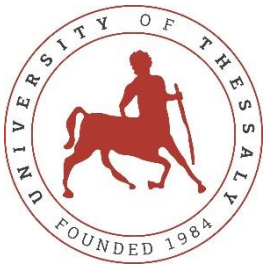


UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

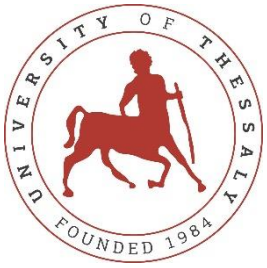
ONOS controller compatibility with P4 programming language for Software Defined Networks

Diploma Thesis

Panagiotis Pavlidis

Supervisor: Athanasios Korakis

February 2023



UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ONOS controller compatibility with P4 programming language for Software Defined Networks

Diploma Thesis

Panagiotis Pavlidis

Supervisor: Athanasios Korakis

February 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Υποστήριξη της P4 γλώσσας από τον ONOS ελεγκτή για δικτύωση καθορισμένη από λογισμικό

Διπλωματική Εργασία

Παναγιώτης Παυλίδης

Επιβλέπων: Αθανάσιος Κοράκης

Φεβρουάριος 2023

Approved by the Examination Committee:

Supervisor	**Athanasios Korakis** Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Antonios Argiriou** Assistant Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Dimitrios Bargiotas** Assistant Professor, Department of Electrical and Computer Engineering, University of Thessaly

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελούν αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλουν οποιασδήποτε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχουν έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Δηλώνω επίσης ότι τα αποτελέσματα της εργασίας δεν έχουν χρησιμοποιηθεί για την απόκτηση άλλου πτυχίου. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο Δηλών

Παναγιώτης Παυλίδης

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The Declarant

Panagiotis Pavlidis

Ευχαριστίες

Με την ολοκλήρωση της διπλωματικής μου εργασίας, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Αθανάσιο Κοράκη για την ευκαιρία και την εμπιστοσύνη που μου έδωσε αναθέτοντάς μου το συγκεκριμένο θέμα.

Είμαι ευγνώμων στον κ. Κώστα Χούμα, μεταδιδακτορικό ερευνητή του NITOS εργαστηρίου, για την υποστήριξη του και την εξαίρετη επικοινωνία που είχαμε σε όλη τη διάρκεια της διπλωματικής μου εργασίας.

Είμαι επίσης ιδιαίτερα ευγνώμων για τους καθηγητές Δημήτριο Μπαργιώτα και Αντώνιο Αργυρίου που διετέλεσαν ως μέλη της εξεταστικής επιτροπής της διπλωματικής μου εργασίας.

Ακόμα θα ήθελα να ευχαριστήσω όλους τους ανθρώπους που ήταν δίπλα μου σε όλα αυτά τα φοιτητικά μου χρόνια και που μοιραστήκαμε στιγμές, καλές και κακές, που με συντροφεύουν ήδη σαν αναμνήσεις και εμπειρίες του παρελθόντος.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου, που με στήριξε και με ενθάρρυνε σε κάθε μου απόφαση όλα αυτά τα χρόνια.

Σε ένα καλύτερο τώρα.

Acknowledgements

Upon completion of my thesis, I would like to thank my supervisor, Mr. Athanasios Korakis, for the opportunity and trust he gave me by assigning me this topic.

I am grateful to Mr. Kostas Choumas for his support and the excellent communication we had throughout this thesis.

I acknowledge the valuable contributions and guidance of Professors Dimitrios Bargiotas and Antonios Argiriou, who served as members of my thesis committee

I would also like to thank all the people who were by my side during all my student years and who shared moments, good and bad, that are already with me like memories and experiences of the past.

Finally, I would like to thank my family, who supported me and encouraged me in every decision I made all these years.

To a better now.

Περίληψη

Η ταχεία πρόοδος της τεχνολογίας έχει οδηγήσει σε αύξηση του αριθμού των συνδεδεμένων συσκευών και του όγκου των δεδομένων που μεταδίδονται μέσω δικτύων. Ως αποτέλεσμα, τα δίκτυα έχουν γίνει μια κρίσιμη υποδομή για την επικοινωνία και τη μεταφορά δεδομένων στον σημερινό κόσμο. Ωστόσο, με την αυξανόμενη πολυπλοκότητα και τη δυναμική φύση των δικτύων, οι παραδοσιακές μέθοδοι διαχείρισης και ελέγχου του δικτύου έχουν καταστεί ανεπαρκείς. Για την αντιμετώπιση αυτών των προκλήσεων, η Δικτύωση καθορισμένη από Λογισμικό (SDN) έχει εισαχθεί ως λύση.

Το SDN είναι μια αρχιτεκτονική δικτύου που διαχωρίζει το επίπεδο ελέγχου από το επίπεδο δεδομένων, επιτρέποντας κεντρικό έλεγχο, καλύτερη ορατότητα και βελτιωμένη ευελιξία του δικτύου. Ένα από τα βασικά πρωτόκολλα στο SDN είναι το OpenFlow, το οποίο επιτρέπει την επικοινωνία μεταξύ του επιπέδου ελέγχου και του επιπέδου δεδομένων, χρησιμοποιώντας ένα καλά καθορισμένο API.

Ένα άλλο πρωτόκολλο που έχει προκύψει στο χώρο του SDN είναι η P4. Η P4 είναι μια γλώσσα προγραμματισμού δικτυακών συσκευών που επιτρέπει τον άμεσο χειρισμό της συμπεριφοράς του επιπέδου δεδομένων του δικτύου, παρέχοντας υψηλό επίπεδο ευελιξίας και προγραμματισμού. Η P4 επιτρέπει επίσης την σχεδίαση της ροής αγωγών επεξεργασίας πακέτων, καθιστώντας τη ένα ιδανικό εργαλείο για την υλοποίηση νέων λειτουργιών δικτύου στο SDN. Οι παραπάνω τεχνολογίες περιλαμβάνονται στο ONOS.

Το λειτουργικό σύστημα ONOS SDN είναι ένας ευρέως χρησιμοποιούμενος ελεγκτής ανοιχτού κώδικα, με παραπάνω από έναν ONOS κόμβους και είναι επεκτάσιμος για τη διαχείριση και τον έλεγχο δικτύων, καθιστώντας το μια δημοφιλή επιλογή για τους χειριστές δικτύων.

Αυτή η διατριβή θα παρουσιάσει μια σύντομη εισαγωγή στο SDN και θα επικεντρωθεί στη μελέτη της γλώσσας P4 και του ONOS. Επιπλέον, μελετά την υποστήριξη της P4 στην πλατφόρμα ONOS.

Λέξεις-κλειδιά: Δίκτυα Υπολογιστών, Δίκτυα Επόμενης Γενιάς, SDN, Openflow protocol, P4, ONOS, SDN controller, P4Runtime API

Abstract

The rapid advancement of technology has led to an increase in the number of connected devices and the amount of data being transmitted over networks. As a result, networks have become a critical infrastructure for communication and data transfer in today's world. However, due to the increasing complexity and dynamic nature of networks, traditional network management and control methods have become inadequate. To address these challenges, Software Defined Networking (SDN) has been introduced as a solution.

SDN is a network architecture that separates the control plane from the data plane, allowing for centralized control, better network visibility, and improved network agility. One of the key protocols in SDN is OpenFlow, which enables communication between control and data plane, using a well-defined API.

Another protocol that has emerged in the SDN space is P4. P4 is a domain-specific programming language that allows for direct manipulation of the network data plane behavior, providing a high level of flexibility and programmability. P4 also lets the design of packet processing pipelines, making it an ideal tool for implementing new network functions in SDN. The above technologies are included in ONOS.

The ONOS SDN operating system is a widely used open-source, highly available and scalable controller for managing and controlling networks making it a popular choice for network operators.

This thesis will present a brief introduction to SDN and will focus in study of P4 language and ONOS as well. Moreover, it studies the support of P4 in ONOS platform.

Keywords:

Computer networks, Next Generation networks, SDN, Openflow protocol, P4, ONOS, SDN controller, P4Runtime API

Table of Contents

CHAPTER 1 INTRODUCTION

1.1 Motivation

The emergence of Software Defined Networking architecture (SDN) have made important changes to the way that networks are configured and operate. First of all, the SDN architecture defines two layers for the network management and a top layer that applications are deployed. From bottom to top we have the infrastructure layer (data plane) that physical network devices are resided and then comes the control plane which is responsible for the logic of the network in order to operate correctly. At the top layer, applications exist. Communication between data plane and control plane is made through the southbound API (SBI) and for the control plane and application layer through Northbound API (NBI). The most representative protocol in SDN world until now, is the Openflow protocol. It defines a standard, open interface, to populate the forwarding tables in network devices, i.e. switches or routers. The utilization of this interface can make different switches from different vendors to be controlled from one common control plane.

OpenFlow operates under the assumption that the behavior of switches is fixed and well-known, as outlined in the documentation of the switch ASIC. Historically, high-performance switch chips had been supported a specific set of protocols only, as they were directly implemented with IEEE and IETF standards in silicon. It was not possible to alter the behavior of these chips or add new protocols or methods for measuring and controlling the data path. Currently, it takes around four years to integrate a new protocol into a fixed-function ASIC [**[1]**]. The emergence of P4 language has been started to provide a solution for it.

P4 is a domain-specific language that changes the traditional approach of networking, which is based on the switch's vendor to determine the limited set of operations it can perform. Instead, in P4, network architects and programmers instruct the switch about what it should do and how it should process packets. P4 gives to them capability to define the headers that switch can recognize, how to match on each header, and actions that switch should perform on each header.

P4 language offers more flexibility and upgradability compared to other solutions like traditional fixed-function switches or hardware solutions. With its programmability, it provides space for innovation in the enterprise networks, like flexibility of the network stack and the ability to update the software without needing to purchase new switches.

1.2 Thesis Outline

The rest of this thesis are organized as follows: Chapter 2 presents an overview around SDN, Openflow protocol and its challenges. In chapter 3, the P4 language and P4Runtime API is covered to a high extent. In chapter 4, ONOS SDN controller is described thoroughly. In chapter 5, the support of P4 in ONOS is presented whereas chapter 6 presents a summary of the conclusions of the current study and outlines plans for further research.

CHAPTER 2 SOFTWARE DEFINED NETWORKING

2.1 Overview

Before the emerging technology of Software Defined Networking (SDN), telecommunication networks were complex and difficult to manage. Network devices ran complex, distributed control software that was closed and proprietary and varied across vendors [**[2]**]. On top of that, there was specific configuration interfaces which varied across vendors or across different products from the same vendor and network devices should be instructed individually through them to operate correctly. This approach had made telecommunication networks difficult to manage and innovation on this field was infeasible.

Software Defined Networking has played an important role in today's networks. Its scope is to provide a flexible way for designing and managing networks. To do this, network control should be decoupled from network infrastructure as well as operate individually and being directly programmable [**3]**. This makes an abstraction on the network infrastructure that can realize its network as a logical unit.

2.2 SDN Architecture

As we stated above, SDN separates the network control which is called control plane from network infrastructure which is called data plane in a manner that control plane as a whole is centralized and manages the entire network. To be more specific, control plane decides how to handle network traffic and installs flow entries in the data plane devices that controls. On the other hand, data plane is responsible for how to process packets that flow through the network and forwards them according to flow rules that control plane has installed to it.

The communication between control plane and data plane is made through well-defined open source APIs that are developed and standardized through the advance of SDN. API that operates to the connection and communication between control and data plane is called a Southbound API, while the API that communicates the control plane with SDN

applications that are built on top of that is called Northbound API. The above description about SDN architecture and the components that makes it alive is illustrated in figure 1.

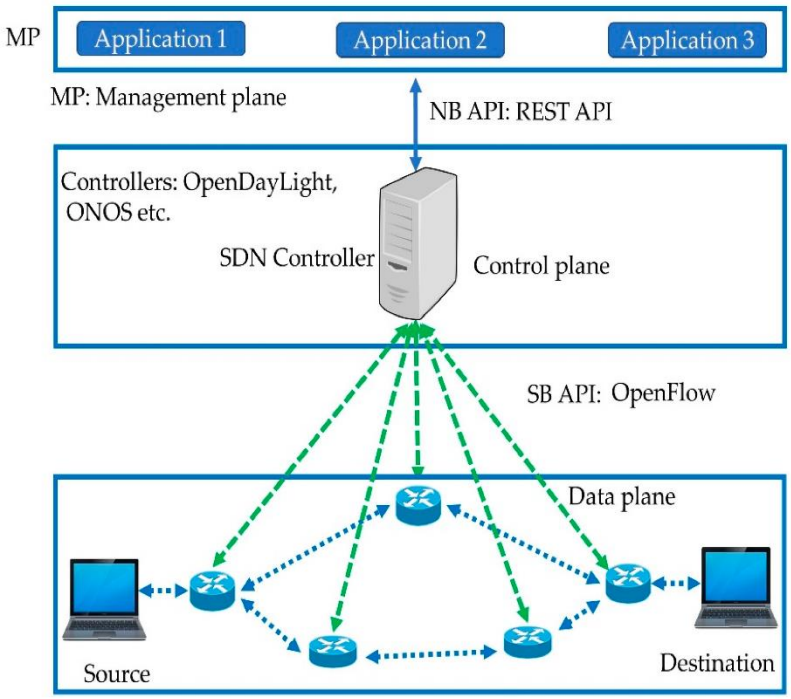


*Figure 1 SDN Architecture [**4**]]*

2.3 Openflow

The aspect of software defined networking has been become alive with the aid of Openflow protocol. It is a standardized protocol by Open Networking Foundation (ONF) and provides the communication channel between controller and infrastructure device that is called Openflow switch in terms of OpenFlow protocol.

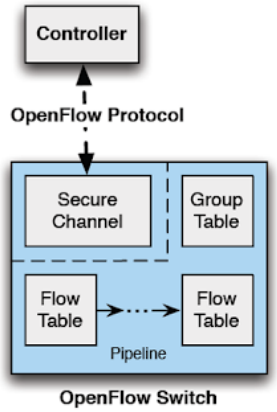


Figure 2 Components of an Openflow Switch

As shown in the figure 2, Openflow switch consists of one or more flow tables and a group table which perform packet lookups and forwarding. Also provides a secure communication channel to communicate with the controller through the Openflow protocol.

It is worth to mention that OpenFlow switch has a pre-existing ASIC design and provides only the interfaces on which the communication with the controller can be achieved. The usage of Openflow protocol offers to the controller the ability to add, delete or modify flow entries in the flow tables of the switch either proactively (before packet injection in the network) or reactively (in response to packets) [**5**]. This is achieved due to instruction set that Openflow provides for controlling the Openflow-enabled switches.

Due to its success to manage the network, it has become widely popular in academia, research and industry and many SDN controllers have been developed with support of it.

2.4 Challenges in SDN

While SDN continues to be developed, there are also some challenges associated with this emerging technology that will need to mention. The main challenges that the SDN technology has, is about reliability, scalability, performance, interoperability and security.

- Reliability
    - Due to its centralized architecture, an SDN controller can become a single point of failure. As a result the entire network which controls may collapse [**6**]. To deal with it, the controller should operate as a cluster with more than one controller instances as a primary-backup scheme. In this case, there should be a mechanism to maintain consistency with primary and backup controller's data.
- Scalability
    - The separation between control plane and data plane has established an individual development for both planes as long as the southbound API connects them. However, when it comes to scale up one of those planes this independent development can have drawbacks. For example, when the

network scales up in the number of switches and the number of end hosts, the SDN control plane (controller) can become a key bottleneck.

- Performance
    - The controller is able to response either proactively or reactively to setup flows. In reactive approach, the setup time to configure a flow in the switch is not negligible. If the reactive method used for a thousand number of flows, then it will be created performance issues on the switch during setup time of them.
- Interoperability
    - While SDN emerges, there are still legacy networks that are in live mode. Thus any new installation of an SDN network should be compatible with traditional networks and their communication interfaces [**7**].
- Security
    - The controller is a target for threats especially when open to unauthorized access. Attacks on the controller can cause serious damage to the network, as it is responsible for controlling the entire network. Authentication and authorization at the controller-application level are required to support network protection.

CHAPTER 3 PROGRAMMING PROTOCOL INDEPENDENT LANGUAGE

3.1 Overview

Programming Protocol Independent Language (P4) is a domain specific language that specifies how network devices process packets in the data plane [**8**]. These devices can be hardware and software switches, routers, NPUs, FPGAs, that are called as targets in P4 term. It is designed to overcome the problem that fixed function switches have in terms of reconfigurability, protocol and target independence [**9**]. The aforementioned problems of fixed function switches are the three main goals of P4 that it tries to solve:

- **Reconfigurability in the field**
    - Once a switch is deployed, it should be able to change how to process packets at any time through the communication channel between his control plane and itself.
- **Protocol independence**
    - There are no predefined network protocols as opposed to Openflow. Instead, headers that describe the protocols that will be used to a P4 program should be specified explicitly.
- **Target independence**
    - As a high level domain specific language, it is designed to operate exactly the same way on every target without take into consideration the underlying hardware. The translation of the target-independent description to target specific representation it is made by a compiler.

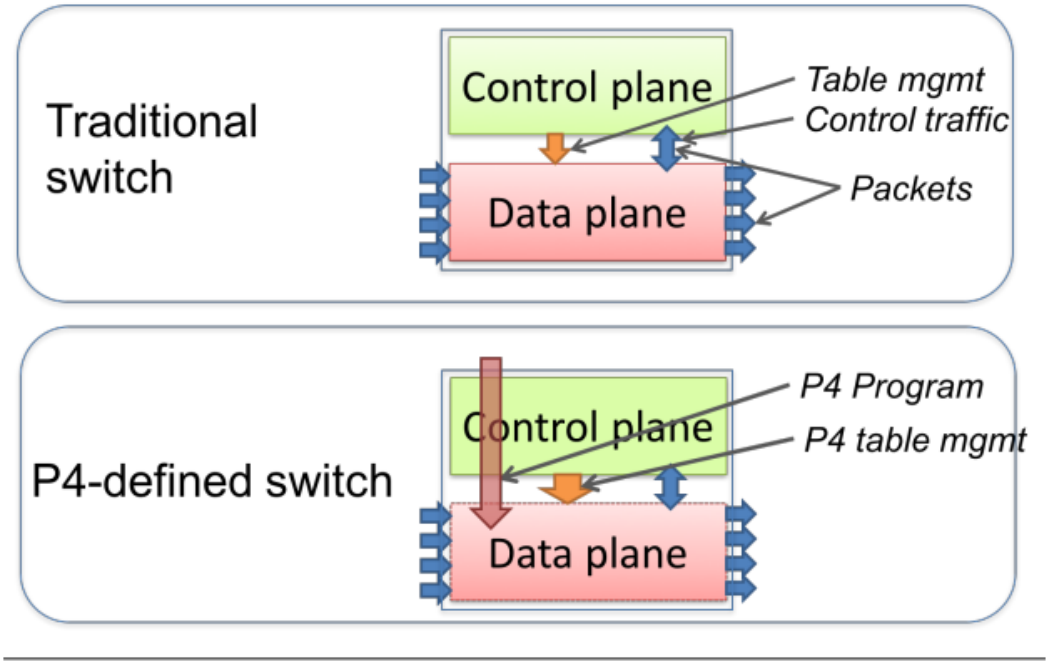


Figure 3 Traditional vs P4 switches

The differences between traditional and P4-defined switches in terms of the configuration of the data plane it can be seen in figure 3. The latter defines the pipeline of the data plane during initialization phase of the P4 program that is executed. Also it is made clear that the set of tables that are exist in a P4 switch versus a traditional switch are depended of the P4 program and not of the target's implementation.

3.2 P4 Workflow

The P4 ecosystem in order to come up live and start working correctly all it needs is a P4 program, a P4 compiler and a P4 target that will run the compiled P4 program. To begin with P4 compiler, there is a reference compiler called p4c [**10**]. This compiler provides a standard front-end and mid-end compiler and can be combined with a target-specific backend compiler to produce a complete P4 compiler. The compiler takes as input the P4 program that is programmed before and can generate 2 distinct files. A binary configuration file for switches and a mapping of tables and actions defined in the P4 program in a format that can be consumed by the control plane. This file is called p4info and provides the means to populate P4 switch's tables with entries. The above description of the P4 workflow is depicted clearly in the figure 4.

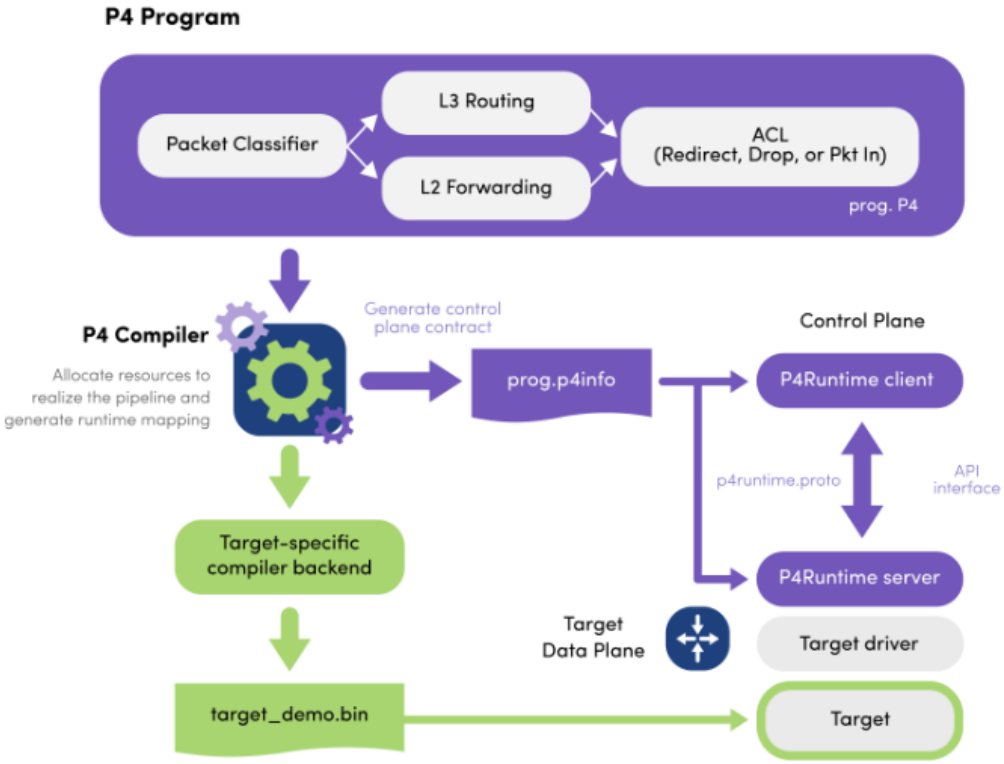


*Figure 4 P4 workflow compiling process [**11**]*

As for P4 targets, there are many different types of devices that can act be supported. To demonstrate a concrete example and understanding of the P4 workflow the BMv2 software switch will be introduced [**12**]. It is a P4 reference switch that was written in C++11. As it is described before, the P4 compiler generates a target specific configuration in order to implement a packet-processing behavior that was defined by a P4 program. In the case of this software switch, configuration comes in JSON format that it is imported as an input to it. It is worth to mention that BMv2 is not meant to be a production grade software switch rather than a tool for prototyping features in P4 that need to be tested. As a result it lacks of performance in terms of throughput and latency against other software switches that expected to be production grade. There are many cases that impact the performance of BMv2, it could be noteworthy a few, regarding the complexity of the P4 program with respect to the amount of headers that were defined to it, the compiler that was used to generate the BMv2 JSON file as well as the version of the software switch.

There are a few notable targets like simple_switch, simple_switch_grpc [**13**]. Basically, these targets are very similar to P4 features that support but they are different in the way that communicate with the control plane and the format of control messages that are sent to the controller. The former accepts TCP connections from a controller with format of the control messages is defined by a Thrift API. On the other hand, the latter accepts TCP connections from a controller using the P4Runtime API for the communication between control and data plane in a message format that the API recognizes. A last thing to mention, is that if P4Runtime API is the selected communication among a switch and a controller, the simple_switch_grpc should be used as a Bmv2 software switch.

3.3 P4 Architecture

P4 targets have architectural models that describe them in terms of the functional blocks that exist in the data plane [**14**]. These architectural models define the P4 programmable blocks as well as the fixed function blocks that coexist in a target's data plane. It also exposes each block's interfaces and their capabilities. Each target can support multiple architectures but a P4 program is not portable across different architectures. In case of targets that support the same architecture, P4 program runs the same for such a target. There are various P4 architectures that are implemented on different targets such as SimpleSumeSwitch architecture which is supported by NetFPGA based devices, v1model architecture that is supported by software switches like BMv2 and much more that expose the portability feature of the P4. Except of multiple architectures, P4, as well as its architectural models, provides data plane interfaces in order a block and other components of the architecture to communicate with each other.

These interfaces can be expressed by fields that indicate the direction of the information in the specified block. The code block 1 section specifies a control block with parameters defined as:

- in → indicates an input value in the block that can only be read
- out → indicates an output value from the block that can be modified and is initially undefined

- inout → indicates that this value acts both as input and output and have the characteristics of both in and out parameters

```
control MatchActionPipe(in bit inputPort, inout H parsedHeaders, out bit outputPort);
```

Code Block 1 P4 interface description

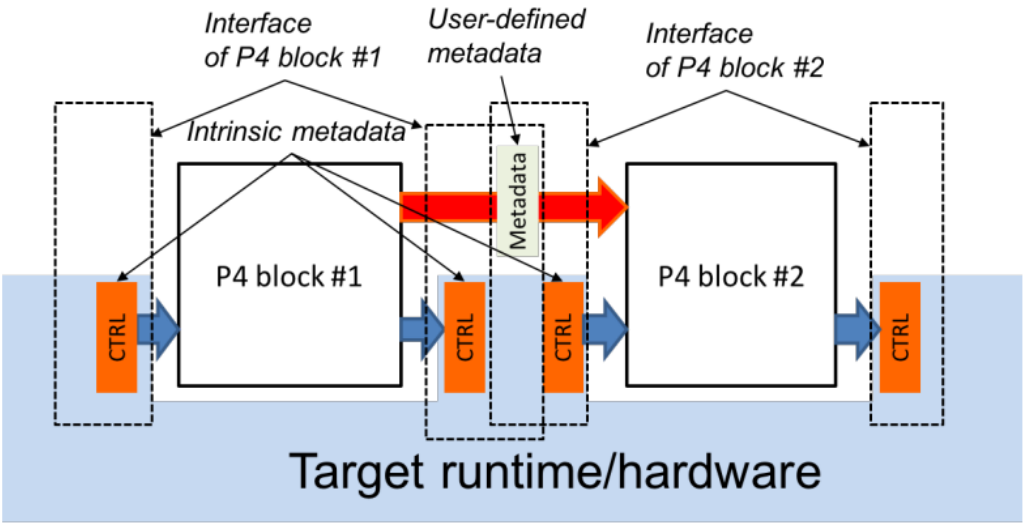


Figure 5 P4 programmable blocks and interfaces

An architecture also provides a set of extern functions and objects that can be used to interact with a P4 program without knowing the implementation of the provided functions and objects. This is a known technique similar to the abstract class of an object-oriented programming model and offered to P4 using P4 extern objects and their functions. An example of such an extern object and its abstract methods is depicted below for a checksum operation.

```
extern Checksum16 {

        Checksum16(); // constructor

        void clear(); // prepare unit for computation

        void update(in T data); // add data to checksum

        void remove(in T data); // remove data from existing checksum bit

        get(); // get the checksum for the data added since last clear

}
```

Code Block 2 P4 extern object definition

3.4 V1Model

This thesis uses the BMv2 software switch which supports the V1Model P4 architecture, in order to investigate further the P4 language. For a better elaboration in P4 architectures, this model will be analyzed. It is worth to mention the programmable and fixed function blocks of a P4 program for the V1Model Architecture which depicted in figure 6.

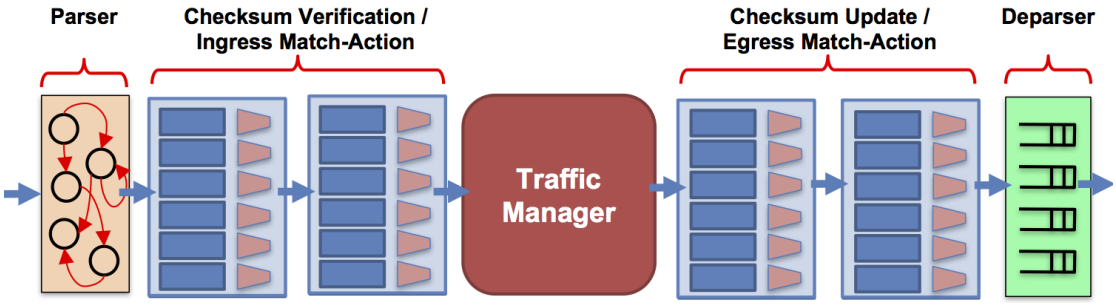


Figure 6 V1model Architecture

Programmable blocks consist of the Parser, Checksum Verification, Ingress Match-Action tables, Checksum Update, Egress Match-Action tables and Deparser. The Traffic manager block comes as a fixed function block provided by the manufacturer and is accompanied with tasks such as queue management, packet scheduling and replication and is out of the scope of this thesis. The functionality of these blocks and metadata fields that are used in the V1Model architecture will be described in the following sections of this chapter.

3.4.1 Standard Metadata

These fields comes with the V1Model as part of the **struct standard_metadata_t** and are associated with a packet**.**

Some fields that is important to be noted are:

- ingress_port → A read-only field that specifies the device's port that a packet comes from. It is an important field for decisions related to ingress or egress match-action tables and parser.

- egress_spec → This field specifies which output port will be used in order a switch to forward a packet. It is assigned during the Ingress control processing.

- egress_port → This field is read-only and is used strictly on egress processing which its value maps to the value of egress_spec that is used in the ingress processing.

- mcast_grp → This field is used on the ingress processing for the multicast feature. A value of zero defines no multicast, otherwise it should have a valid value of a multicast group that is defined to the bmv2 runtime interfaces.

3.4.2 Parser

The job of a Parser is to signify how headers of a packet will be parsed. It can be expressed as a finite state machine as depicted in figure 7. It consists of states that identifies the parsed headers and assign to the parsed header struct the runtime variables that can exclude from packet headers while parsing.

The parser always starts with the start state and ends with either accept or reject state. From the starting and ending state, there are many transition states as defined in the P4 program that programs the Parser. For each intermediate transition the parsed header values are assigned to the respective header struct. These data are extracted using an extern object called packet_in, that represent incoming network packets. Moreover, it has various methods for data extraction and lookahead operations in a packet level manipulation. A P4 parser implementation with transition and select features for various headers is depicted in the code block 3.

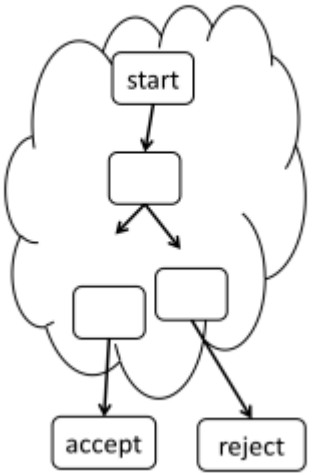


Figure 7 Parser FSM visualization

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

    state start {
        transition select(standard_metadata.ingress_port) {
            CPU_PORT: parse_packetOut;
            default: parse_ethernet;
            }
    }

    state parse_packetOut {
        packet.extract(hdr.pktOut);
        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;
    }
}
```

Code Block 3 A P4 parser implementation

3.4.3 Ingress Processing and Egress Processing

The ingress processing performs packet modifications and decides the output port to forward the packet. After ingress processing has completed, there is another one match-action pipeline that is called egress processing. There can be implemented various modifications in the headers' fields. Moreover, it is important to be noticed that the output port can only be read and not changed at all, in the egress pipeline. The packet's processing, either on ingress or egress, is performed through match-actions operations which are done from blocks called tables and actions respectively. Also, there is an apply block in order to control the order that those match-action tables will be initiated and executed.

3.4.3.1 Table Block

Tables provide the match-action utility upon the parsed headers' struct in order to take an action for a specific behavior, for example defining a multicast group or an output port. For illustration purposes take a look on the code block 4.

```
table mac_table {
        key = {
            hdr.ethernet.dstAddr : exact;
        }
        actions = {
            mac_to_port;
            send_to_cpu;
            broadcast;
        }
        size = 1024;
        default_action = send_to_cpu();
    }
```

Code Block 4 A P4 table

A table with the name mac_table is defined. Each table consist of a key or a set of keys which are the match interface. Furthermore, it consists of an action or a set of actions based on the key's output result which is called a table hit or a miss. These two kind of contents can be manipulated either asynchronously using the control plane to write / read entries to / from the table or statically when the P4 program is initiated and executed in the BMv2

switch. The key attribute has two parts. A field of the parsed headers that recognized by the parser and a match kind field that describe the algorithm that will be used to look up in the table entries to see if there is a match with the parsed header field. This match kind can take 3 values in all architectures which are exact, ternary and lpm (longest prefix method). In v1model we have 3 additional options the range, optional and selector values. Also there is an option to define the size of table entries that a table can have with the size parameter. There is a default action too, that it is performed in case of none of the actions defined in the actions' set are not matched with the corresponding result of the keys' set. This default action should be declared in the actions' set as well. In case it is not, then implicitly the primitive action NoAction acts as a default action by the compiler.

3.4.3.2 Action Blocks

Actions provide a way to read or write data that are being processed. They have the ability to change the behavior of the data plane, using a dynamic approach, with the aid of the control plane which action data of an action are provided through it. An action can be executed either from a table match or by other actions. The code block 5 section illustrates various action blocks.

```
1)    action drop() {
          mark_to_drop(standard_metadata);
      }

2)    action mac_to_port(egressSpec_t port) {
          standard_metadata.egress_spec = port;
      }

3)    action broadcast(McastGrp_t mgrp) {
          standard_metadata.mcast_grp = mgrp;
      }

4)    action send_to_cpu() {
          standard_metadata.egress_spec = CPU_PORT;
      }
```

Code Block 5 Multiple actions implementation

In the first action block, a simple drop action is depicted that uses the primitive action block mark_to_drop which provided from the core library of P4. Also, a common action that is used to P4 programs is the mac_to_port. In this case, the action data (port) are populated from the control plane, either dynamically or statically (when the P4 program is loaded in the BMv2), provide an output port for the packet being processed. A more comprehensive action is the broadcast action, which forwards a packet in all interfaces that have been assigned to a multicast group and defined through the control plane. After population of the action data (mgrp), P4 should assign these data to standard_metadata.mcast_grp field to perform the broadcast action. The last action is placed in order to provide information about communication between the control plane and the data plane that will be introduced in next sections and should define a specific cpu_port for that communication while BMv2 program is running.

3.4.3.3 Apply Block

Apply block section is responsible for invoking tables and their corresponding actions inside the Ingress or Egress control Block. Logic for when and with what order each table is invoked in order to implement the appropriate result. An apply block that implements a mechanism to process a packet if it is received from the control plane is described in code block 6.

```
apply {

        if(standard_metadata.ingress_port == CPU_PORT) {

            temp_port = (bit<9>)hdr.pktOut.egress_port[5:3];
            mcast = (bit<7>)hdr.pktOut.egress_port[0:0];
            if(mac_table_check.apply().miss) {
                if(mcast == 1) {
                    standard_metadata.mcast_grp = (bit<16>) mcast;
                    standard_metadata.ingress_port = temp_port;
                }
            }
            else {
                if(standard_metadata.mcast_grp == 1) {
                    standard_metadata.ingress_port = temp_port;
                }

            }
            hdr.pktOut.setInvalid();
        }
}
```

Code Block 6 Apply block implementation

3.4.4 Deparser

Deparser comes as the last control block of a P4 program and has the responsibility to construct the modified packet in order to be forwarded across the network. To do this, it uses an extern object called packet_out which has an emit function that appends the data to an output packet with a LIFO like formation of the packet headers. Below it is depicted a Deparser of a switch network device that propagates packets across the network.

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

Code Block 7 Deparser implementation

3.5 P4Runtime

The P4Runtime API is a control plane specification for controlling the behavior of the data plane of a device that is defined by a P4 program [**15**]. A combination with the P4 language makes it a target, protocol and pipeline independent API. That's because it works with different switches from different vendors, allows the control of any protocol that could be implemented and of many pipelines that have been specified as well. It uses protocol buffers for serialization and deserialization of the data and gRPC for the implementation of communication channels.

Protocol buffers are an open source project that define a data format, which is language and platform neutral with the ability of serializing or deserializing structured data into .proto files [**16**]. After creation of such files, a protoc compiler can generate code to manipulate messages that are described in a protobuf file in many general purpose programming languages. The generated code consists of simple accessors for each field and methods to serialize and parse the whole structure to and from raw bytes.

gRPC is another open source project which defines a Remote Procedure Call (RPC) framework that can run in any platform. Like many RPC systems, gRPC is based around the

idea of defining a service, signifying the functions that can be called remotely with their parameters and their return types [**17**]. It uses protocol buffers as an Interface Definition Language and message format communication as well.

Figure 8 depicts P4Runtime as an independent API that is written in protobuf file format [**18**]. It consists of a service that defines methods that could be called from a P4Runtime client (or stub) to the P4Runtime server in the form of request-reply actions.

```
service P4Runtime {
  // Update one or more P4 entities on the target.
  rpc Write(WriteRequest) returns (WriteResponse) {
  }
  // Read one or more P4 entities from the target.
  rpc Read(ReadRequest) returns (stream ReadResponse) {
  }

  // Sets the P4 forwarding-pipeline config.
  rpc SetForwardingPipelineConfig(SetForwardingPipelineConfigRequest)
      returns (SetForwardingPipelineConfigResponse) {
  }
  // Gets the current P4 forwarding-pipeline config.
  rpc GetForwardingPipelineConfig(GetForwardingPipelineConfigRequest)
      returns (GetForwardingPipelineConfigResponse) {
  }

  // Represents the bidirectional stream between the controller and the
  // switch (initiated by the controller), and is managed for the following
  // purposes:
  // - connection initiation through client arbitration
  // - indicating switch session liveness: the session is live when switch
  //   sends a positive client arbitration update to the controller, and is
  //   considered dead when either the stream breaks or the switch sends a
  //   negative update for client arbitration
  // - the controller sending/receiving packets to/from the switch
  // - streaming of notifications from the switch
  rpc StreamChannel(stream StreamMessageRequest)
      returns (stream StreamMessageResponse) {
  }

  rpc Capabilities(CapabilitiesRequest) returns (CapabilitiesResponse) {
  }
}
```

Figure 8 P4Runtime service API

As it is stated in the P4 compiler section, during compilation, a p4info file is generated that describes the P4 program in protobuf format. This p4info is used from the controller to control the entities defined to the P4 program. Combined with P4Runtime protobuf file, the communication of the control plane (p4runtime client) with the data plane (p4runtime server) can be controlled. This workflow is depicted in figure 9.

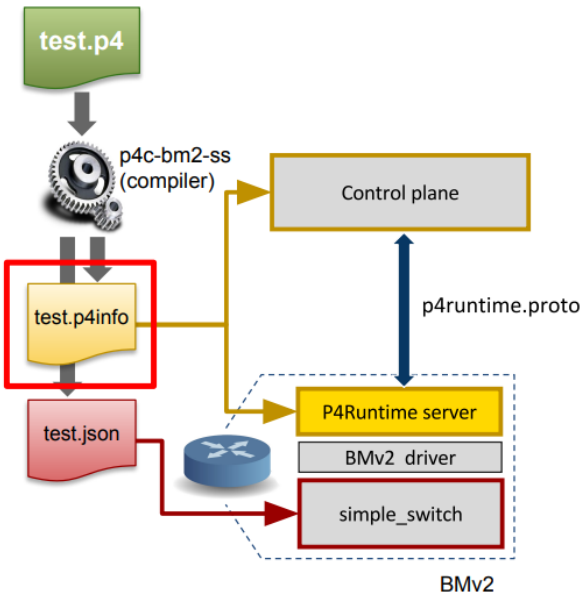


Figure 9 P4Runtime Workflow

It is worth to mention that someone in order to write control plane software it only needs to have access to P4Info file for a P4 program. If that's the case, it can derive the structure of the P4 program from the metadata that are described in this file but a P4 program for that P4info, ideally, should be available. Furthermore, P4Runtime supports the use of multiple controllers, either remote or local, so it can serve as a High availability service (HA) if it has at least one backup controller, which is crucial nowadays.

3.5.1 P4Runtime Hands-on Example

This section refers to a P4Runtime control plane application that has been developed in order to learn and experiment with P4 and P4Rutnime API during this thesis.

First of all, let's introduce the architecture that it was used for this thesis in figure 10. A P4 Target is defined that is controlled by a remote controller which uses the P4Runtime API for connection establishment between the control plane software (p4runtime client) and

data plane (p4runtime server). To be able a controller to send or receive packets to the data plane or from the data plane should configure a control plane port that will be used and data plane should specify two headers for packetIn and packetOut respectively.

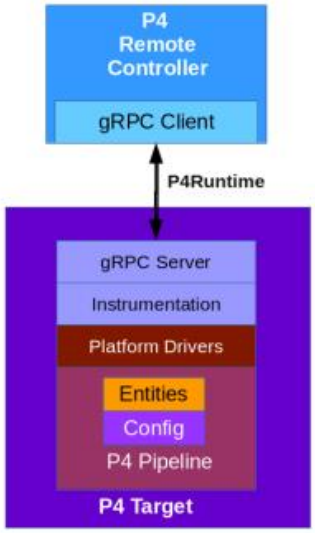


Figure 10 Illustration of the controller use-case.

As part of the P4 program the modifications that should be made to enable the packet I / O operations are displayed in code block 8. Also it needs to be defined a constant variable like CPU_PORT that should have the port that will be used from the device to communicate with the control plane.

```
@controller_header("packet_in")
header PacketIn_t {
  bit<16> ingress_port;
  macAddr_t srcAddr;
}

@controller_header("packet_out")
  header PacketOut_t {
  bit<16> egress_port;
}
```

Code Block 8 packet I / O headers' definition

As a next step, the program that instantiate a p4runtime device should be configured to have the above cpu-port either programmatically in the device's Class as a member field or as a flag in the command that is executed in a terminal or a combination of them, according to the case. After that, it is possible for the control plane to send or receive packets.

Before move into the control plane description, a P4info file for a P4 program that is generated from the compiler is depicted in figure 11. The mapping between the P4 program and the P4info can be concluded clearly.

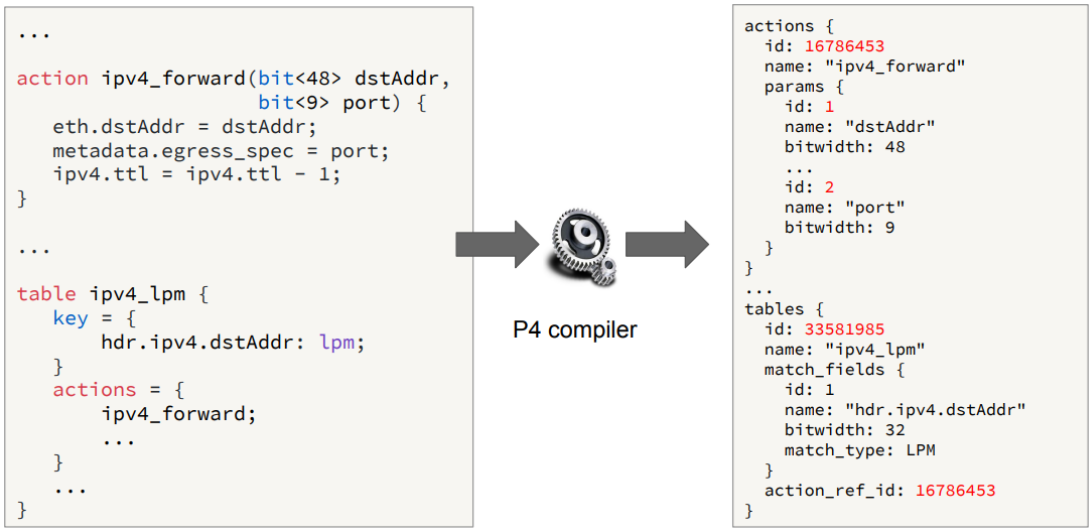


Figure 11 P4info visualization

A controller should make some basic operations on the P4 network devices that will control before it actually starts to control flows. Firstly, it will acquire the P4info from the path that is located and modified it with a helper function in order to be able to consume it. Then, it will open connections to the P4 network devices that want to control using an incremental port and an IP that these devices operate. After that, it will send a MasterArbitrationMessage request. This will make the controller instance to act as a primary controller (master controller) of these network devices before performing any other write operations. Furthermore, it will integrate the P4 program on the switches using the SetForwardingPipelineConfig request. This takes as an argument the device config file, which in case of this thesis is the BMv2 JSON file and the P4info file as well. When these steps are completed, the controller is able to perform write requests to switches using

multithreading programming to control each stream connection separately. The described process is displayed in the below code fragments.

```
    p4info_helper_router = p4runtime_lib.helper.P4InfoHelper(p4info_file_path[0])
    p4info_helper_switch = p4runtime_lib.helper.P4InfoHelper(p4info_file_path[1])

    try:
        s1 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
            name='s1',
            address='127.0.0.1:50051',
            device_id=0,
            proto_dump_file='logs/s1-p4runtime-requests.txt')
        s2 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
            name='s2',
            address='127.0.0.1:50052',
            device_id=1,
            proto_dump_file='logs/s2-p4runtime-requests.txt')
        s3 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
            name='s3',
            address='127.0.0.1:50053',
            device_id=2,
            proto_dump_file='logs/s3-p4runtime-requests.txt')
        s4 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
            name='s4',
            address='127.0.0.1:50054',
            device_id=3,
            proto_dump_file='logs/s4-p4runtime-requests.txt')
```

Code Block 9 Parsing P4info and connection establishment to switches

```
# Send master arbitration update message to establish this controller as
        # master (required by P4Runtime before performing any other write
operation)
        s1.MasterArbitrationUpdate()
        s2.MasterArbitrationUpdate()
        s3.MasterArbitrationUpdate()
        s4.MasterArbitrationUpdate()

        s1.SetForwardingPipelineConfig(p4info=p4info_helper_switch.p4info,
                                       bmv2_json_file_path=bmv2_file_path[1])
        print("Installed P4 Program using SetForwardingPipelineConfig on s1")
        s2.SetForwardingPipelineConfig(p4info=p4info_helper_switch.p4info,
                                       bmv2_json_file_path=bmv2_file_path[1])
        print("Installed P4 Program using SetForwardingPipelineConfig on s2")
        s3.SetForwardingPipelineConfig(p4info=p4info_helper_router.p4info,
                                       bmv2_json_file_path=bmv2_file_path[0])
        print("Installed P4 Program using SetForwardingPipelineConfig on s3")
        s4.SetForwardingPipelineConfig(p4info=p4info_helper_router.p4info,
                                       bmv2_json_file_path=bmv2_file_path[0])
        print("Installed P4 Program using SetForwardingPipelineConfig on s4")
```

Code Block 10 Mastership request and load pipeline to switches

CHAPTER 4 OPEN NETWORK OPERATING SYSTEM

4.1 Overview

Open **N**etwork **O**perating **S**ystem (**ONOS**) is an open source SDN network operating system and controller that provides services like high availability (HA), scale-out and performance that are required by current telecommunication networks [**19**]. It also acts as a foundation of managing and building next-generation networks with the evolution of network programmability and cloud computing. The main features that make ONOS the leading SDN controller, not only in research but also in the industry, are:

- The support of managing the entire network and network components such as switches and links with CLI and GUI configuration options and using software applications as well.
- The ability to install / load / run software applications or modules on top of the ONOS core using well defined APIs in northbound as well as in southbound interfaces that consist of customized communication routing, management, or monitoring services for software-defined networks.
- ONOS platform and applications act as an extensible, modular, distributed SDN controller.

4.2 Design Principles

ONOS is designed as a multi-module project whose modules can be loaded dynamically and managed as OSGi bundles. The ONOS kernel and core services, as well as ONOS applications, are written in Java and can be installed in a single JVM due to OSGi bundles' definitions [**20**]. Its design is based on four principles:

- Code Modularity → It is possible to add new functionalities independently.
- Configurability → It provides static or dynamic loading and unloading of modules using apache Karaf as its OSGi framework.

- Separation of Concern → each subsystem should have distinct boundaries to facilitate modularity. ONOS has been partitioned into:
    - Protocol-aware network-facing modules (southbound API).
    - Protocol-agnostic system core that keeps track the network state.
    - Applications that consume information provided by the core to implement their desired functionality.
- Protocol agnosticism → its core and its applications should not be bounded to any protocol specific implementation. Instead, a new network plugin should be created in southbound API that will provide the desired information to the core without any modifications to other ONOS system components.

4.3 System Components

ONOS architectural design consists of tiers that contain a specific functionality. Figure 12 depicts this tier-level architecture of ONOS with its core in the middle.

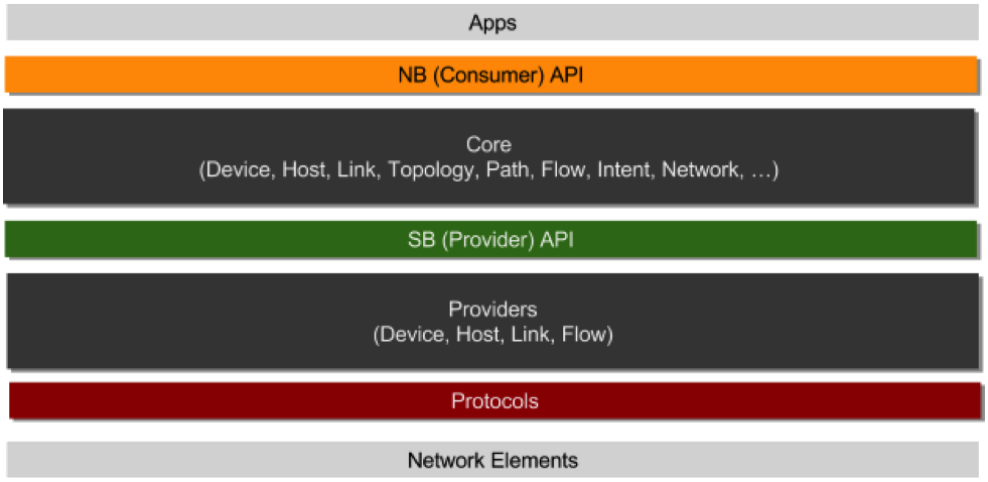


Figure 12 ONOS Tier architecture

A service, in ONOS terms, is a vertical slice of multiple components that offers a piece of functionality from this tier-level architecture. The set of multiple components that make up a service is called a subsystem. The basic primary subsystems of ONOS are:

- Device Subsystem – Management of physical devices.
- Link Subsystem - Management of physical links.
- Host Subsystem - Management of end-station hosts and their positions within the network.
- Topology Subsystem - Management of snapshots of the network's graphical representation taken in chronological order.
- PathService – Calculating / determining routes between network devices or endpoints using the most recent network layout snapshot.
- FlowRule Subsystem - Manages the match / action flow rules installed on physical devices and provides flow analytics.
- Packet Subsystem - Enables applications to monitor data packets received from network devices and transmit data packets out through the network using one or more network devices.

The basic subsystems that are part of ONOS are illustrated in the figure 13.

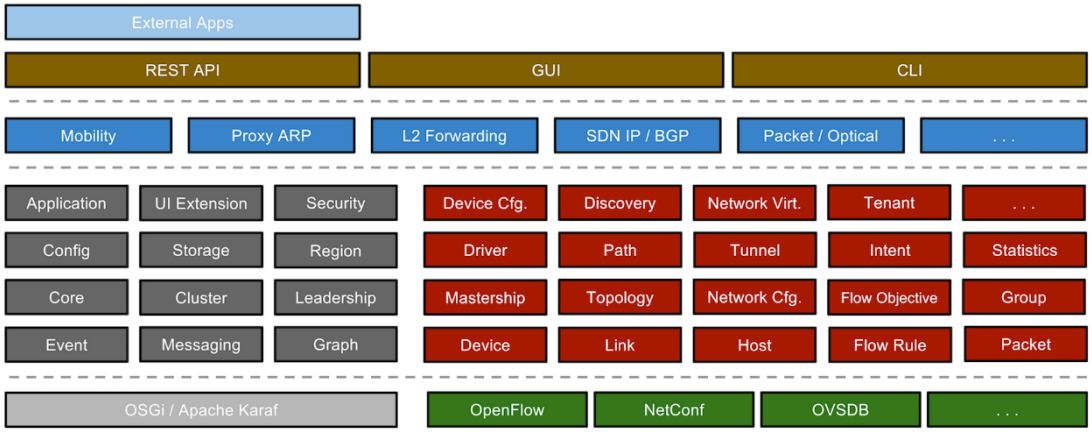


Figure 13 ONOS subsystems overview

It is worth to note, how the information from the infrastructure devices is propagated through ONOS to the applications that run on top of it and vice versa, as well as the structure that provides the communication interfaces of the various components. According to it, figure 14 shows the relationship between the components of a subsystem.

As we move from bottom to top we come across with the Provider Component. Provider Component and its interfaces are responsible for providing not only the abstraction of network infrastructure to the above layers but also the protocol specific communication with the network devices. From Provider's perspective, there is an interface called Provider that interprets every command or information from the core into network specific protocols to talk to the underline network environment and vice versa. Furthermore, each Provider has a providerId that links itself with devices that manages.

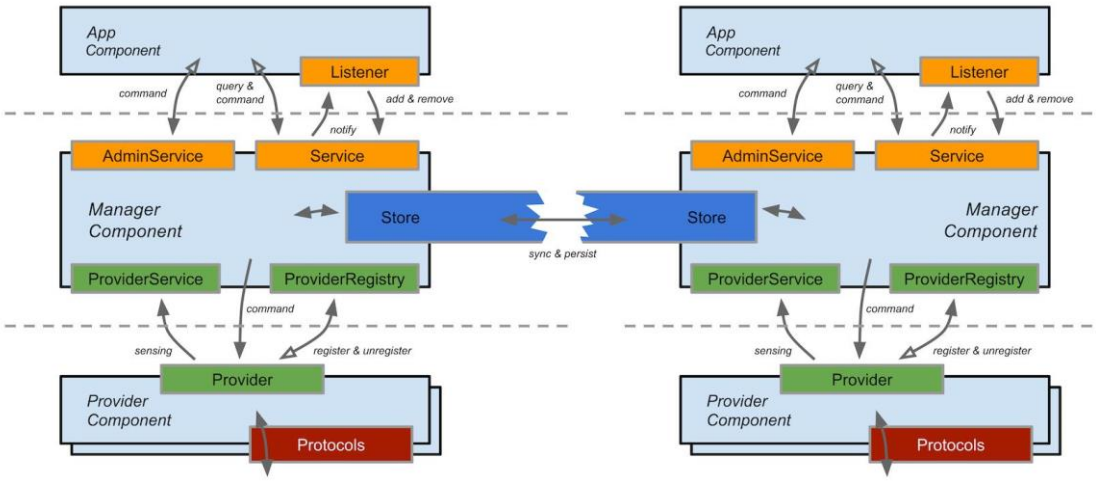


*Figure 14 Relationship between subsystems' components [**21**]*

In the middle of the figure 14, we can see a component called Manager. It resides in the ONOS core and acquires information from Provider and pass it to applications and other services. As for the manager-provider communication, there are two interfaces that co-exist and communicate with the Provider interface. Those are:

- ProviderService
    - This interface is responsible for creating a service through which providers would inject information of network environment to the core.

- ProviderRegistry
    - This interface acts as a common registry where providers can interact with the Manager and offers functionality like registration of a provider to the Manager or deregistration of a provider from the Manager. Also provides retrieval of existing registered providers.

Except from the above interfaces, Manager's component has two additional interfaces that are exposed to the application-manager communication which are: AdminService and Service Interface.

- AdminService
    - It is responsible for administration of the network state or system using administrator commands.
- Service
    - It offers to the ONOS applications or other ONOS core components information about aspects of the network status.

Moreover, applications or consumers of Service interface can either obtain information by requesting it from the service in a synchronous manner or implement an EventListener which will query a specific information of the Manager's service interface if the event that will be monitored gets triggered.

Furthermore, inside the ONOS core and very closely to the Manager component resides the Store. It is responsible for organizing, saving, and keeping updated the information collected by the Manager, guaranteeing its accuracy and stability when there are multiple ONOS instances. This is accomplished through direct communication with the Stores on other ONOS instances.

At the top layer of the figure, we have the applications that consume the information from Manager to implement various functionalities such as displaying network topology in the ONOS GUI or routing the network traffic. Like Provider, applications have unique application ID in order to track application-related context.

In ONOS, the information units consists of events and descriptions. Each of these cannot be changed once created and associated with specific network entities. Descriptions are used to distribute information about entities that comes from southbound API. On the other hand, events are generated through the Store in the Manager component in order to notify Applications' Listeners or other Manager Components in a distributed manner. Once the event is generated, the Manager uses the storeDelegate interface to take the event out of the Store. Then, passes it to the EventDeliveryService that resides in Manager for distributing the event to the interested listeners that implements the EventListener interface either as an internal class of a Manager or an application component. The above description is depicted in figure 15.

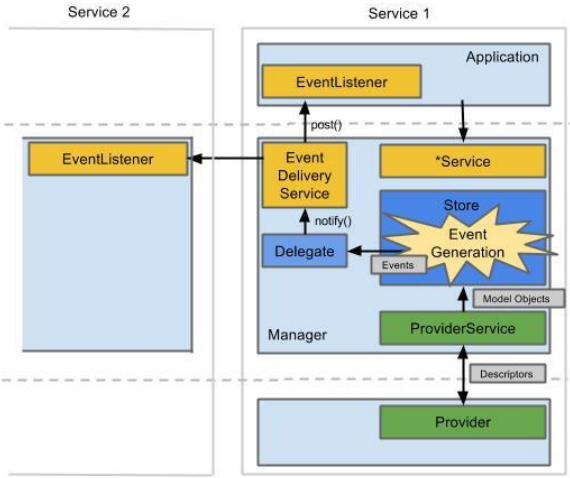


Figure 15 Information Units workflow

4.4 Network State Construction

A very important information that a control plane should keep is the network state. When control plane collects this information can make it available to the applications for further use as a protocol agnostic topology. To make it possible, ONOS uses mechanisms like network discovery and configuration instructed by ONOS itself and applications or operators, respectively.

ONOS has two representations of network elements that controls. One of them is the Model Object representation which is a protocol agnostic construct of network elements used by applications and core components. The other representation is a protocol specific reference of network elements that a provider makes use of it. For example, components of the Device Subsystem, such as DeviceStore, DeviceManager and DeviceListener refers to a network device as Device construct while OpenflowDeviceProvider sees this network element as OpenflowSwitch.

There are various types for representing a Model Object across different functionalities of ONOS. Network Topology, Network Control and Network Packets representations are some of them. Each network infrastructure element is described with a corresponding Model Object name and makes up the Network Topology representation, which is depicted in figure 16.

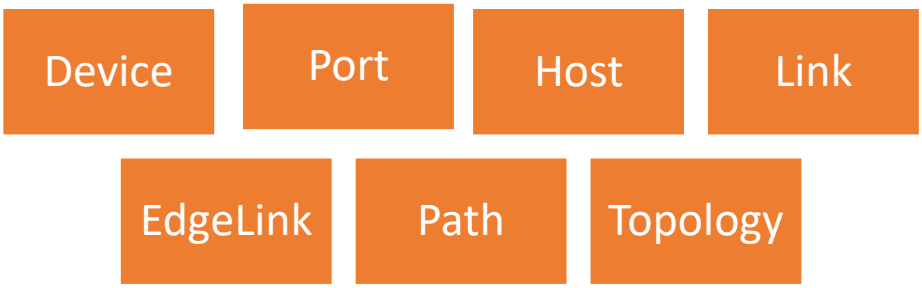


Figure 16 Model Object names of network topology

Furthermore, the network control uses a match-action approach that in ONOS terms is called Criteria-Treatment respectively, regarding applications' space. The network can be controlled either with a high level FlowRule approach that represents match-action pairs or with a FlowObject approach which is used by protocol agnostic applications. Moreover, there is the Intent approach that simply allows applications to specify what they want to happen rather than how to happen and let ONOS do the rest. Lastly, network Packets in ONOS are represented as OutboundPackets when packets will be emitted to the network and InboundPackets when packets are received by ONOS from a network device.

4.5 Device Subsystem

The Device subsystem manages the network devices, including tracking and control of them. This subsystem is crucial for many of the core functions in ONOS and is used by operators and applications to interact with the network. This subsystem or its provider creates and maintains the Device and Port model objects which are meaningful for most ONOS's core subsystems.

The Device subsystem follows a specific architecture and includes:

- The DeviceManager which is able to connect with more than one providers at the same time, through the DeviceProviderService interface and with more than one listeners as well, through the DeviceService interface.

- The DeviceProviders, which supports network protocol libraries or ways to connect to the network related to their Device's specifications.

- The DeviceStore that is able to track Device model objects and produce DeviceEvents.

One meaningful DeviceProvider is the OpenFlowDeviceProvider. ONOS uses it when should communicate with Openflow networks. Before proceeding with a description of the Openflow subsystem and consequently with device subsystem, it is important to recall that the network representation in ONOS. As it is stated before, network representation is visualized differently in the core tier which uses protocol agnostic models and in the provider tier which uses protocol specific models. Having said that, table 1 illustrates the mapping of objects between core and provider tier for the Openflow related network components and properties.

DeviceManager	OpenFlowDeviceProvider
Device	OpenFlowSwitch
DeviceId/ElementId	Dpid
Port	OFPortDesc
MastershipRole	RoleState

Table 1 Object Mapping between Manager tier and Provider tier for Openflow

The OpenFlow southbound includes two components: OpenFlowDeviceProvider and OpenFlow driver. Although these components should not be referred as an ONOS subsystem with its strictly meaning, we will refer to them as the OpenFlow subsystem. This subsystem implements the OpenFlow protocol on the controller side using Java bindings created with Loxi [**22**].

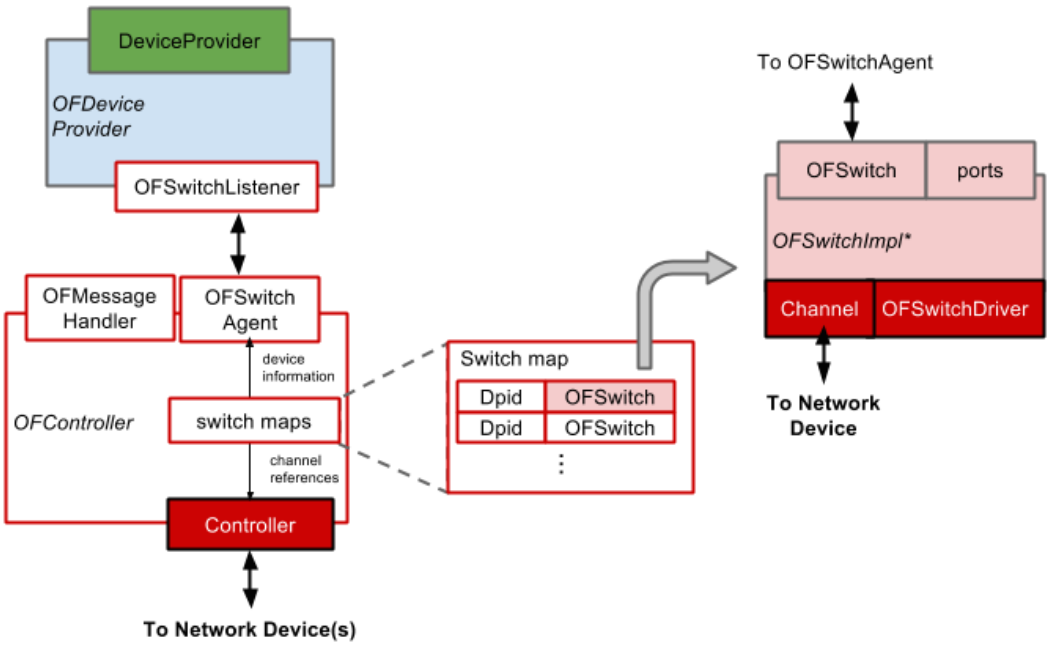


Figure 17 Southbound API of Openflow protocol interfacing with Provider

The blue and green blocks in figure 17 represent the Provider Component and Provider interface respectively. The block that are red and pink in color, as well as those that have

red outlines signify the components that make up the "Protocols" block, which is shown in red in the above figure as well. These are responsible for communication with the physical devices using TCP protocol.

The OpenFlowController coordinates the functions of OpenFlow. It creates a correlation between the DPIDs of switches and the objects they reference in OpenFlowSwitch and produces events that can be accessed by providers as listeners through subscription. These listeners include:

- OpenFlowSwitchListener: listens to switch events such as switch connections and disconnections. Examples include OpenFlowDeviceProvider and OpenFlowLinkProvider.
- OpenFlowEventListener: listens to OpenFlow messages, like OpenFlowRuleProvider.
- PacketListener: listens to packets destined to the controller from the network (PacketIns). Such providers are OpenFlowPacketProvider, OpenFlowLinkProvider, and OpenFlowHostProvider.

The OpenFlowController also manages and establishes communication pathways between each Switch object. It establishes connections and monitors the status of every switch that is connected, through the OpenFlowSwitchAgent. When a connection is made, the Controller creates a Switch object and associates it with a TCP OpenFlow channel (the OFChannelHandler) with a correlation to the TCP connection (labeled Channel in figure 17).

The OpenFlow Switch object symbolizes a network device that is part of the OpenFlow system and comprises ports, a unique identifier, information about the device, and a link to the physical device that is connected through a communication channel. Each Switch object corresponds to a single OpenFlow connection that is coming from the network.

The Switch object has two different kinds of interfaces:

- OpenFlowSwitch: which faces north towards Providers
- OpenFlowSwitchDriver: which faces south towards the channel and Controller

The OpenFlowSwitch enables providers and other components of the ONOS to engage with the Switch object. The OpenFlowSwitchDriver is responsible for managing the specific protocol intricacies that necessitate minimal or no interference from the remainder of the system. These intricacies encompass elements of the OpenFlow handshake that are specific to various types of switches, and the implementation of certain verifications on incoming and outgoing messages.

4.6 Device Driver Subsystem

The main goal of this subsystem is to keep device-specific code apart from other parts of the system. This subsystem provides a method to manage and allow applications to interact with the device-specific code that will be required for an extended period of time, through abstractions that are not dependent on specific devices or protocols. Also, as devices are updated and replaced at different times than network control and management systems, this subsystem enables the dynamic loading of device-specific code asynchronously. ONOS uses a driver mechanism that allows for selective support of features and avoids a monolithic driver approach. This is because different families of devices may have shared and device-only features. The driver mechanism separates various aspects of behavior, enabling features to potentially originate from different sources and to be shared through inheritance within a product line with similar characteristics.

In ONOS, a driver is described as a group of related devices or a single device with:

- a unique name
- support of multiple Behaviors classes
- shared behaviors from other drivers

The Delivery Mechanism of the driver functionality, in ONOS, comes with two interfaces:

- DriverProvider: responsible for providing device drivers and their behaviors
    - Set<Driver> getDrivers()

- DriverAdminService: responsible for tracking and managing device drivers by administrating driver providers through:
    - Set<DriverProvider> getProviders()
    - registerProvider(DriverProvider)
    - unregisterProvider(DriverProvider)

Also, there is a Lookup Mechanism for the device drivers that is based in the DriverService. Apps and other ONOS subsystems can use it in order to find suitable drivers for the device through searching:

- by driver name
- by device manufacturer, H/W version & S/W version
- by supported Behaviour
- by device ID

If there is no connection to a device but still someone wants to talk about a device and its information one can acquire access to DriverData, which contains data learned from previous interactions with a device. As DriverData's functionality is that:

- provides Behaviors for spread information about a device
- has parent Driver

Except from that, if someone wants to interact directly with a device requires a DriverHandler context that:

- provides Behaviors for communicating with a device
- has DriverData
- has parent Driver

4.7 Distributed Operation

ONOS has been created from the beginning as a ground-up distributed SDN operating system. Thus, can be instantiated as a multi-node (multiple servers) system in order to construct a cluster that each node should communicate with the other nodes of the cluster. The existence of more than one ONOS instances provides fault-tolerance and resilience to the system when an individual instance of ONOS fails. Moreover, it provides scalability due to the fact that ONOS as a cluster can handle higher workloads than a single instance deployment could.

A multi-instance ONOS deployment is a group of one or more ONOS instances, also known as nodes, each of which has a unique NodeId. Each node in the cluster is responsible for maintaining and sharing information about a specific part of the network. This information is shared with the other nodes in the cluster through events that are generated in the Store and distributed through it via distributed mechanisms. Except for sharing data across nodes, an ONOS cluster must also manage the addition and removal of nodes and delegate control over devices to ensure that each device has a primary controller. The Cluster subsystem is responsible for managing these tasks.

4.7.1 Distributed Stores

The distribution of the information between nodes as a part of the distribution mechanism that should be used depends upon service's requirements such as strongly consistent or eventually consistent models.

Two different nodes synchronize their subsystems directly through the Store. The Store only synchronizes the state of a subsystem that is a part of. For example, a DeviceStore only knows about the state of devices and does not track host or link information. Figure 18 shows two nodes and a subsystem "A" that is present in both nodes.

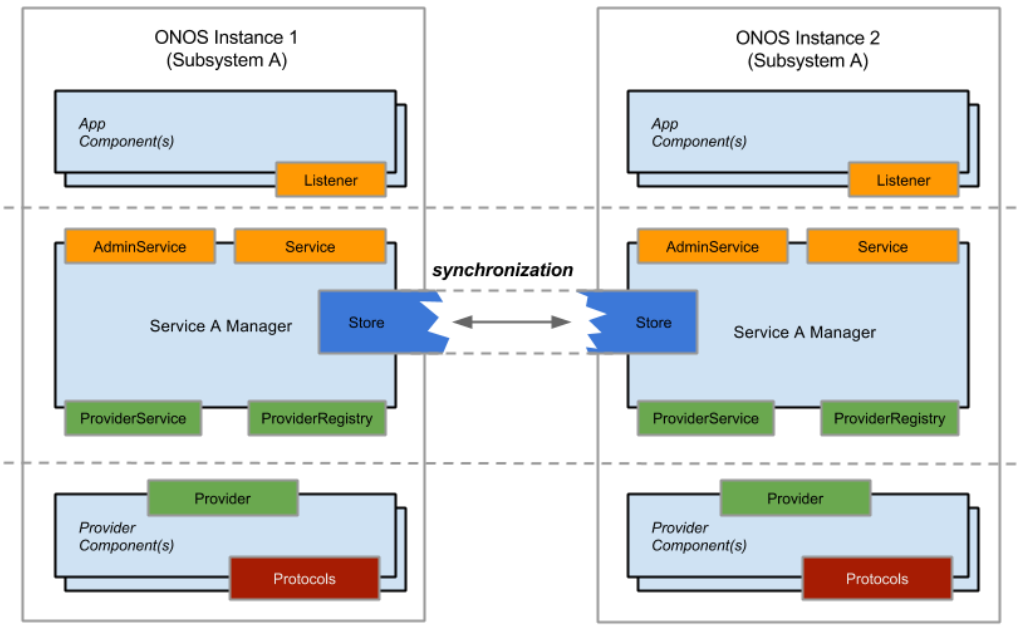


Figure 18 Subsystem's synchronization in ONOS cluster

During the development of ONOS have been made 2 main changes according data coordination architectural concepts. In prior release and specifically in version 1.4 of ONOS, Atomix framework [**23**] is used instead of the existing Hazelcast's distributed structures that were used as a strongly consistent backend. This was embedded inside the ONOS node. But in the version 1.14 of ONOS and after, a new architecture introduced that decoupled cluster management, service discovery and persistent data storage from the ONOS node and moved it to a distinct Atomix cluster. Before this change, the embedded Atomix nodes within ONOS instances were used to create Raft clusters, replicate state and coordinate state changes. In ONOS 1.14 and after, an Atomix cluster should be initiated first and then ONOS nodes should connect to Atomix nodes to form the cluster. This new architecture of ONOS-Atomix is depicted to figure 19.

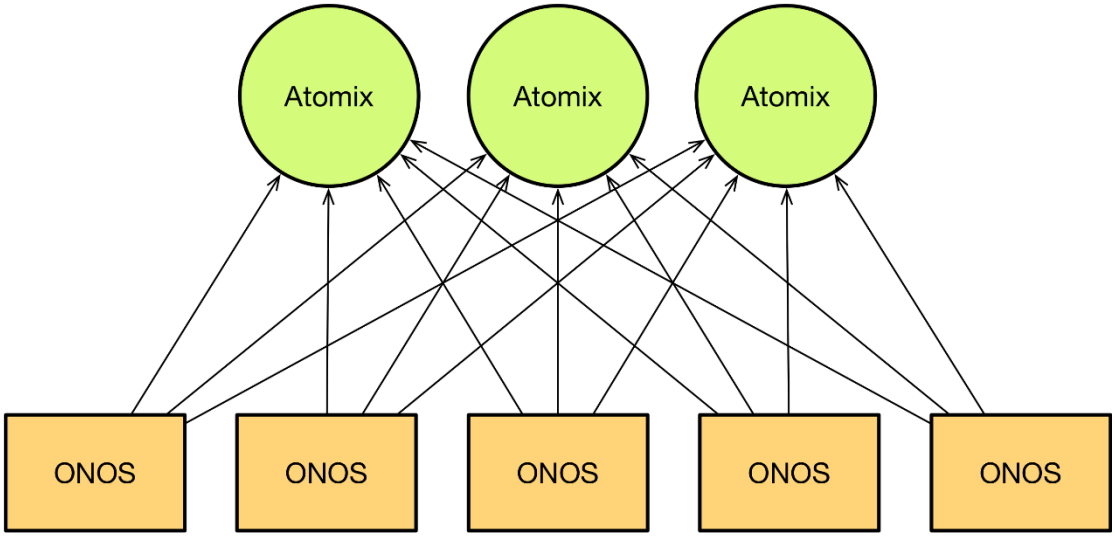


Figure 19 ONOS 1.14 and after cluster architecture

4.8 Intent Framework

The Intent Framework is a subsystem that allows applications to control the network in the form of policy, called Intents, rather than specific mechanisms. The ONOS core processes Intents and translates them into actionable operations on the network environment through a process called Intent compilation. These operations, called installable Intents, are then carried out by the Intent installation process, which can result to changes in network environment such as provisioning tunnel links, installing flow rules on a switch or reserving optical wavelengths. The Intent Framework has been architected to be extensible, which enables the incorporation of additional Intents, as well as their corresponding compilers and installers, to ONOS dynamically during runtime. This feature allows for the expansion of the default set of connectivity and policy-based Intents that are available within ONOS.

An Intent is an object model that is immutable, which represents a request from an application to the ONOS core to modify the network's behavior. It is composed of several elements: Network Resource, Constraints, Criteria, and Instructions. The Network Resource is a collection of object models, such as links, that are impacted by the Intent. Constraints are weights applied to a collection of network resources, such as bandwidth, optical frequency, and link type. Criteria are packet header fields or patterns that illustrate a segment of traffic, and are represented by the Intent's TrafficSelector as a batch of objects

that implement the Criterion interface. Instructions are actions that are applied to a segment of traffic, such as header field modifications or setting egress traffic through specific ports and are represented by the Intent's TrafficTreatment as a collection of objects that implement the Instruction interface. Furthermore, every Intent is identified by a unique IntentId generated at the time of creation and the ApplicationId of the application that submitted it.

4.9 FlowRule Subsystem

The FlowRule subsystem manages the flow rules in the system and installs them on the devices that are present in the network. It employs a distributed authoritative flow table, where the primary copy of the flow rules is retained by the controller and propagated to the devices. This approach ensures that it does not attempt to gather information from the network or integrate flows that are already present on devices. In the event that ONOS identifies a flow on a device that is not in accordance with its authoritative flow table, it will eliminate that flow. The FlowRuleService API is used to add flows into the FlowRule subsystem and they can exist to one of several states: PENDING_ADD, ADDED, PENDING_REMOVE, REMOVED, or FAILED.

The PENDING_ADD status suggests that the FlowRule subsystem has received a request from the application to install a flow rule, yet has not yet detected the flow on the device. The request is then forwarded to the node that holds the master copy of the device in question, which employs the appropriate FlowRuleProvider to install the flow on the device. Once the FlowRule subsystem detects the flow on the device, it moves to the ADDED state. Similarly, the PENDING_REMOVE state signifies that the FlowRule subsystem has received a request from the application to remove the flow, but has not yet received confirmation that the flow has been removed from the device. The FlowRuleProvider is instructed to remove the flow from the device and once confirmation is received, the flow is transitioned to the REMOVED state. In case the device indicates that the flow rule installation has failed, the flow is transitioned to the FAILED state.

CHAPTER 5 P4 AND ONOS

5.1 Pipeline Independent Framework

ONOS was initially designed to operate around Openflow and fixed-function switches [**24**]. As P4 was emerging, ONOS should be extended to support P4 programs and dynamically configured pipelines. Before P4 was able to be supported to ONOS, there were pipeline agnostic apps in ONOS for the Openflow and legacy switches. There should be a way to reuse these apps with P4 programs as well. The solution to it was given by a new horizontal subsystem extension of ONOS core, the Pipeline Independent (**PI**) framework as depicted to figure 20.

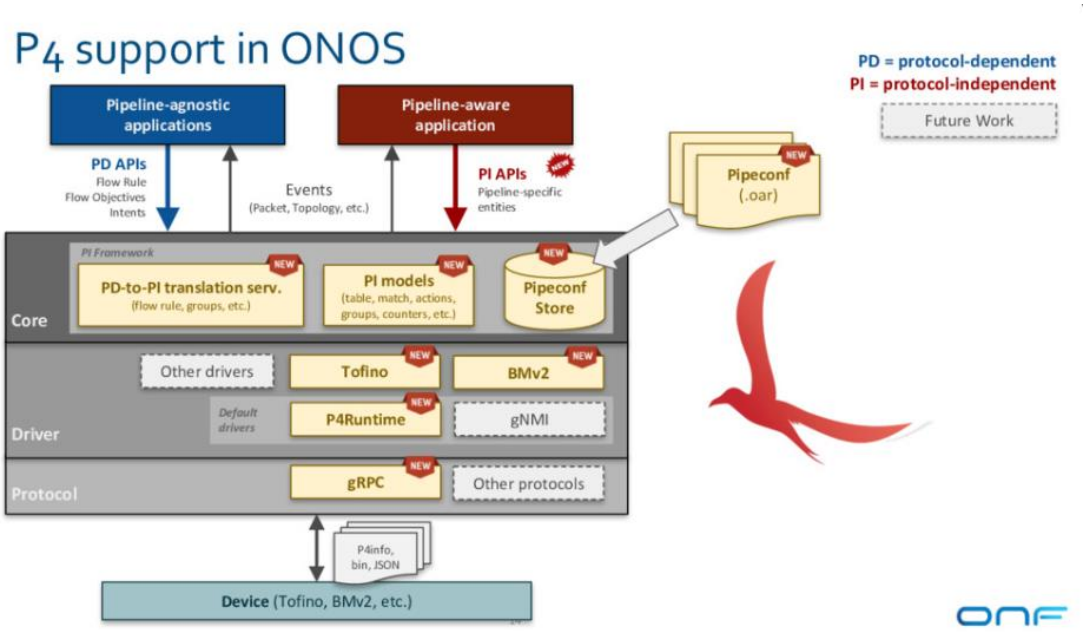


*Figure 20 P4 support in ONOS [**25**]*

PI framework was designed to represent the characteristics that P4 has around protocol, program and pipeline independent facilities [**26**]. This framework enables ONOS to control devices related to P4 and P4Runtime API. It consists of 3 important modules:

- PI module
- FlowRule translation service
- Pipeconf

If ONOS wants to control a P4 device, it needs to write a Pipeconf application.

PI module contains Java classes and interfaces to model and control a programmable data plane. The design of this module is derived from the abstraction that is provided from the protobuf files of P4Info and P4Runtime API. As a result, there is a PI model package on the ONOS core codebase which is based to p4info.proto file and a PI runtime package as well, that is based to p4runtime.proto file. Also, there is a PI service package for services-related functionalities such as Pipeconf management and control. In general, this module models everything around P4 scope.

As for the FlowRule translation service, together with the aid of the Pipeconf service, that is described later, translate pipeline-specific entities from protocol-dependent representations to PI ones. It has the option to validate the translated entities using the P4Info that is loaded from the Pipeconf as illustrated in figure 21.

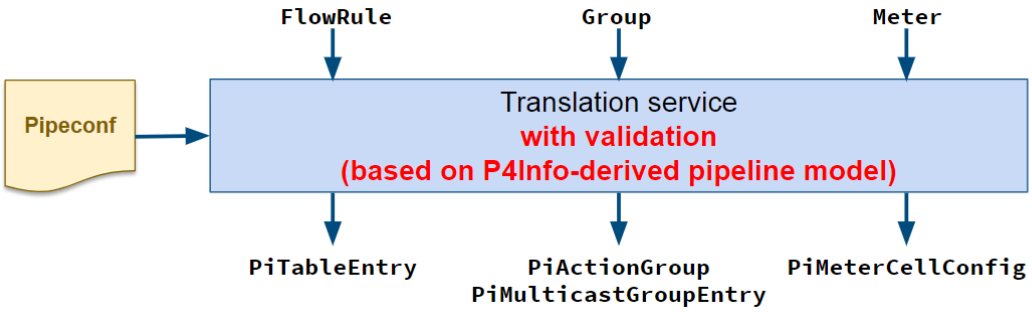


Figure 21 Translation service example

5.2 Pipeconf

Pipeconf is the way that ONOS use, in order to control and manage a P4 device. It is an ONOS app (.oar, ONOS app archive) which includes some required Java files, the P4Info and the BMv2 JSON that are produced after compilation of P4 for BMv2 software switch. It mainly consists of 3 things:

- Pipeline model
    - It is derived automatically from P4Info and describes the pipeline that ONOS should control.

- Target specific binaries to deploy the pipeline to the correspondent device
    - In our case the bmv2 JSON as an output of p4c.
- Pipeline-specific driver behaviors
    - E.g. mapping of ONOS flow programming API to P4 pipeline entities through Pipeliner implementation

5.2.1 PipeconfLoader

These functionalities, which a pipeconf provides, have a one to one mapping with Java classes that implement them. The first thing to do with a pipeconf is to load it. This can be done with *PipeconfLoader.java* that will register the pipeconf through PiPipeconfService. By doing it, the pipeconf will be available for the other subsystems. In the *PipeconfLoader.java* file, it is described everything that a pipeconf have packaged such as p4info, target-specific binaries, added behaviors like Pipeliner or PipelineInterpreter.

```
@Activate
public void activate() {
    // Registers the pipeconf at component activation.
    if (pipeconfService.getPipeconf(PIPECONF_ID).isPresent()) {
        // Remove first if already registered, to support reloading of the
        // pipeconf during the tutorial.
        pipeconfService.unregister(PIPECONF_ID);
    }
    removePipeconfDrivers();
    try {
        pipeconfService.register(buildPipeconf());
    } catch (P4InfoParserException e) {
        log.error("Unable to register " + PIPECONF_ID, e);
    }
}

@Deactivate
public void deactivate() {
    // Do nothing.
}

private PiPipeconf buildPipeconf() throws P4InfoParserException {

    final URL p4InfoUrl = PipeconfLoader.class.getResource(P4INFO_PATH);
    final URL bmv2JsonUrlUrl = PipeconfLoader.class.getResource(BMV2_JSON_PATH);
    final PiPipelineModel pipelineModel = P4InfoParser.parse(p4InfoUrl);

    return DefaultPiPipeconf.builder()
            .withId(PIPECONF_ID)
            .withPipelineModel(pipelineModel)
            .addBehaviour(PiPipelineInterpreter.class, InterpreterImpl.class)
            .addBehaviour(Pipeliner.class, PipelinerImpl.class)
            .addExtension(P4_INFO_TEXT, p4InfoUrl)
            .addExtension(BMV2_JSON, bmv2JsonUrlUrl)
            .build();
}
```

Figure 22 Pipeconf creation and loading ***Error! Reference source not found.***

5.2.2 Pipeline Interpreter

Another component that a pipeconf may contain is the Interpreter. The Interpreter maps ONOS internal data structures to PI structures that represent the PI framework [**28**]. By this way, information is abstracted and can be mapped to P4 program-specific entities. This functionality have many use cases like flow rule operation and packet I/O operations with a P4 device. To be more specific, the Java interface class is called *PiPipelineInterpreter.java* and it provides:

- mapCriterionType: maps ONOS Criterion type to PI match field id
- mapFlowRuleTableId: maps ONOS numeric table id to PI table id
- mapTreatment: maps ONOS treatment to an action on a PI pipeline
- mapOutboundPacket: maps ONOS outbound packet to PI packet operations
- mapInboundPacket: maps PI packet-in operation to ONOS inbound packet
- mapLogicalPort: maps ONOS port number to similar data plane port id

Figures 23 and 24 represents an implementation of Interpreter for mapOutboundPacket alongside with the helper function of buildPacketOut that was examined during this thesis.

```java
@Override
public Collection<PiPacketOperation> mapOutboundPacket(OutboundPacket packet)
        throws PiInterpreterException {
    TrafficTreatment treatment = packet.treatment();

    // Packet-out in main.p4 supports only setting the output port,
    // i.e. we only understand OUTPUT instructions.
    List<OutputInstruction> outInstructions = treatment
            .allInstructions()
            .stream()
            .filter(i -> i.type().equals(OUTPUT))
            .map(i -> (OutputInstruction) i)
            .collect(toList());

    if (treatment.allInstructions().size() != outInstructions.size()) {
        // There are other instructions that are not of type OUTPUT.
        throw new PiInterpreterException("Treatment not supported: " + treatment);
    }

    ImmutableList.Builder<PiPacketOperation> builder = ImmutableList.builder();
    for (OutputInstruction outInst : outInstructions) {
        if (outInst.port().isLogical() && !outInst.port().equals(FLOOD)) {
            throw new PiInterpreterException(format(
                    "Packet-out on logical port '%s' not supported",
                    outInst.port()));
        } else if (outInst.port().equals(FLOOD)) {
            // To emulate flooding, we create a packet-out operation for
            // each switch port.
            final DeviceService deviceService = handler().get(DeviceService.class);
            for (Port port : deviceService.getPorts(packet.sendThrough())) {
                builder.add(buildPacketOut(packet.data(), port.number().toLong()));
            }
        } else {
            // Create only one packet-out for the given OUTPUT instruction.
            builder.add(buildPacketOut(packet.data(), outInst.port().toLong()));
        }
    }
    return builder.build();
}
```

*Figure 23 implementation of the mapOutboundPacket function [**29**]*

```java
private PiPacketOperation buildPacketOut(ByteBuffer pktData, long portNumber)
        throws PiInterpreterException {

    // Make sure port number can fit in v1model port metadata bitwidth.
    final ImmutableByteSequence portBytes;
    try {
        portBytes = copyFrom(portNumber).fit(V1MODEL_PORT_BITWIDTH);
    } catch (ImmutableByteSequence.ByteSequenceTrimException e) {
        throw new PiInterpreterException(format(
                "Port number %d too big, %s", portNumber, e.getMessage()));
    }

    // Create metadata instance for egress port.
    // *** TODO EXERCISE 4: modify metadata names to match P4 program
    // ---- START SOLUTION ----
    final String outPortMetadataName = "egress_port";
    // ---- END SOLUTION ----
    final PiPacketMetadata outPortMetadata = PiPacketMetadata.builder()
            .withId(PiPacketMetadataId.of(outPortMetadataName))
            .withValue(portBytes)
            .build();

    // Build packet out.
    return PiPacketOperation.builder()
            .withType(PACKET_OUT)
            .withData(copyFrom(pktData))
            .withMetadata(outPortMetadata)
            .build();
}
```

Figure 24 BuildPacketOut implementation

It is also possible to not provide an interpreter implementation in the pipeconf. In this case, the code will not be easily maintained and apps would not be compatible with the underlying pipeline. The translation service uses the pipeliner's functionality to implement the translation to PI framework for the flow rules.

5.2.3 Pipeliner

The pipeline-agnostic apps use the FlowObjective service in order to program the network. To support P4, FlowObjective should be transformed to one or many flow rules in order to control the network. Then using the Interpreter can map the flow rules to the abstraction the PI framework provides, for interaction with P4 target-specific entities. The transition from Flow Objectives to flow rules is being made available through the Pipeliner interface. This interface provides:

- init: initialization environment like device's id exposure and FlowRule and Group Service.
- Filter: indicates rules that allow or block packets from entering the Pipeliner
- Forward: describes how packets need to be processed and maps them into flow rule and group
- Next: installs the next hop elements in the device

Figures 25 and 26 represents a Pipiliner interface alongside with an implementation of the Forward method.

```
public interface Pipeliner extends HandlerBehaviour {

    /**
     * Accumulator enabled property. Determines whether the accumulator is enabled.
     * The accumulator is assumed to be disabled if this property is undefined.
     *
     * If enabled, the pipeliner will try to accumulate objectives and create
     * batches of flow rules when possible.
     *
     */
    String ACCUMULATOR_ENABLED = "accumulatorEnabled";

    /**
     * Initializes the driver with context required for its operation.
     *
     * @param deviceId the deviceId
     * @param context  processing context
     */
    void init(DeviceId deviceId, PipelinerContext context);

    /**
     * Installs the filtering rules onto the device.
     *
     * @param filterObjective a filtering objective
     */
    void filter(FilteringObjective filterObjective);

    /**
     * Installs the forwarding rules onto the device.
     *
     * @param forwardObjective a forwarding objective
     */
    void forward(ForwardingObjective forwardObjective);

    /**
     * Installs the next hop elements into the device.
     *
     * @param nextObjective a next objectives
     */
    void next(NextObjective nextObjective);
```

Figure 25 Pipeliner Interface

```
public void forward(ForwardingObjective obj) {
    if (obj.treatment() == null) {
        obj.context().ifPresent(c -> c.onError(obj, ObjectiveError.UNSUPPORTED));
    }

    // Whether this objective specifies an OUTPUT:CONTROLLER instruction.
    final boolean hasCloneToCpuAction = obj.treatment()
            .allInstructions().stream()
            .filter(i -> i.type().equals(OUTPUT))
            .map(i -> (Instructions.OutputInstruction) i)
            .anyMatch(i -> i.port().equals(PortNumber.CONTROLLER));

    if (!hasCloneToCpuAction) {
        // We support only objectives for clone to CPU behaviours (e.g. for
        // host and link discovery)
        obj.context().ifPresent(c -> c.onError(obj, ObjectiveError.UNSUPPORTED));
    }

    // Create an equivalent FlowRule with same selector and clone_to_cpu action.
    final PiAction cloneToCpuAction = PiAction.builder()
            .withId(PiActionId.of(CLONE_TO_CPU))
            .build();

    final FlowRule.Builder ruleBuilder = DefaultFlowRule.builder()
            .forTable(PiTableId.of(ACL_TABLE))
            .forDevice(deviceId)
            .withSelector(obj.selector())
            .fromApp(obj.appId())
            .withPriority(obj.priority())
            .withTreatment(DefaultTrafficTreatment.builder()
                                   .piTableAction(cloneToCpuAction).build());

    if (obj.permanent()) {
        ruleBuilder.makePermanent();
    } else {
        ruleBuilder.makeTemporary(obj.timeout());
    }

    final GroupDescription cloneGroup = Utils.buildCloneGroup(
            obj.appId(),
            deviceId,
            CPU_CLONE_SESSION_ID,
            // Ports where to clone the packet.
            // Just controller in this case.
            Collections.singleton(PortNumber.CONTROLLER));

    switch (obj.op()) {
        case ADD:
            flowRuleService.applyFlowRules(ruleBuilder.build());
            groupService.addGroup(cloneGroup);
            break;
        case REMOVE:
            flowRuleService.removeFlowRules(ruleBuilder.build());
            // Do not remove the clone group as other flow rules might be
            // pointing to it.
            break;
        default:
            log.warn("Unknown operation {}", obj.op());
    }

    obj.context().ifPresent(c -> c.onSuccess(obj));
}
```

Figure 26 Pipeliner Implementation

5.3 Use case scenarios

In order to demonstrate the above concepts, some use case scenarios will be analyzed including device discovery mechanism, packet I/O operations and flow rule operations.

5.3.1 Device Discovery

To a better demonstration of how PI architecture, as an intermediate, interacts with ONOS and P4Runtime API as well as P4 devices, a following device discovery example is used. This example illustrates the initial part of a connection establishment from pipeconf compilation and loading to ONOS to connection establishment and set up pipeline in a P4 device. Figure 27 represents the device discovery process.

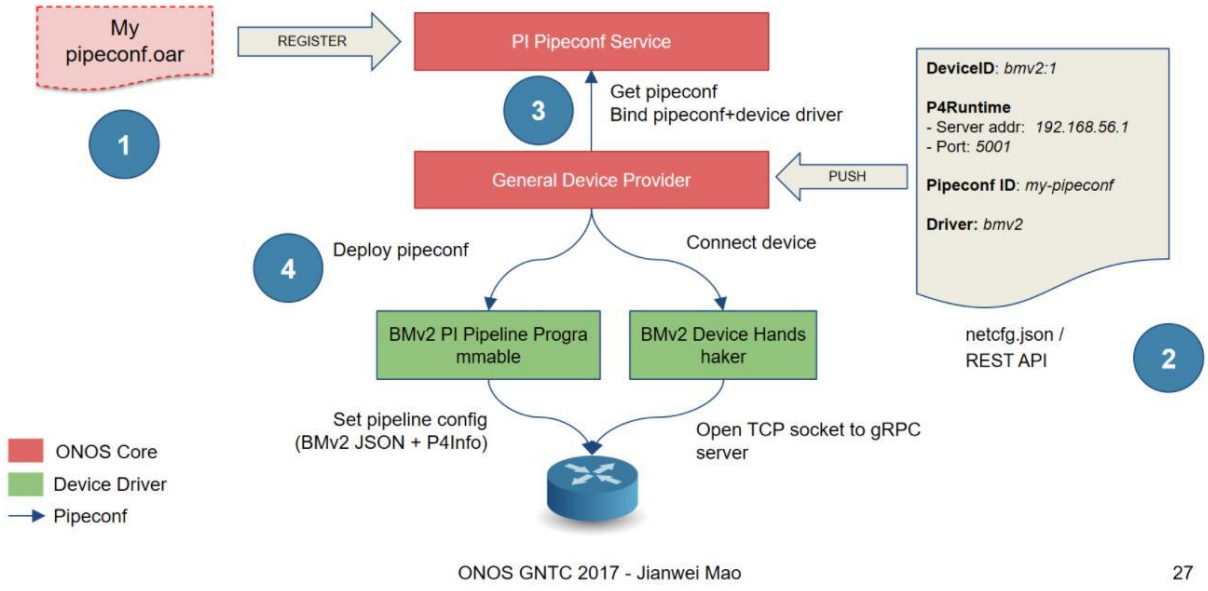


Figure 27 Device discovery process through PI framework

As it can be seen, first step is to compile the pipeconf into the appropriate form (as an ONOS app) and then loaded it to ONOS [**30**]. When pipeconf is loaded, it will register in the PiPipeconfService of the PI framework using the PipeconfLoader.java class. Using the General device Provider, PiPipeconfService will get the pipeconf and bind it with device drivers as well using a device ID that is provided on the pipeconf. Then again, the general device provider will establish a connection with the device using the BMv2 Device

Handshaker and will set the pipeline of the device (using P4Info and BMv2 JSON) as it is described in pipeconf.

5.3.2 Packet I / O

A great use case for interpreter is the packet input / output operations. Apps send or receive packets through PacketRequest or PacketManager service that side in the core tier of ONOS. In case of packet emissions to a P4 device, when a packet is in one of the above services, it will be sent out to the P4Runtime Packet Provider in order to forward to the device Driver of the P4Runtime Packet Programmable. In this step, interpreter logic is used to abstract the internal packet representation to PI framework representation. So after that, the packet is no longer of type OutboundPacket that ONOS internally use, but as PiPacketOperation and then is propagated to the P4Runtime Protocol implementation, which is the P4RuntimeClientImpl.java in the protocols package. In this phase, the PiPacketOperation is transformed, using the P4Info from the loaded pipeconf, to the actual P4 representation entity which is PacketOut and through StreamClientImpl.java is sent it to the corresponding device with the appropriate encoding. Figure 28 illustrates the above description.

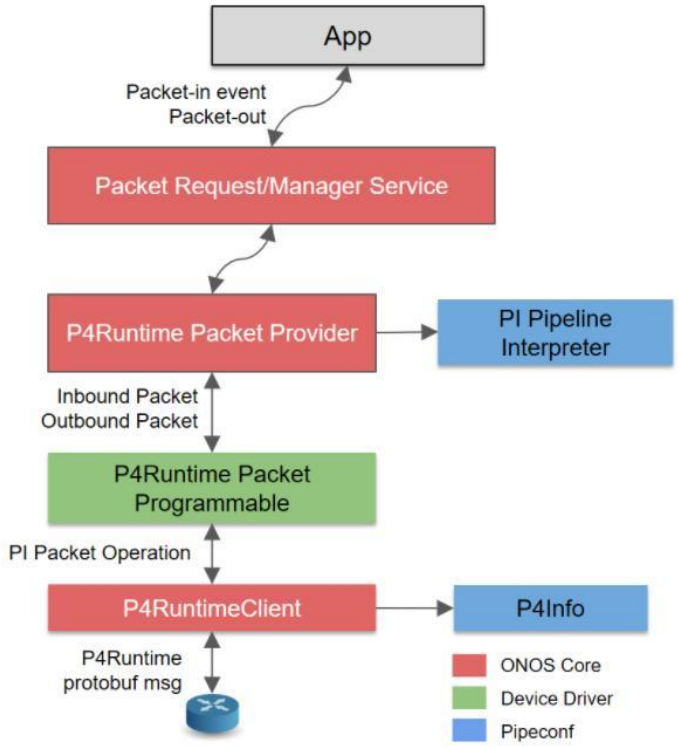


Figure 28 Packet I / O internal information flow

5.3.3 Flow Rule Operations

As a last use case for illustrating pipeconf operations with the other subsystems is flow Rule Operations. At first, there are apps either pipeline-agnostic or pipeline-aware. The former, will use the pipeliner that is provided through the pipeconf in order to translate the Flow Objectives that a pipeline-agnostic app uses to FlowRule. In the case of pipeline-aware apps, the FlowRule service is used directly. After its usage, the *P4RuntimeFlowRuleProgrammable.java* takes up to translate the ONOS internal data structures to PI architecture structures. This is achieved using the PI FlowRuleTranslation service with the aid of interpreter implementation that a pipeconf provides. The entry point for that is through the *P4RuntimeFlowRuleProgrammable.java* that will propagate the abstraction of related entities to the P4Runtime client using the *WriteRequestImpl.java.* Using the P4Info that is provided from the pipeconf, the encoding to valid protobuf messages will be done and the request will be sent to the specific P4 device. Figure 29 visualizes the above description.

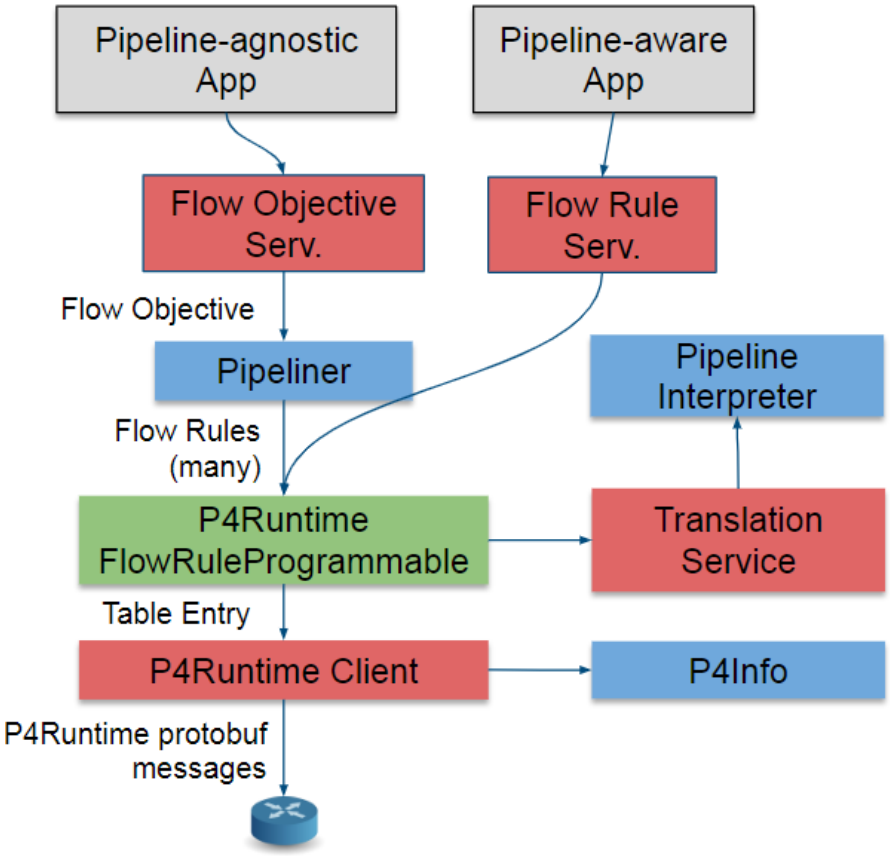


Figure 29 Flow Rule operations path

CHAPTER 6 CONCLUSION AND FUTURE WORK

In this thesis, the SDN solution was presented which seems to gain a great part of today's business networks. To this way, the Openflow protocol was presented to a high level and new concepts concerning P4 programming language as well as P4Runtime API described thoroughly. Furthermore, ONOS controller was pointed out which plays a key role to industry as a good option for production-grade controllers to help the transition to the SDN world. To this part, the P4 compatibility in ONOS was mentioned in order to better understanding the steps that should be made to talk to P4 switches from the ONOS perspective. Moreover, this thesis aimed to provide more documented information, either in P4 or ONOS, to make it possible to future researchers find related information on these topics.

Future plans include investigation on other ONF projects that include both technologies and steering innovation like SD-RAN and SD-Fabric for 5G related topics. At last, a study and contribution to micro-onos can be made having as a base the monolithic ONOS implementation that this thesis is based, while trying to make it a cloud-native SDN controller.

REFERENCES

[1] P4.org. (2016, May 18). Clarifying the differences between P4 and OpenFlow. Retrieved January 14, 2023, from Open Networking Foundation website: https://opennetworking.org/news-and-events/blog/clarifying-the-differences-between-p4-and-openflow/

[2] Feamster, N., Rexford, J., & Zegura, E. (2014). The road to SDN: An intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, *44*(2), 87–98. https://doi.org/10.1145/2602204.2602219

[3] Sloane, T. (2013, May 2). Software-defined networking: The new norm for networks. Retrieved January 15, 2023, from Open Networking Foundation website: https://opennetworking.org/sdn-resources/whitepapers/software-defined-networking-the-new-norm-for-networks/

[4] Ali, J., Lee, G., Roh, B., Ryu, D. K., & Park, G. (2020). Software-defined networking approaches for link failure recovery: A survey. *Sustainability*, *12*(10), 4255. https://doi.org/10.3390/su12104255

[5] ONF. (2012). *OpenFlow Switch Specification*. ONF. Retrieved from https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf

[6] Jammal, M., Singh, T., Shami, A., Asal, R., & Li, Y. (2014). Software defined networking: State of the art and research challenges. *Computer Networks*, *72*, 74–98. https://doi.org/10.1016/j.comnet.2014.07.004

[7] Abigail O. Jefia, Segun I. Popoola, & Aderemi A. Atayero. (2018, September 27). *Software-Defined Networking: Current Trends, Challenges, and Future Directions*. Presented at the Industrial Engineering and Operations Management, Washington DC, USA. Washington DC, USA.

[8] P4~16~ language specification. Retrieved January 15, 2023, from https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html

[9] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Walker, D. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, *44*(3), 87–95. https://doi.org/10.1145/2656877.2656890

[10] p4lang /p4c: P4_16 reference compiler. Retrieved from https://github.com/p4lang/p4c

[11] P4 – language consortium. Retrieved January 15, 2023, from https://p4.org/

[12] p4lang / behavioral-model: The reference P4 software switch. Retrieved from https://github.com/p4lang/behavioral-model

[13] The BMv2 Simple Switch target. Retrieved from https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md

[14] Kaur, S., Kumar, K., & Aggarwal, N. (2021). A review on P4-Programmable data planes: Architecture, research efforts, and future directions. *Computer Communications*, *170*, 109–129. https://doi.org/10.1016/j.comcom.2021.01.027

[15] P4runtime specification. Retrieved January 15, 2023, from https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html

[16] Overview | protocol buffers. Retrieved January 15, 2023, from Google Developers website: https://developers.google.com/protocol-buffers/docs/overview

[17] Core concepts, architecture and lifecycle. Retrieved January 15, 2023, from GRPC website: https://grpc.io/docs/what-is-grpc/core-concepts/

[18] p4lang / p4runtime: P4runtime.proto. Retrieved from https://github.com/p4lang/p4runtime/blob/main/proto/p4/v1/p4runtime.proto

[19] *Introducing ONOS - a SDN network operating system for Service Providers*. Retrieved from https://stordirect.com/wp-content/uploads/woocommerce-products-data/product_documents/Open-Network-Operating-System-ONOS_Whitepaper.pdf

[20] Open network operating system (Onos) sdn controller for sdn/nfv solutions. Retrieved January 15, 2023, from Open Networking Foundation website: https://opennetworking.org/onos/

[21] Architecture and internals guide—Onos—Wiki. Retrieved January 15, 2023, from https://wiki.onosproject.org/display/ONOS/Architecture+and+Internals+Guide

[22] OpenFlowJ Loxi · floodlight/loxigen Wiki. Retrieved January 15, 2023, from GitHub website: https://github.com/floodlight/loxigen

[23] Atomix. Retrieved January 15, 2023, from https://atomix.io/

[24] Security & performance analysis brigade—ONOS - Wiki. Retrieved January 15, 2023, from https://wiki.onosproject.org/pages/viewpage.action?pageId=12422167&preview=/12422167/72253443/ONOS%2BP4%20SecPerf%20Workshop%20%40%20TMA%202019.pdf

[25] Onos support for p4 runtime. Retrieved January 15, 2023, from Speaker Deck website: https://speakerdeck.com/pichuang/onos-support-for-p4-runtime?slide=14

[26] Yi Tseng. (2017, September). *20170925 onos and p4*. Retrieved from https://www.slideshare.net/YiTseng/20170925-onos-and-p4

[27] Ngsdn-tutorial/app/src/main/java/org/onosproject/ngsdn/tutorial/pipeconf/PipeconfLoader.java. Retrieved from https://github.com/opennetworkinglab/ngsdn-tutorial/blob/advanced/app/src/main/java/org/onosproject/ngsdn/tutorial/pipeconf/PipeconfLoader.java

[28] Manzanares-Lopez, P., Muñoz-Gea, J. P., & Malgosa-Sanahuja, J. (2021). P4-kbr: A key-based routing system for p4-programmable networks. *Electronics*, *10*(13), 1543. https://doi.org/10.3390/electronics10131543

[29] Ngsdn-tutorial /solution/app/src/main/java/org/onosproject/ngsdn/tutorial/pipeconf/InterpreterImpl.java/. Retrieved from https://github.com/opennetworkinglab/ngsdn-tutorial/blob/advanced/solution/app/src/main/java/org/onosproject/ngsdn/tutorial/pipeconf/InterpreterImpl.java

[30] Yang, p4 runtime in the onos architecture—Programmer sought. Retrieved January 15, 2023, from https://programmersought.com/article/61418255779/