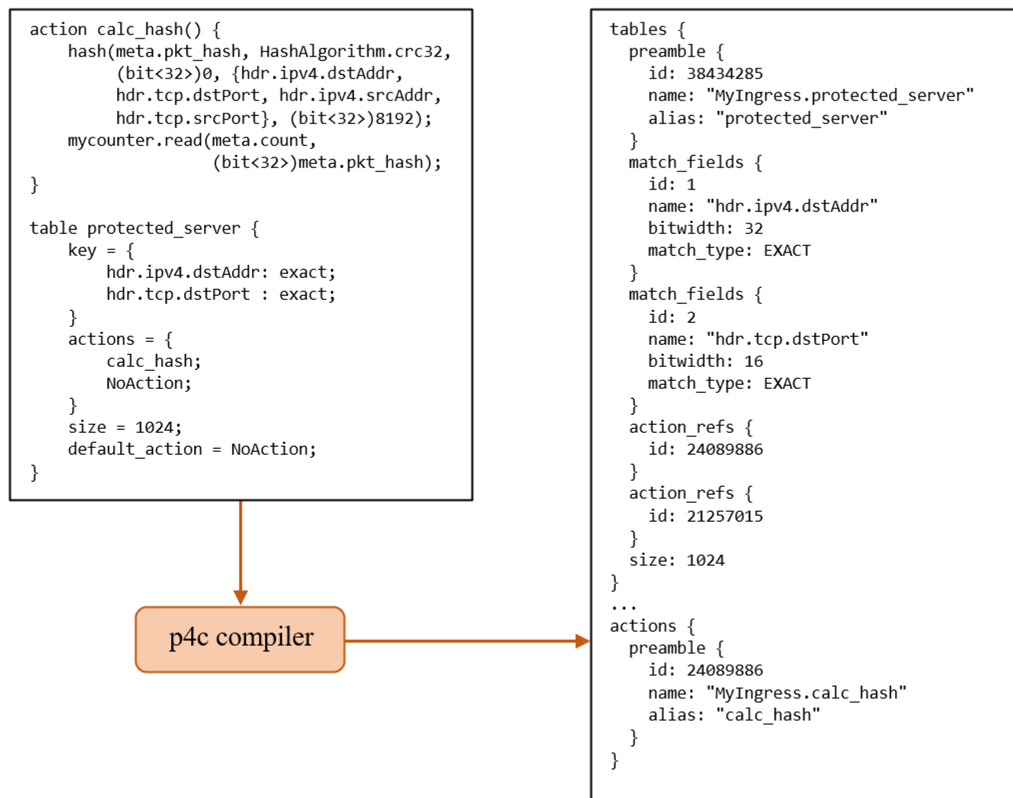


Figure 4.8: P4info metadata related to `protected_server` and `calc_hash`.

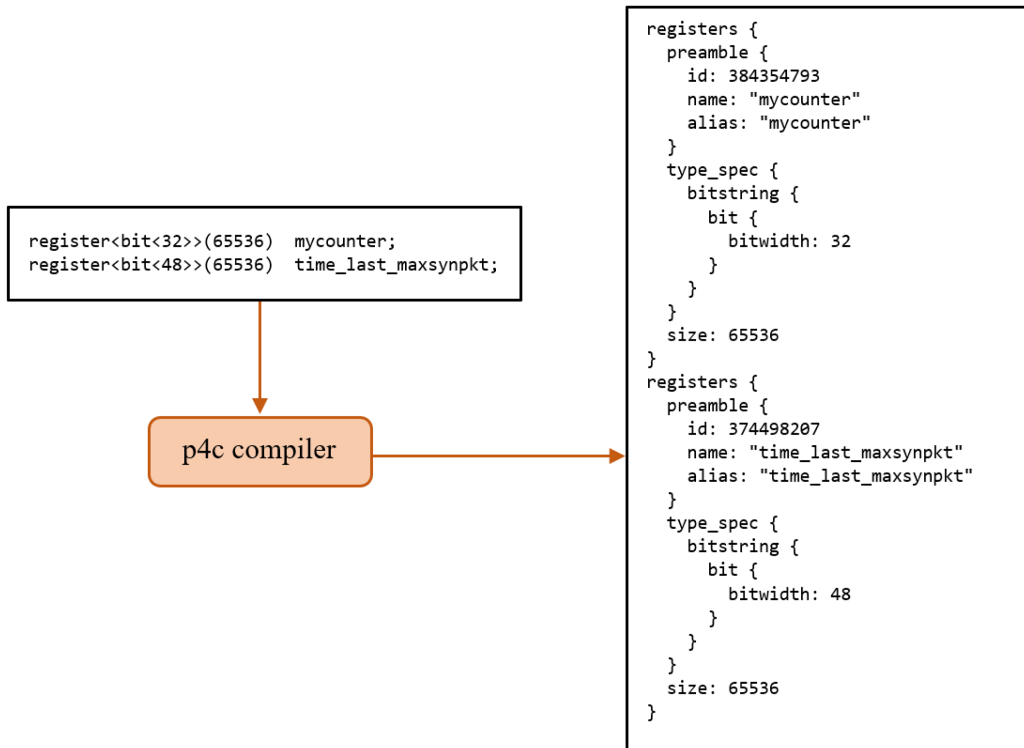


Figure 4.9: P4info metadata related to the registers.

As shown in Figure 4.7, P4 targets always include a gRPC Server and controllers should implement a gRPC Client for the connection between the Control Plane and the Data Plane.
The represented `p4runtime.proto` and `p4info.proto` are two protobuff files that describe respectively the API structure of P4Runtime and the P4Info file structure.
The implemented controller uses the same `p4runtime_lib` library already presented in Section 3.4, and therefore the previously described files `helper.py`, `switch.py`, `bmv2.py`, and `convert.py`. In particular, the script `switch.py` contains the `SwitchConnection` class that establishes connections to the switch and provides methods that construct the P4Runtime messages and makes the actual gRPC service call. Some of the utilized methods are listed in the following:

- `MasterArbitrationUpdate`: it establishes this controller as a master,

operation required by P4Runtime before performing any other write operation;

- `SetForwardingPipelineConfig()`: it installs the P4 program on the switch;
- `WriteTableEntry`: it builds the P4Runtime message needed to add a new table entry;
- `DeleteTableEntry`: it builds the P4Runtime message needed to remove an existing table entry;
- `ReadTableEntries`: it displays all the table entries.

In the implemented script `controller.py` three functions are defined: `updateForwardingRules`, `readTableRules`, and `updateAntiDOSRules`. All of them are shown in the code below.

```
def updateForwardingRules(mode, p4info_helper, sw, dst_ip_addr,
                          dst_eth_addr, egress_port):

    """
    Installs the antiDOS rule

    :param mode: add or del
    :param p4info_helper: the P4Info helper
    :param sw: the switch that performs the forwarding
    :param dst_ip_addr: the destination IP address to match on
    :param dst_eth_addr: ethernet address for next hop
    :param egress_port: egress port for next hop
    """

    # Forwarding Rule

    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.ipv4_lpm",
        match_fields={
            "hdr.ipv4.dstAddr": (dst_ip_addr, 32)
        },
        action_name="MyIngress.forward",
        action_params={
            "dstAddr": dst_eth_addr,
            "port": egress_port
        })
```

```
    if mode == "add":
        sw.WriteTableEntry(table_entry, False)
        print("Installed forwarding rule on %s" % sw.name)
    elif mode == "del":
        sw.DeleteTableEntry(table_entry, False)
        print("Deleted forwarding rule on %s" % sw.name)

def updateAntiDOSRules(mode, p4info_helper, sw, dst_ip_addr,
                       dst_port):

    """
    Installs the antiDOS rule

    :param mode: add or del
    :param p4info_helper: the P4Info helper
    :param sw: the switch connected to the protected server
    :param dst_ip_addr: the destination IP address to match on
    :param dst_port: the port number to match on
    """

    # AntiDOS Rule

    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.protected_server",
        match_fields={
            "hdr.ipv4.dstAddr": (dst_ip_addr),
            "hdr.tcp.dstPort":(dst_port)
        },
        action_name="MyIngress.calc_hash",
        action_params={

        })
    if mode == "add":
        sw.WriteTableEntry(table_entry, False)
        print("Installed antiDOS rule on %s" % sw.name)
    elif mode == "del":
        sw.DeleteTableEntry(table_entry, False)
        print("Deleted antiDOS rule on %s" % sw.name)

def readTableRules(p4info_helper, sw):

    """
    Reads the table entries from all tables on the switch.

    :param p4info_helper: the P4Info helper
```

```
    :param sw: the switch

    """
    print('\n--- Reading tables rules for %s ---' % sw.name)
    for response in sw.ReadTableEntries():
        for entity in response.entities:
            entry = entity.table_entry
            table_name = p4info_helper.
                         get_tables_name(entry.table_id)
            print('%s: ' % table_name, end=' ')
            for m in entry.match:
                print(p4info_helper.get_match_field_name(
                         table_name, m.field_id), end=' ')
                print('%r' % (p4info_helper.
                         get_match_field_value(m),),
                         end=' ')
            action = entry.action.action
            action_name = p4info_helper.get_actions_name(
                         action.action_id)
            print('->', action_name, end=' ')
            for p in action.params:
                print(p4info_helper.get_action_param_name(
                         action_name, p.param_id), end=' ')
                print('%r' % p.value, end=' ')
            print()
```

The `updateForwardingRules` function can be exploited to add or remove one or more entries from the `ipv4_lpm` Match-Action Table depending on the parameter called `mode`. The `updateAntiDOSRules` function, in the same way, can be exploited to add or remove one or more entries from the `protected_server` Match-Action Table. In both the functions a `table_entry` is built using the needed parameters (the match fields, the list of actions, and the action data) and then it is passed to the `WriteTableEntry` or the `DeleteTableEntry` function depending on the `mode` input parameter.
Through these functions it is possible, therefore, to reconfigure the switch at runtime by adding or removing table entries from the available Match-Action Tables. Finally, in the code below, the *main* function of the `controller.py` is shown and performs the following operations:

- instantiating a P4Runtime helper from the `antiDOS.p4.p4info` file. The P4InfoHelper provides translation methods from entity names to ID numbers and vice versa and therefore is used to parse the P4Info file.

- creating the P4Runtime gRPC connection to the switch s1.
- establishing the controller as master and, in this way, enabling the controller to write new entries in the tables.
- simply installing the P4 program on the switch, when the `mod` parameter is equal to `init` or it is not present.
- adding the three needed rules (two for forwarding purposes and one for protection purposes) when the `mod` parameter is equal to `add`.
- removing the rules mentioned above when the `mod` parameter is equal to `del`.

```
def main(p4info_file_path, bmv2_file_path, mod):

    # Instantiate a P4Runtime helper from the p4info file

    p4info_helper = p4runtime_lib.helper.P4InfoHelper
                                        (p4info_file_path)

    try:

        # Create a switch connection object for s1;

        s1 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
             name='s1',
             address='127.0.0.1:50051',
             device_id=0,
             proto_dump_file='logs/s1-p4runtime-requests.txt')

        # Establish this controller as master

        s1.MasterArbitrationUpdate()

        if mod == "init":

            # Install the P4 program on the switches

            s1.SetForwardingPipelineConfig(p4info=p4info_helper
                .p4info, bmv2_json_file_path=bmv2_file_path)
            print("Installed P4 Program using
                SetForwardingPipelineConfig on s1")
```

```
        elif mod == "add":

            # Write the forwarding rules in the s1 switch

            updateForwardingRules("add", p4info_helper, sw=s1,
                                  dst_ip_addr="10.0.1.1",
                                  dst_eth_addr="08:00:00:00:01:11",
                                  egress_port=1)
            updateForwardingRules("add", p4info_helper, sw=s1,
                                  dst_ip_addr="10.0.2.1",
                                  dst_eth_addr="08:00:00:00:02:22",
                                  egress_port=2)

            # Write the antiDOS rules in the s1 switch

            updateAntiDOSRules("add", p4info_helper, sw=s1,
                                  dst_ip_addr="10.0.1.1",
                                  dst_port=65000)

        elif mod == "del":

            # Delete the forwarding rules from the s1 switch

            updateForwardingRules("del", p4info_helper, sw=s1,
                                  dst_ip_addr="10.0.1.1",
                                  dst_eth_addr="08:00:00:00:01:11",
                                  egress_port=1)
            updateForwardingRules("del", p4info_helper, sw=s1,
                                  dst_ip_addr="10.0.2.1",
                                  dst_eth_addr="08:00:00:00:02:22",
                                  egress_port=2)

            # Delete the antiDOS rule in the s1 switch

            updateAntiDOSRules("del", p4info_helper, sw=s1,
                                  dst_ip_addr="10.0.1.1",
                                  dst_port=65000)

    except KeyboardInterrupt:
        print(" Shutting down.")
    except grpc.RpcError as e:
        printGrpcError(e)

    ShutdownAllSwitchConnections()
```

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(
                    description='P4Runtime Controller')
    parser.add_argument('--p4info', help='p4info proto in
                    text format from p4c', type=str,
                    action="store", required=False,
                    default='./build/antiDOS.p4.p4info.txt')
    parser.add_argument('--bmv2-json', help='BMv2 JSON file
                    from p4c', type=str, action="store",
                    required=False,
                    default='./build/antiDOS.json')
    parser.add_argument('--mod', help='updating mode',
                    type=str, action="store", required=False,
                    default='init')
    args = parser.parse_args()

    if not os.path.exists(args.p4info):
        parser.print_help()
        print("\np4info file not found: %s\nHave you run
                    'make'?" % args.p4info)
        parser.exit(1)
    if not os.path.exists(args.bmv2_json):
        parser.print_help()
        print("\nBMv2 JSON file not found: %s\nHave you run
                    'make'?" % args.bmv2_json)
        parser.exit(1)
    main(args.p4info, args.bmv2_json, args.mod)
```

4.3 Interaction with the orchestration system

In this last Section, the complete implemented system is presented. Figure 4.10 shows all the implementation layers.

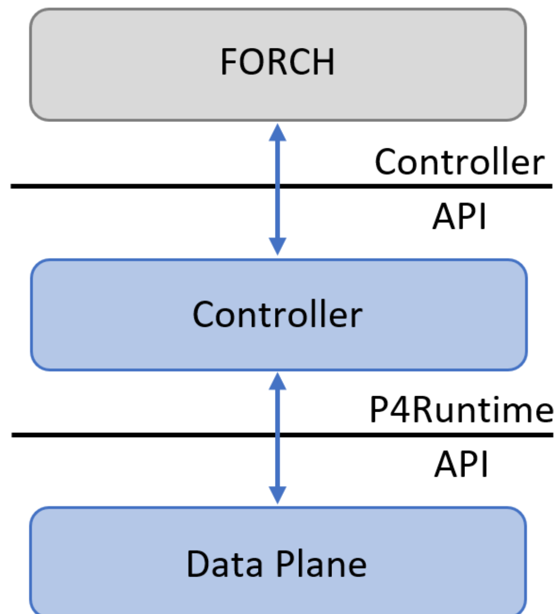


Figure 4.10: Complete system.

Starting from the bottom, the Data Plane layer consists of the P4 program `antiDOS.p4` installed on the switch (or switches) taken into account. The P4 program allows having installed in the switch the tables and the actions needed in order to perform the service that guarantees protection from DoS attacks. Through the P4Runtime APIs, the Data Plane can communicate with the Controller (the `controller.py` script) presented in Section 4.2. The Controller exhibits three main available functions: `updateForwardingRules` for adding a new table entry (into the `ipv4_lpm` forwarding table) or removing an existing table entry (from the `ipv4_lpm` forwarding table), `updateAntiDOSRules` for adding a new table entry (into the `protected_server` table) or removing an existing table entry (from the `protected_server` table), and `readTableRules` for reading all the table entries.

Finally, the controller can be integrated into the FORCH system (a system for Fog ORCHestration) described in Section 2.4.
The complete system is, in this way, able to dynamically reconfigure the targeted switches and, therefore, to provide in an efficient and flexible way new services whenever they are needed. In the presented use case, the system can dynamically provide the DoS prevention service for any connected target servers, at any moment, and in an efficient and fast way by simply calling the `updateAntiDOSRules` function of the Controller program with the correct parameters. In particular, for adding a new table entry in the `protected_server` Match-Action Table, the needed parameters to be passed to the `updateAntiDOSRules` function are the following: the `mode` parameter (that should be equal to `"add"`), the switch on which the rule has to be added, and the IP address and port number of the server that needs protection. It is also possible to dynamically remove the table entries, and therefore to disable the DoS prevention service for the targeted servers. In this case, the first parameter should be equal to `"del"` and the other ones remain the same.

Chapter 5

Performance Evaluation

This Chapter aims at presenting the performance evaluation of the system previously described in Chapter 4. The resources used to test the system behavior are: the Virtual Machine presented also in Chapter 3, the network emulation system Mininet, the BMv2 (Behavioral Model version 2) software switch and the p4c reference compiler.
Figure 5.1 depicts the behavior of the `antiDOS.p4` P4 program. In particular, the graphical representation of the program performance shows the packet rate captured on the two interfaces of switch s1: the packet rate of traffic incoming at the *eth2* interface (arriving from host h2) and the packet rate of traffic outgoing from the interface *eth1* (directed to host h1). The test performed to output the graph above is made by capturing on the two interfaces while the command

```
hping3 -S --interval 1 -V -s 65001 -k -p 65000 10.0.1.1
```

is running in the host h2 terminal.
This command includes the following options:

- `-S` : it is part of the TCP-related options and it is used to set the SYN flag to 1;
- `--interval sec` : it allows specifying the amount of time (in seconds) between which each packet is sent. In this test, the host h2 sends one SYN packet per second;
- `-V` : it enables verbose output. TCP replies will be shown with several details (e.g. IP address, flags, TTL, etc.);

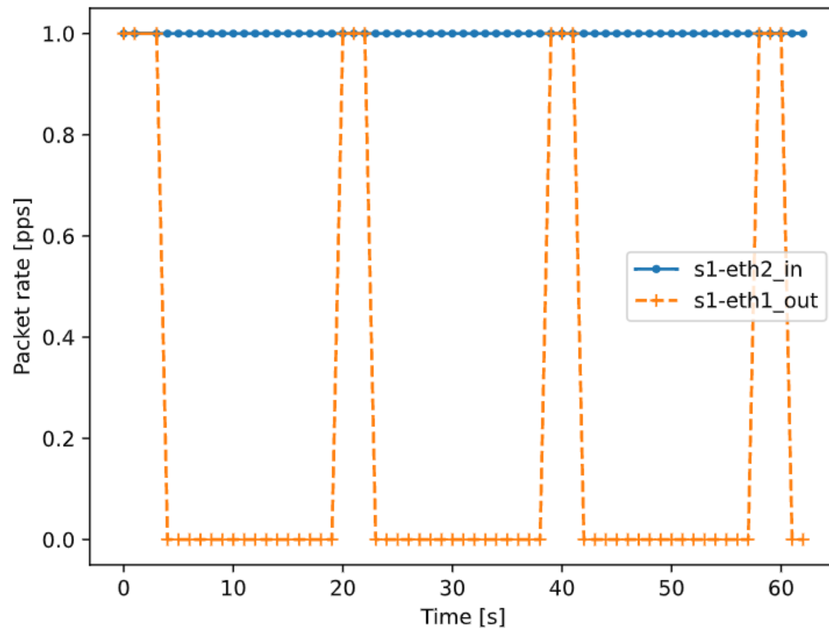


Figure 5.1: `antiDOS.p4` Program Performance.

- `-s source_port` : it represents the port number of the host from which the traffic is sent;
- `-k` : usually the shown command is used with a *baseport* that is increased by one for each packet, while this option allows keeping the same source port for each sent packet;
- `-p destination_port` : it represents the port number of the destination host towards which the traffic is directed;
- the destination IP address of the host h1.

One packet per second is sent by the host h2 and it is captured on the *eth2* interface of the switch, behavior represented by the blue dots in Figure 5.1.
At the same time, only a part of the packets is forwarded to the host h1 and captured on *eth1* interface, as it is possible to see by the trend of the orange crosses in Figure 5.1.
The parameters of the `antiDOS.p4` program are set as follows:

- The MAX_SYN parameter is set to 3. Therefore, after three SYN packets arrived from a certain IP address and port number and directed to the target IP address and port number, the traffic is not forwarded anymore toward the host h1. The successive packets which are characterized by the same 4-tuple (source IP address, source port number, destination IP address, and destination port number) are directly dropped by the switch.
- The RESET_INTERVAL parameter is set to 15000000 microseconds, which correspond to 15 seconds. After this amount of time, the packet counter is reset and host h2 is enabled again to establish new connections with host h1.

```
#define MAX_SYN 3
#define RESET_INTERVAL 15000000
                            /*microseconds -> 15 s*/
```

By launching the previously described command in these conditions, a cyclic behavior is exhibited: three packets are forwarded to host h1, then for 15 seconds all packets are dropped and never arrive to host h1, then three packets are forwarded to host h1, etc. This behavior is perfectly described by the graph in Figure 5.1.
A second test is performed in order to analyze the performance of the Controller. By modifying the code of the controller.py program as shown in the following pieces of code, the time needed to add, delete or reconfigure a rule or a set of rules is measured. The first test is made by measuring the time needed by the controller to insert one or more table entries in the different Match-Action Tables.
In Figure 5.2 the time needed to insert a forwarding rule into the ipv4_lpm table is measured for one hundred different rules. In the controller.py program this is achieved through the following changes to the code presented in Section 4.2. One hundred table entries are created and added to the forwarding table while the program stores the time before and after each table entry is inserted. As shown in Figure 5.2, the mean needed time to install a forwarding rule is 2.49 milliseconds.

```
timelist=[]
for n in range(1, 101):
    starttime = perf_counter_ns()
    updateForwardingRules("add", p4info_helper, sw=s1,
```

```
                              dst_ip_addr=f"10.0.1.{n}",
                              dst_eth_addr=f"08:00:00:00:{n:02x}:11",
                              egress_port=1)
    timelist.append(perf_counter_ns() - starttime)
print(timelist)
with open(str(pathlib.Path(__file__).parent.joinpath("results",
          "addforwsingle.txt")), "xt") as f:
    f.write("\n".join(str(t) for t in timelist))
```

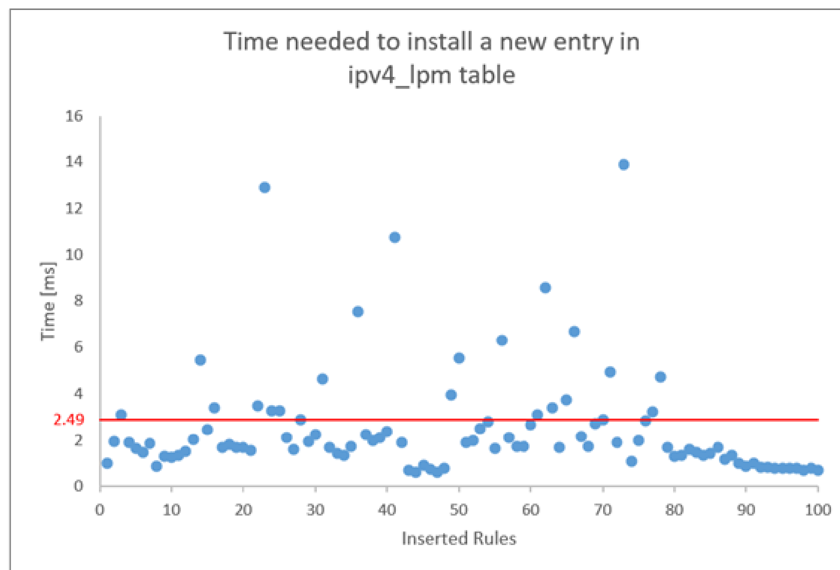


Figure 5.2: Insertion of a Forwarding Rule.

In Figure 5.3 the time needed to insert a DoS prevention rule into the `protected_server` table is measured for one hundred different rules. Also in this case some changes are made in the `controller.py` program. One hundred table entries are created and added to the `protected_server` table while the program stores the time before and after each table entry is inserted. As shown in Figure 5.3 the mean needed time to install a DoS prevention rule is 2.75 milliseconds.

```
timelist=[]
for n in range(1, 101):
    starttime = perf_counter_ns()
    updateAntiDOSRules("add", p4info_helper, sw=s1,
                       dst_ip_addr=f"10.0.1.{n}", dst_port=65000)
```

```
    timelist.append(perf_counter_ns() - starttime)
print(timelist)
with open(str(pathlib.Path(__file__).parent.joinpath("results",
          "addantiDOSsingle.txt")), "xt") as f:
    f.write("\n".join(str(t) for t in timelist))
```

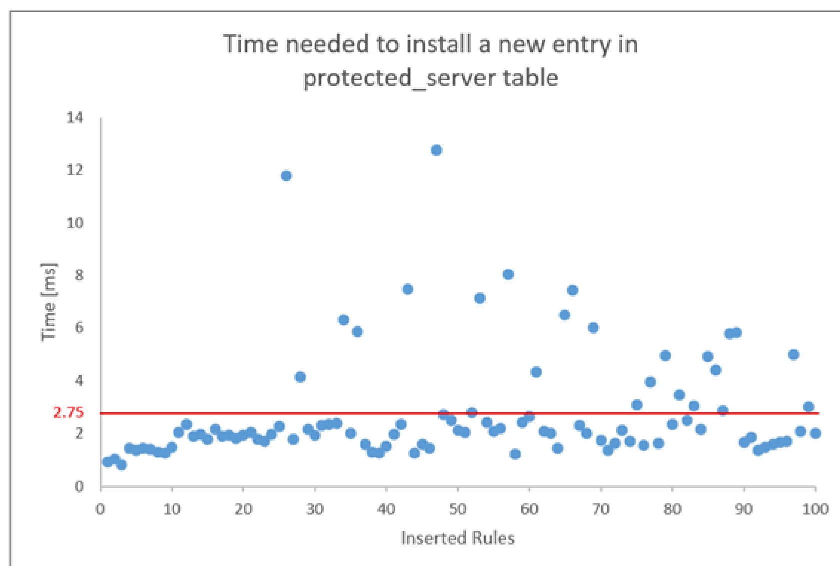


Figure 5.3: Insertion of a DoS Protection Rule.

In Figure 5.4 the time needed to insert a set of three rules is measured for one hundred different sets of rules. In particular, two forwarding entries are added to the `ipv4_lpm` table and one entry is added to the `protected_server`. Also in this case some changes are made in the `controller.py` program. One hundred sets of the three table entries are created and added to the two tables while the program stores the time before and after each set of table entries is inserted. As shown in Figure 5.3, the mean needed time to install a set made of the just mentioned rules is 4.02 milliseconds.

```
timelist=[]
for n in range(1, 101):
    starttime = perf_counter_ns()
    updateForwardingRules("add", p4info_helper, sw=s1,
                          dst_ip_addr=f"10.0.1.{n}",
                          dst_eth_addr=f"08:00:00:00:01:{n:02x}",
                          egress_port=1)
```

```
    updateForwardingRules("add", p4info_helper, sw=s1,
                          dst_ip_addr=f"10.0.2.{n}",
                          dst_eth_addr=f"08:00:00:00:02:{n:02x}",
                          egress_port=2)
    updateAntiDOSRules("add", p4info_helper, sw=s1,
                       dst_ip_addr=f"10.0.1.{n}", dst_port=65000)
    timelist.append(perf_counter_ns() - starttime)
print(timelist)
with open(str(pathlib.Path(__file__).parent.joinpath("results",
        "addcomplete.txt")), "xt") as f:
    f.write("\n".join(str(t) for t in timelist))
```

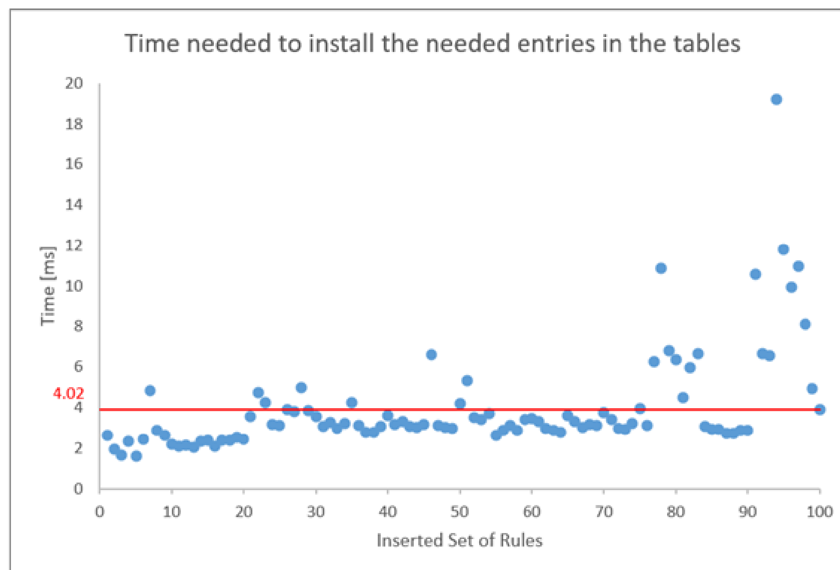


Figure 5.4: Insertion of a set of three Rules.

The same test is performed to measure the time needed to delete one or more table entries.
In Figure 5.5 the time needed to remove a forwarding rule from the `ipv4_lpm` table is measured for one hundred different rules. The `controller.py` program is modified as follows. One hundred table entries are deleted from the forwarding table while the program stores the time before and after each table entry is deleted. As shown in Figure 5.5 the mean needed time to remove a forwarding rule is 2.38 milliseconds.

```
timelist=[]
```

```
for n in range(1, 101):
    starttime = perf_counter_ns()
    updateForwardingRules("del", p4info_helper, sw=s1,
                          dst_ip_addr=f"10.0.1.{n}",
                          dst_eth_addr=f"08:00:00:00:{n:02x}:11",
                          egress_port=1)
    timelist.append(perf_counter_ns() - starttime)
print(timelist)
with open(str(pathlib.Path(__file__).parent.joinpath("results",
          "delforwsingle.txt")), "xt") as f:
    f.write("\n".join(str(t) for t in timelist))
```

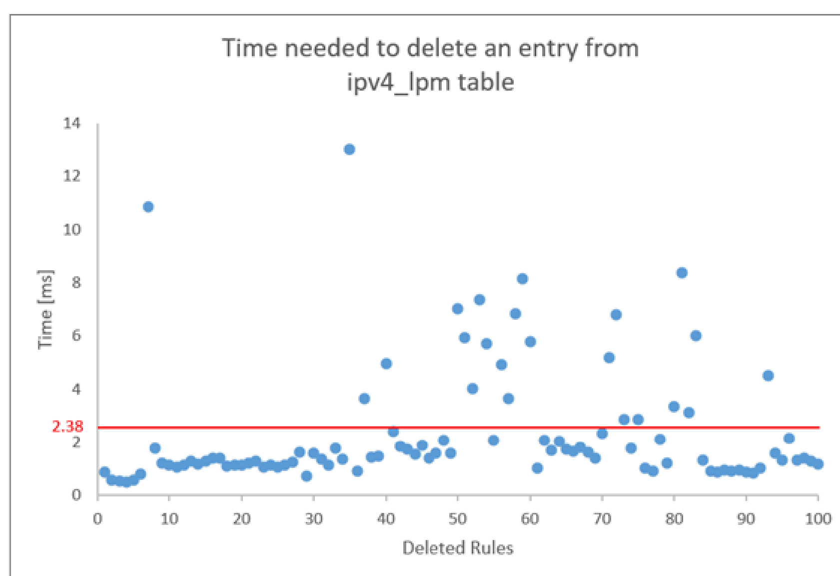


Figure 5.5: Removal of a Forwarding Rule.

In Figure 5.6 the time needed to remove a DoS prevention rule from the `protected_server` table is measured for one hundred different rules. Also in this case some changes are made in the `controller.py` program and they are shown in the code below. One hundred table entries are removed from the `protected_server` table while the program stores the time before and after each table entry is deleted. As shown in Figure 5.6 the mean needed time to delete a DoS Protection rule is 1.25 milliseconds.

```
timelist=[]
for n in range(1, 101):
```

```
    starttime = perf_counter_ns()
    updateAntiDOSRules("del", p4info_helper, sw=s1,
                        dst_ip_addr=f"10.0.1.{n}", dst_port=65000)
    timelist.append(perf_counter_ns() - starttime)
print(timelist)
with open(str(pathlib.Path(__file__).parent.joinpath("results",
          "delantiDOSsingle.txt")), "xt") as f:
    f.write("\n".join(str(t) for t in timelist))
```

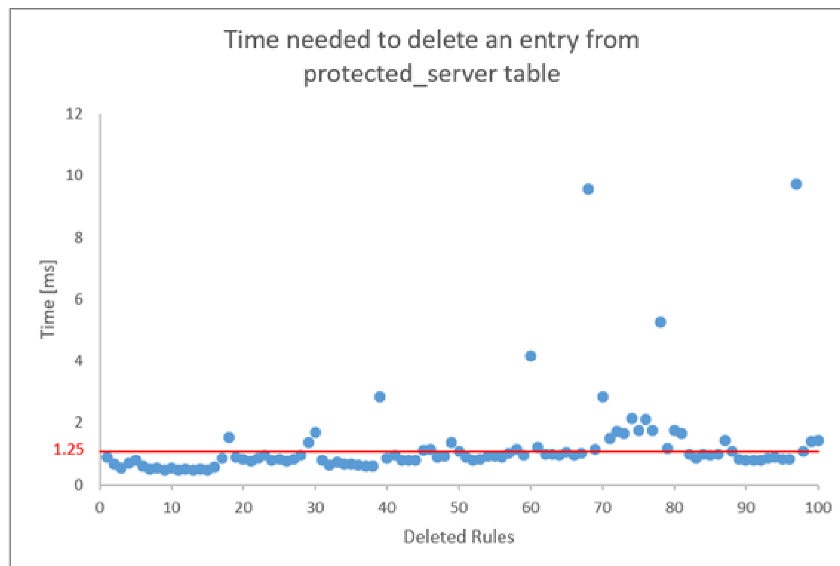


Figure 5.6: Removal of a DoS Protection Rule.

In Figure 5.7 the time needed to remove a set of three rules is measured for one hundred different sets of rules. In particular, two forwarding entries are removed to the `ipv4_lpm` table and one entry is removed from the `protected_server`. The changes made in the `controller.py` program are shown in the code below: one hundred sets of the three table entries are deleted from the two tables. As shown in Figure 5.7 the mean needed time to delete set of three rules is 8.75 milliseconds.

```
timelist=[]
for n in range(1, 101):
    starttime = perf_counter_ns()
    updateForwardingRules("del", p4info_helper, sw=s1,
                           dst_ip_addr=f"10.0.1.{n}",
```

```
                             dst_eth_addr=f"08:00:00:00:01:{n:02x}",
                             egress_port=1)
    updateForwardingRules("del", p4info_helper, sw=s1,
                             dst_ip_addr=f"10.0.2.{n}",
                             dst_eth_addr=f"08:00:00:00:02:{n:02x}",
                             egress_port=2)
    updateAntiDOSRules("del", p4info_helper, sw=s1,
                         dst_ip_addr=f"10.0.1.{n}", dst_port=65000)
    timelist.append(perf_counter_ns() - starttime)
print(timelist)
with open(str(pathlib.Path(__file__).parent.joinpath("results",
          "delcomplete.txt")), "xt") as f:
    f.write("\n".join(str(t) for t in timelist))
```

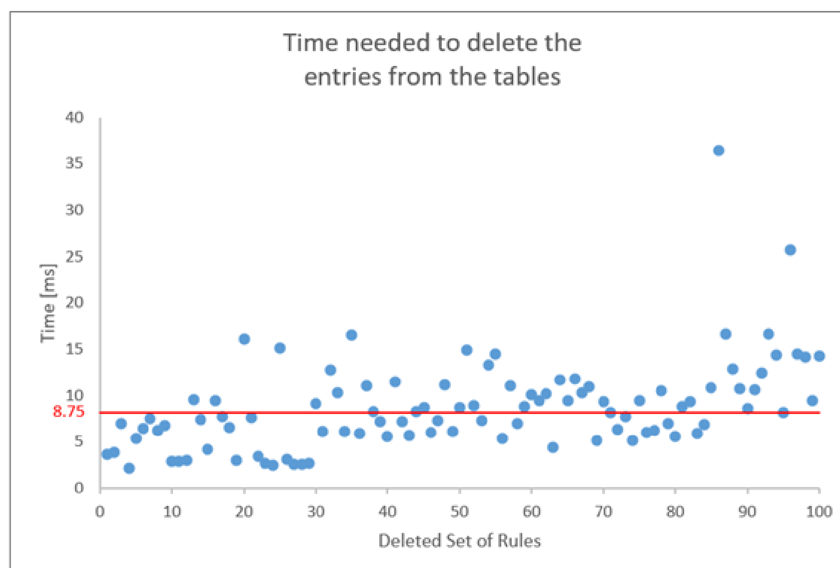


Figure 5.7: Removal of a set of three Rules.

Finally, the same test is made while reconfiguring table entries by removing and adding them again in the tables.
In Figure 5.8 the time needed to reconfigure a forwarding rule in the `ipv4_lpm` table is measured for one hundred different rules. The `controller.py` program is modified as follows. One hundred table entries are deleted from the forwarding table and added again to the same table. As shown in Figure 5.8 the mean needed time to reconfigure a forwarding rule is 4.27 milliseconds.

```
timelist=[]
for n in range(1, 101):
    starttime = perf_counter_ns()
    updateForwardingRules("del", p4info_helper, sw=s1,
                          dst_ip_addr=f"10.0.1.{n}",
                          dst_eth_addr=f"08:00:00:00:{n:02x}:11",
                          egress_port=1)
    updateForwardingRules("add", p4info_helper, sw=s1,
                          dst_ip_addr=f"10.0.1.{n}",
                          dst_eth_addr=f"08:00:00:00:{n:02x}:11",
                          egress_port=1)
    timelist.append(perf_counter_ns() - starttime)
print(timelist)
with open(str(pathlib.Path(__file__).parent.joinpath("results",
          "deladdforwsingle.txt")), "xt") as f:
    f.write("\n".join(str(t) for t in timelist))
```

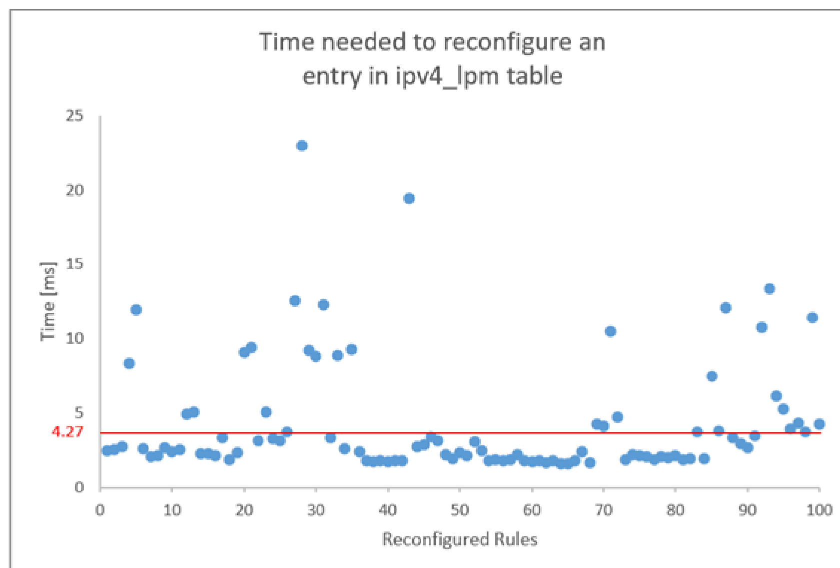


Figure 5.8: Reconfiguration of a Forwarding Rule.

In Figure 5.9 the time needed to reconfigure a DoS prevention rule in the `protected_server` table is measured for one hundred different rules. Also in this case some changes are made in the `controller.py` program and they are shown in the code below. One hundred table entries are removed and added

again in the `protected_server` table. As shown in Figure 5.9 the mean needed time to reconfigure a DoS Protection rule is 4.57 milliseconds.

```
timelist=[]
for n in range(1, 101):
    starttime = perf_counter_ns()
    updateAntiDOSRules("del", p4info_helper, sw=s1,
                       dst_ip_addr=f"10.0.1.{n}", dst_port=65000)
    updateAntiDOSRules("add", p4info_helper, sw=s1,
                       dst_ip_addr=f"10.0.1.{n}", dst_port=65000)
    timelist.append(perf_counter_ns() - starttime)
print(timelist)
with open(str(pathlib.Path(__file__).parent.joinpath("results",
        "deladdantiDOSsingle.txt")), "xt") as f:
    f.write("\n".join(str(t) for t in timelist))
```

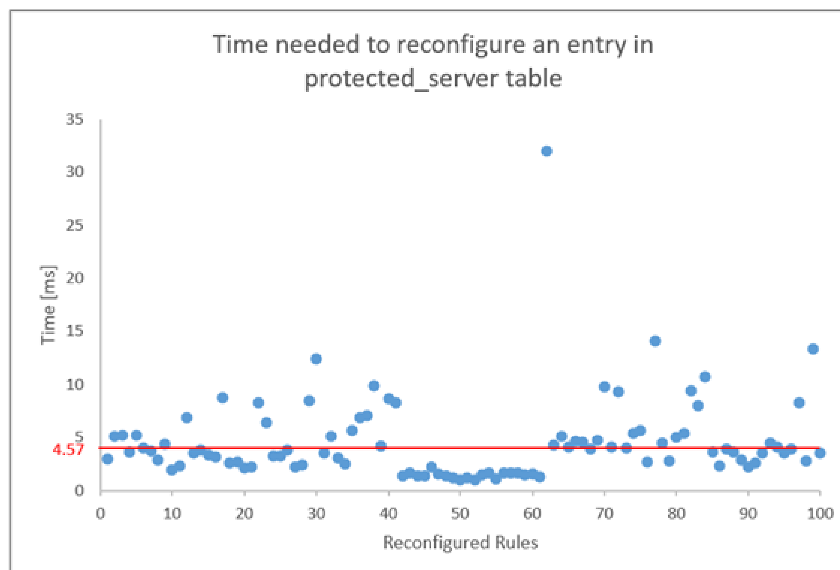


Figure 5.9: Reconfiguration of a DoS Protection Rule.

In Figure 5.10 the time needed to reconfigure a set of three rules is measured for one hundred different sets of rules. The changes made in the `controller.py` program are shown in the code below. One hundred sets of the three table entries are deleted and added again in the two tables. As shown in Figure 5.9 the mean needed time to reconfigure set of three rules is 16.20 milliseconds.

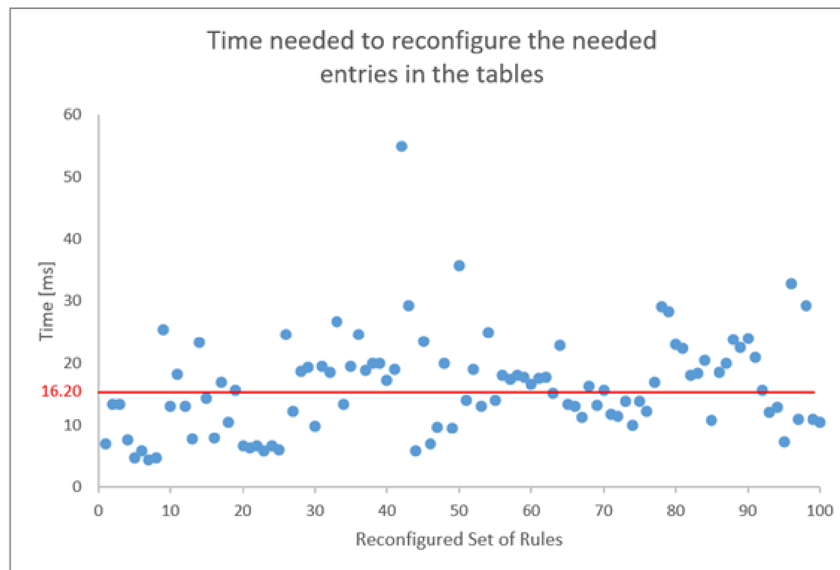


Figure 5.10: Reconfiguration of a set of three Rules.

```
timelist=[]
for n in range(1, 101):
    starttime = perf_counter_ns()
    updateForwardingRules("del", p4info_helper, sw=s1,
                          dst_ip_addr=f"10.0.1.{n}",
                          dst_eth_addr=f"08:00:00:00:01:{n:02x}",
                          egress_port=1)
    updateForwardingRules("del", p4info_helper, sw=s1,
                          dst_ip_addr=f"10.0.2.{n}",
                          dst_eth_addr=f"08:00:00:00:02:{n:02x}",
                          egress_port=2)
    updateAntiDOSRules("del", p4info_helper, sw=s1,
                       dst_ip_addr=f"10.0.1.{n}", dst_port=65000)
    updateForwardingRules("add", p4info_helper, sw=s1,
                          dst_ip_addr=f"10.0.1.{n}",
                          dst_eth_addr=f"08:00:00:00:01:{n:02x}",
                          egress_port=1)
    updateForwardingRules("add", p4info_helper, sw=s1,
                          dst_ip_addr=f"10.0.2.{n}",
                          dst_eth_addr=f"08:00:00:00:02:{n:02x}",
                          egress_port=2)
    updateAntiDOSRules("add", p4info_helper, sw=s1,
```

```
                    dst_ip_addr=f"10.0.1.{n}", dst_port=65000)
    timelist.append(perf_counter_ns() - starttime)
print(timelist)
with open(str(pathlib.Path(__file__).parent.joinpath("results",
         "deladdcomplete.txt")), "xt") as f:
    f.write("\n".join(str(t) for t in timelist))
```

Chapter 6

Conclusion

The adoption of the Data Plane Programmability paradigm can be an optimal solution in the Fog Computing scenario, allowing for great flexibility and adaptability in deploying services and managing resources.
In this thesis, a system for dynamic service provisioning is proposed, where the available resources may vary over time thanks to a Fog Orchestration system able to act as a resource management and service provisioning layer placed in between the service consumers and the Fog infrastructure. Moreover, the presented P4 program and the implemented controller guarantee a high degree of flexibility in reconfiguring the Data Plane devices, and therefore the devices functionalities and offered services.
The simple use case implementation presented in the previous Chapters demonstrates the feasibility of this dynamic service provisioning system, along with some experimental measurements. The implementation layers and all the components are described and discussed in detail, as it is for the final performance of each of them. As a result of this work, it is possible to notice that the insertion, deletion, and reconfiguration of one or more rules in the network devices (that corresponds to the effective service provisioning or service disabling) can be performed in a matter of few milliseconds. It is also shown a graphical representation of the traffic captured by a device on which the P4 program was installed, in order to demonstrate the operating principle and the effective performance.
The DoS prevention use case presented in this thesis is just an example of a service that can be dynamically deployed and deactivated depending on the needs of the network. This architecture can be further extended to many different

services in order to face the different needs of several application contexts.

Bibliography

[1] Gianluca Davoli, Walter Cerroni, Davide Borsatti, Mario Valieri, Daniele Tarchi, and Carla Raffaelli. “A Fog Computing Orchestrator Architecture With Service Model Awareness”. In: *IEEE Transactions on Network and Service Management* (Aug. 2022).

[2] *P4Tutorial*. June 2022. URL: https://github.com/p4lang/tutorials.

[3] Eder Ollora Zaballa, David Franco, Marina Aguado, and Michael Stübert Berger. “Next-Generation SDN and Fog Computing: A New Paradigm for SDN-Based Edge Computing”. In: *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Ed. by Anton Cervin and Yang Yang. Vol. 80. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 9:1–9:8. ISBN: 978-3-95977-144-3. DOI: 10.4230/OASIcs.Fog-IoT.2020.9. URL: https://drops.dagstuhl.de/opus/volltexte/2020/12003.

[4] Zifan Zhou, Eder Ollora Zaballa, Michael Stübert Berger, and Ying Yan. “Detection of Fog Network Data Telemetry Using Data Plane Programming”. In: *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Ed. by Anton Cervin and Yang Yang. Vol. 80. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 12:1–12:11. ISBN: 978-3-95977-144-3. DOI: 10.4230/OASIcs.Fog-IoT.2020.12. URL: https://drops.dagstuhl.de/opus/volltexte/2020/12006.

[5] Duy Thanh Le, Marcel Großmann, and Udo R. Krieger. *Cloudless Resource Monitoring in a Fog Computing System Enabled by an SDN/NFV Infrastructure*. workingpaper. 2022. DOI: 10.25972/OPUS-28072.

[6] Amirah Alomari, Shamala K. Subramaniam, Normalia Samian, Rohaya Latip, and Zuriati Zukarnain. “Resource Management in SDN-Based Cloud and SDN-Based Fog Computing: Taxonomy Study”. In: *Symmetry* 13.5 (2021). ISSN: 2073-8994. DOI: 10.3390/sym13050734. URL: https://www.mdpi.com/2073-8994/13/5/734.

[7] Bowei Guan and Shan-Hsiang Shen. “FlowSpy: An efficient network monitoring framework using P4 in software-defined networks”. In: *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*. IEEE. 2019, pp. 1–5.

[8] Takuji Tachibana, Kazuki Sawada, Hiroyuki Fujii, Ryo Maruyama, Tomonori Yamada, Masaaki Fujii, and Toshimichi Fukuda. “Open Multi-Access Network Platform with Dynamic Task Offloading and Intelligent Resource Monitoring”. In: *IEEE Communications Magazine* 60.8 (2022), pp. 52–58.

[9] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. “Modular switch programming under resource constraints”. In: *USENIX NSDI*. 2022, pp. 1–15.

[10] Aldo Febro, Hannan Xiao, Joseph Spring, and Bruce Christianson. “Edge security for SIP-enabled IoT devices with P4”. In: *Computer Networks* 203 (2022), p. 108698. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2021.108698. URL: https://www.sciencedirect.com/science/article/pii/S1389128621005612.

[11] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. “A survey on data plane programming with P4: Fundamentals, advances, and applied research”. In: *The Journal of Network and Computer Applications* (Dec. 2022). DOI: 10.1016/MCOM.2022.103561.

[12] *Introduction to $P4_{16}$ presentation*. URL: https://opennetworking.org/wp-content/uploads/2020/12/p4_d2_2017_p4_16_tutorial.pdf.

[13] *P4 Language and Related Specifications*. URL: https://p4.org/specs/.

[14] *P4 Language Consortium*. URL: https://p4.org/.

[15] *P4 Language Tutorials*. URL: https://docs.google.com/presentation/d/1zliBqsS8IOD4nQUboRRmF_19poeLLDLadD5zLzrTkVc/edit#slide=id.g37fca2850e_6_141.

[16] *The $P4_{16}$ Language Specification (v1.2.3)*. The P4 Language Consortium. July 2022. URL: https://p4.org/p4-spec/docs/P4-16-v1.2.3.pdf.

[17] *In-band Network Telemetry (INT) Dataplane Specification*. The P4 Language Consortium. Nov. 2020. URL: https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[18] *BEHAVIORAL MODEL (bmv2)*. URL: https://github.com/p4lang/behavioral-model.

[19] *p4c*. URL: https://github.com/p4lang/p4c.

[20] *Mininet. An Instant Virtual Network on your Laptop*. URL: https://mininet.org/.

[21] *Implementing Basic Forwarding*. URL: https://github.com/p4lang/tutorials/tree/master/exercises/basic.

[22] *Implementing Basic Tunneling*. URL: https://github.com/p4lang/tutorials/tree/master/exercises/basic_tunnel.

[23] *P4.org API Working Group Charter*. URL: https://p4.org/p4-spec/docs/P4_API_WG_charter.html.

[24] *The P4Runtime Specification (v1.3.0)*. The P4 Language Consortium. Dec. 2020. URL: https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.pdf.

[25] *p4lang/p4Runtime repository:P4Runtime Protobuf Definition Files and Specification*. URL: https://github.com/p4lang/p4runtime.

[26] *Implementing a Control Plane using P4Runtime*. URL: https://github.com/p4lang/tutorials/tree/master/exercises/p4runtime.

[27] *Implementing ECN*. URL: https://github.com/p4lang/tutorials/tree/master/exercises/ecn.

[28] *Implementing MRI*. URL: https://github.com/p4lang/tutorials/tree/master/exercises/mri.

[29] *Implementing a Basic Stateful Firewall*. URL: https://github.com/p4lang/tutorials/tree/master/exercises/firewall.