P4Debugger: Tracing through Network Changes with Table-Version Packet Tainting

Chatzivasileiou Antonios

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Xenofontas Dimitropoulos*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

This research has been financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EDK-03389).

University of Crete
Computer Science Department

P4debugger: Tracing through Network Changes with Table-Version Packet Tainting

Thesis submitted by
Chatzivasiliou Antonios
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: [illegible]
Chatzivasiliou Antonios

Committee approvals: [illegible]
Xenofontas Dimitropoulos
Associate Professor, Thesis Supervisor

[illegible]
Kostas Magoutis
Associate Professor, Committee Member

[illegible]
Christos Liaskos
Assistant Professor, Committee Member

Departmental approval: [illegible]
Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, October 2021

P4Debugger: Tracing through Network Changes with Table-Version Packet Tainting

Abstract

The increasing consumer demands for network performance and flexibility have led enterprises to expand in Software Defined Networking (SDN). SDN is a technology that changes the way that networks work by separating the network's control logic from the underlying switches and routers, promoting centralization of network control, and introducing the ability to program the network. This allows Network administrators to use network devices and supervise them from a programmable controller. However, configuring these devices in combination with the significant growth of protocol headers increases the complexity and makes them prone to bugs. P4 is a language that works in conjunction with SDN, which expresses how packets are forwarded by the programmable network devices and have the ability to add custom headers to packets.

In this thesis, we introduce P4debugger, a prototype network debugger for SDN developers which exploits the abilities of the P4. Our debugger is divided into two parts; The first part is that we taint each packet that passes through the switch with some information that allows us to backtrace them, detect loops, as well as inspect them for any policy violation. For inspecting the extra information, we use monitors that sample and analyze the packets that pass through each switch. Network administrators can preview from these monitors the behavior of the network based on the fields in the custom header of the packets. The second part of our implementation is that our controllers save the state of the flow tables before they make any changes to them. We implemented a web app to preview a visual representation of the network topology, and based on the flows the switch had at a specified time, the web app simulates its behavior. By employing this web app, the user is capable of observing the reachability as well as problems such as network loops.

We evaluate our thesis by presenting three errors commonly seen by SDN programmers, which provide a solid example of how P4debugger helps a programmer find the source of the problem. Finally, we calculated the overhead that P4debugger applies to the network topology, concluding that we contribute a valuable tool to Network administrators.

P4debugger: Παρακολούθηση αλλαγών δικτύου χρησιμοποιώντας προσαρμοσμένες κεφαλίδες και καταγραφή πινάκων

Περίληψη

Οι αυξανόμενες απαιτήσεις των καταναλωτών για αποδοτικό και ευέλικτο δίκτυο έχουν οδηγήσει τις επιχειρήσεις να επεκταθούν στο Software Defined Networking (SDN). Το SDN είναι μια τεχνολογία που αλλάζει τον τρόπο λειτουργίας των δικτύων διαχωρίζοντας τη λογική ελέγχου του δικτύου από τα υποκείμενα switches και routers, προωθώντας τον συγκεντρωτισμό του ελέγχου του δικτύου και εισάγοντας τη δυνατότητα προγραμματισμού του. Αυτό επιτρέπει στους διαχειριστές δικτύου να χρησιμοποιούν συσκευές δικτύου και να τις εποπτεύουν από έναν προγραμματιζόμενο ελεγκτή. Ωστόσο, η διαμόρφωση αυτών των προγραμματιζόμενων συσκευών σε συνδυασμό με τη σημαντική αύξηση των κεφαλίδων πρωτοκόλλου, αυξάνει την πολυπλοκότητα και τις καθιστά επιρρεπείς σε σφάλματα. Το P4 είναι μια γλώσσα που λειτουργεί σε συνδυασμό με το SDN, η οποία εκφράζει τον τρόπο με τον οποίο τα πακέτα προωθούνται από τις προγραμματιζόμενες συσκευές δικτύου και έχουν τη δυνατότητα προσθήκης προσαρμοσμένων κεφαλίδων σε πακέτα.

Σε αυτήν την εργασία, εισάγουμε το P4debugger, ένα πρωτότυπο πρόγραμμα εντοπισμού σφαλμάτων δικτύου για προγραμματιστές SDN το οποίο εκμεταλλεύεται τις ικανότητες του P4. Το πρόγραμμα εντοπισμού σφαλμάτων χωρίζεται σε δύο μέρη. Το πρώτο μέρος είναι ότι χρησιμοποιούμε μια προσαρμοσμένη κεφαλίδα για να εισάγουμε πληροφορία σε κάθε πακέτο που περνάει από το switch με αποτέλεσμα να μπορούμε να κάνουμε ιχνηλάτηση του πακέτου, να εντοπίσουμε τυχόν βρόχους, καθώς και να ελέγξουμε για κάποια παραβίαση της πολιτικής του δικτύου. Για τον έλεγχο των επιπλέον πληροφοριών, χρησιμοποιούμε παρατηρητές που αναλύουν δειγματοληπτικά τα πακέτα που περνούν από κάθε switch. Οι διαχειριστές δικτύου χρησιμοποιώντας τους παρατηρητές, κατανοούν τη συμπεριφορά του δικτύου με βάση τα πεδία στην προσαρμοσμένη κεφαλίδα των πακέτων. Το δεύτερο μέρος της εφαρμογής μας είναι ότι οι ελεγκτές μας αποθηκεύουν τους πίνακες ροής πριν κάνουν οποιεσδήποτε αλλαγές σε αυτούς. Κατασκευάσαμε μια εφαρμογή ιστού για προεπισκόπηση μιας οπτικής αναπαράστασης της τοπολογίας του δικτύου και με βάση τις ροές που είχε το switch σε μια συγκεκριμένη χρονική στιγμή, η εφαρμογή ιστού προσομοιώνει τη συμπεριφορά της. Χρησιμοποιώντας αυτήν την εφαρμογή ιστού, ο χρήστης είναι σε θέση να παρατηρήσει την δυνατότητα προσέγγισης του δικτύου καθώς και προβλήματα όπως οι βρόχοι.

Αξιολογούμε τη διατριβή μας παρουσιάζοντας τρία συνήθη λάθη που παρατηρούν οι προγραμματιστές SDN, τα οποία παρέχουν ένα σταθερό παράδειγμα για το πώς το P4debugger βοηθά έναν προγραμματιστή να βρει την πηγή του προβλήματος. Τέλος, υπολογίσαμε τις επιπτώσεις που έχει ο P4debugger όσον αφορά την απόδοση του δικτύου, καταλήγοντας ότι συνεισφέρουμε ένα πολύτιμο εργαλείο στους διαχειριστές δικτύου.

Ευχαριστίες

Θα ήθελα πρωτίστως να ευχαριστήσω τον επόπτη καθηγητή μου κύριο Ξενοφώντα Δημητρόπουλο, για τη στήριξη και την εμπιστοσύνη που μου έδειξε καθ' όλη τη διάρκεια των μεταπτυχιακών σπουδών μου. Επίσης θα ήθελα να ευχαριστήσω τον Δρ. Βασίλειο Κοτρώνη για τις συμβουλές, τις ιδέες και τον χρόνο που αφιέρωσε.

Θα ήθελα να εκφράσω τις ευχαριστίες μου στον Καθηγητή κ. Κώστα Μαγκούτη και στον Καθηγητή κ. Χρήστο Λιάσκο για τη μεγάλη προθυμία να συμμετέχουν στην τριμελή επιτροπή.

Ακόμα, θα ήθελα να ευχαριστήσω το Ινστιτούτο Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας για την υποτροφία που μου προσέφερε καθ' όλη τη διάρκεια της μεταπτυχιακής μου εργασίας.

Πολλές ευχαριστίες θα ήθελα να δώσω στα μέλη της ομάδας του Inspire Group Γιώργο Νομικό, Μάνο Λακιωτάκη, Βασίλη Πετρόπουλο και Γιώργο Επταμηνιτάκη. Επίσης, θα ήθελα να ευχαριστήσω τους φίλους μου, Κώστα Φετφατζή Σωτήρη Τοτόμη, Κώστα Αρακαδάκη και Ιάσωνα Μαστοράκη για την στήριξη και τον παραγωγικό χρόνο που περάσαμε μαζί.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου και ιδιαίτερα τους γονείς μου Παντελή και Χρυσούλα, καθώς και τα αδέρφια μου Στέλλα, Μαριαλένα, Νικόλα, Ορέστη, Νεφέλη και τη σύντροφό μου Ανδριάνη για την υποστήριξη και την αγάπη που με περιέβαλλαν.

Στην οικογένειά μου

Table of contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software Defined Networking (SDN) is an emerging paradigm that decouples the control plane from the data plane to simplify network management and enable innovations in networking [28]. This split architecture indicates the control plane to make all the control logic decisions and the data plane to execute these forwarding decisions. SDN can implement networks that can rapidly change the demands for network resources or quickly evolve to match users' demands. Some technologies that match with the above statement are Network Function Virtualization (NFV) [30], Internet of Things (IoT) [17] and cloud computing [26].

SDN has undergone continual development in the past years and has been a research subject from academia and industry people. Many IT corporations have adopted SDN technology to deploy their Networks. More specifically, Google launched B4 project [32], Facebook launched edge fabric [38] and Microsoft published Ananta [36]. In addition, SDN is recognized as the critical technology that enables the development of many other network technologies such as 5G, IoT, and NFV. Therefore, SDN constitutes the best choice for modern network management. Nevertheless, despite the significant attention and adoption of SDN, most network operators have concerns about the reliability of this technology [42].

These concerns are because SDN simplifies network management by providing a centralized API (SDN controller) where network management programs can be written. The SDN controller must manage policy configurations, host migrations, react to failures, and many other events. The advantage of a software-based management plane is that we can dynamically counter all the above events and make our network as adaptive and flexible as it can be. However, this creates a highly complex system that is prone to bugs [31] [39].

When we have a network problem (e.g., a loop or blackholing), it suggests that we have a bug in our control logic (controller), and network administrators need to trace the source of this bug and fix it as fast as possible. Nevertheless, this debugging procedure is highly time-consuming and demands lots of resources as developers may spend many hours trying to find the bug inspecting various forwarding tables, and creating custom traffic in order to understand where packets escape the original flow [37].

A single misconfiguration or a typographical error can cause many problems, from reachability issues to loops or misleading packets, which lead to limitation of the providing services and may cost millions. Those problems can appear as soon as we program the control plane, but there is a possibility that we trigger these wrong configurations when our network reaches a different state (e.g., when a switch shuts down). So as long as the specific switch is down, we may have loops or blackholing problems. However, when the switch is up again, the network returns to a healthy state. When network administrators build a dynamic network configuration, it is impossible to check all these cases.

P4 (Programming Protocol-Independent Packet Processors) is a domain-specific programming language that takes the concept of SDN one step further by making the forwarding devices fully programmable. P4 is used to define the behavior of the data plane in P4 programmable devices. The combination of SDN and P4 gives network administrators the capability for a dynamic and flexible, fully programmable network.

Furthermore, P4 forwarding devices allow the programmer to implement custom protocols without any dependency on the switch vendor. P4 language is able to only include the protocols that the device needs for forwarding the packet, which significantly reduces the complexity and enables easier verification. Finally, hardware P4 switches can reach up to 12.8 Tb/s with up to 400GbE port like Tofino 2 switch [5].

Our goal is to help network administrators to find potential errors in their topology fast and with ease. This way, they will not hesitate to try more venturesome approaches on their network, which may lead them to develop it further. In this thesis, we present P4Debugger, a prototype debugger for SDN networks, which exploits the abilities of the P4 language to be a reliable and efficient tool. P4Debugger provides a web application where network administrators can load past or current time flow tables and simulate the forwarding state of their network. This enables them to research previous states of the network and compare the differences in its behavior. Moreover, we taint each packet that passes through the switch with some information that allows us to backtrace them, detect loops, as well as inspect them for any policy violation.

1.2 Outline of the Thesis

The thesis is structured as follows. In Chapter 2, we introduce all the relevant theoretical background related to SDN, P4 language, as well as all the development platforms that we used in this thesis. In Chapter 3, we present the related work we found during our literature review, and we annotate the most important elements of each work. Chapter 4 is where we analyze the Implementation of P4Debugger, we divide this chapter into individual pieces, and the most important of them is the Data plane, which describes the functionality of our P4 program, after that is the Control Plane where we show how we manage the P4 switches, and finally the Web Application where we describe how we parse and visualize our data. In Chapter 5, we present some use case scenarios and describe how our Debugger deals with the individual problems. In Chapter 6, we present the results from the performance evaluation, and finally, in Chapter 7, we present our conclusion and possible future work.

Chapter 2

Theoretical Background

2.1 Traditional Switches

The basic role of a Link-Layer switch is to receive incoming frames and forward them onto outgoing links.Although switches are **transparent** for hosts and routers, they play the foremost role in their interconnection [29]. The main operations that a switch perform are filtering, forwarding, self-learning, and prevention of loops.

2.1.1 Media access control address

The link-layer address is variously called as such physical address or a LAN address, but the most popular one is the Media access control (MAC) address. Hosts and routers do not have a MAC address, but rather their network interfaces are the ones that do. A host or a router may have multiple interfaces, which corresponds to multiple MAC addresses. The link-layer switches do not have MAC addresses on their interfaces; they do their job transparently, which means that the router or host does not have to address a specific frame to the intermediate switch.

The MAC address, which tries to reach all hosts and routers in a network, except the interface where it comes from, is named **broadcast address** and is represented as ff:ff:ff:ff:ff:ff. The second category of special MAC addresses is multicast addresses. These addresses belong to multiple hosts and are identified based on the least significant bit of the destination MAC being set to 1.

2.1.2 Filtering and Forwarding

Filtering is the operation that the switch does in order to determine if the frame should be forwarded to some interface or just dropped. **Forwarding** is the operation of the switch to determine the interfaces to which a frame should be forwarded. As a means to perform these operations, the switch is using a **switch table**. This table contains information about the *MAC* address of the associated host, the *switch interface*, and the *time* at which the entry is placed in the table. A

MAC address is unique and can matches with only one interface, except for **multicast** and **broadcast** addresses that associate with multiple interfaces. The switch checks the destination MAC address at the header of the frame and performs a lookup at the switch table to find the matched interface. If a matched interface is different from the interface that the packet came from, then the switch forwards the packet to the specific interface. If there is no match for that MAC address, the switch forwards copies of the frame to all interfaces, except the interface that the packet came from, so-called broadcasting. Finally, the filtering method is applied when the matched entry has the same interface that the packet came from. In this case, there is no need to forward the frame to any interface. The switch filters the frame by discarding it.

2.1.3 Learning

Especially in large topologies, it is not easy to populate the switch tables manually. Thankfully the switch has a property that can build its table automatically. This **self-learning capability** is accomplished by saving for every "new" incoming frame its source MAC address, the interface from which the frame arrived, and the time that the event happened. Since the network topology can change and the switch table size is finite, switches must keep the most recent hosts in their tables. Keep most recent hosts to their tables is accomplished by updating the time for active entries and removing the entries if no frames are received with that address after some period of time (**aging time**) [29].

2.1.4 Loop Prevention

A reliable and fault-tolerant network is required to have redundant links. These extra links imply loops in the network, which can degrade user connectivity and even destroy it entirely. Unlike the IP layer, which has a time to live (TTL) counter, which decreases for every (Layer 3) device that passes through, Layer 2 devices do not have any built-in protection against loops. [25]. The solution to this problem is to use the Spanning Tree Protocol (STP). There are many different versions of this protocol, such as Rapid Spanning Tree Protocol (RSTP) and Multiple Spanning Tree Protocol (MSTP), but the first STP is standardized in *IEEE 802.1D 1998* [14]. STP prevents loops by creating a spanning tree in the network, which keeps only one link between two nodes. Furthermore, STP blocks the traffic from any other link that is not part of this topology.

2.2 Software Defined Networking

We can divide computer networks into three different planes: the data, control, and management plane. The data plane is represented by the networking devices, which are responsible for forwarding the traffic. The control plane is the network's control logic, meaning that it represents the policies used to populate the forwarding tables of the networking devices, and the management plane can access remotely and configure the policies of the control plane using services such as Simple Network Management Protocol (SNMP) [22]. So basically, the management plane defines the policy, the control plane compels the policy, and the data plane forwards the data based on the above [28].

Traditional IP networks, although widely used, are tough to manage. These networks have many types of equipment such as routers, switches, middleboxes, firewalls, network address translators, server load balancers, and intrusion detection systems [21]. The data and control plane is tight together in the networking device, making the whole structure completely decentralized. As for the management plane, a programming interface is used by the network operators in order to apply the network policies (which are becoming more and more complicated) to each network device separately using low-level commands that usually are unique for each vendor and even across different products from the same vendor [28]. The challenges mentioned above have decreased the innovation rate, increased complexity, running, and operational costs.

Software-Defined Networking (SDN) is an emerging architecture that breaks the vertical integration by decoupling the control plane and data plane of network devices. With the uncoupling of the control and data plane, network devices take over a much simpler role, which is to forward the traffic, and the control plane can be managed by a centralized controller, which facilitates the network configuration, policy enforcement increases the innovation rate.

With these actions, the need for a programming interface between switches and the SDN controller is mandatory. Southbound Interface (SI) provides communication between data and the control plane using the southbound API. Furthermore, the Northbound Interface (NI) uses the northbound API in order to translate the high-level policy rules from the management plane to low-level instruction sets for the control plane, as shown in Figure 2.1.

2.2.1 SDN Applications

We are going to provide some case examples that SDN is used for.

2.2.1.1 Monitoring and Measurement

Network architectures are very complicated, and the amount of data needed to handle them is higher than ever. So it is critical to know if we take as good advantage of our resources as possible. SDN provides applications that give us the luxury to measure the traffic between the links and monitor the latency. The above information can make the network operators alter the traffic flow, alert or optimize the network.

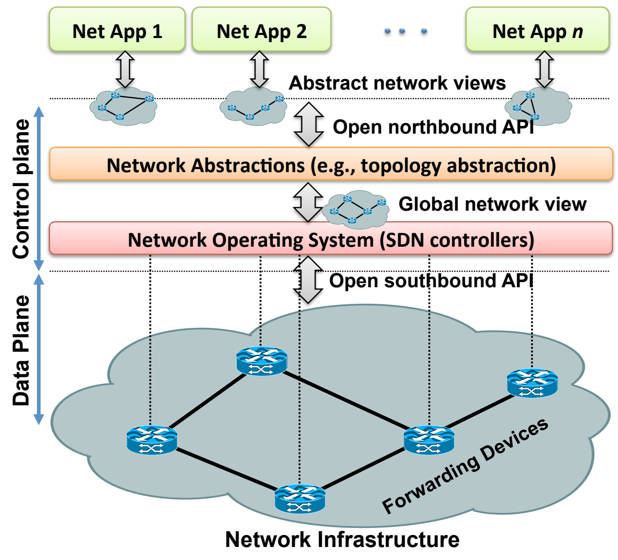


Figure 2.1: Caption

2.2.1.2 Security

Network Function Virtualization (NFV) combined with SDN creates a proactive environment supporting virtual services that run in the network layer, reducing the risk of harmful attacks as well as responding much quicker to similar incidents. When a security breach occurs, it is crucial to identify it quickly and ensure that other network components are safe. In order to accomplish that, the integration of security services into SDN creates a more proactive environment.

2.2.1.3 Content Availability

Service providers use content servers in order to deliver or cache the media that users need. These content servers need to distribute the media across multiple geographic areas and do it efficiently, so they do not affect the Quality of Service (QoS). SDN applications are easy to expand across an entire data center and because abstracting the network controls allows for easy and efficient transport of data across the data center.

2.2.2 Openflow

The most common application programming interface (API) in SDN is OpenFlow [34]. Although it started as an academic research project between Stanford University and the University of California at Berkeley, OpenFlow gained a big part of

[0] asdfsadfsafd

the industry. Google deployed Openflow to interconnect its datacenter backbone network across the globe [27]. Furthermore, Alibaba, ATT, the U.S. National Security Agency (NSA), and Microsoft are deploying OpenFlow, which proves that SDN and especially Openflow is a protocol that more and more companies are adopting.

When a switch supports the OpenFlow API, we call it OpenFlow switch. The basic idea of how OpenFlow switches operate is that most switches and routers are using TCAM's to construct the flow tables, which are used to implement firewalls, NAT, QoS, etc. OpenFlow switches have extracted the most common flow-table functions from the industry switches and implemented them on their switches. This way, Openflow creates a protocol that is open, simple, and user-friendly.

The essential parts of an OpenFlow switch are three. The first one is the **Flow Table**, which has an action associated with each flow entry, the second one is the **Secure Channel** that connects the switch with the remote controller, and the third one is the **OpenFlow Protocol**, which, as we previously said, it provides the way for the controller to communicate with the switch (over the Secure Channel).

Na anaferw meionektimata tou openflow kai pinakaki apo p4 14 paper.

2.3 P4 Programming language

2.3.1 Overview

Programming Protocol-independent Packet Processors (P4) is a programming language which first presented by Pat Bosshart and his team in 2014 [18]. Two years after, in 2016, Mihai Budiu and Chris Dodd presented a new version of the language with $P4_{16}$ as a reference [19], and in order to distinguish the two versions, when we are referring to the first one, we use $P4_{14}$.

P4 is a high-level language that works in parallelism with multiple SDN controllers such as OpenFlow. Although OpenFlow is the most common SDN controller, it has a significant disadvantage: it operates only in explicitly specified protocol headers. The more header fields OpenFlow supports, the more complex the specification becomes.

P4 language was developed with three main goals: **Reconfigurability**, **Protocol Independence**, and **Target independence**. Reconfigurability indicates that programmers can change how the forwarding devices process packets after they have deployed. Protocol independence refers to forwarding devices that should not be bound to a specific network protocol, and target independence aims that programmers are able to configure how packets are processed regardless of the hardware device [18].

Industrial networks are constantly adding new encapsulation layers to the packets to improve security or for better regulation, which extends the controller's API specification and creates the need to develop new software switches that can manage these new protocols. P4 is based on application-specific integrated circuits (ASICs) that can reach speeds up to a terabit. These chips can provide adjustable mechanisms for parsing packets and reconfigurable match-action tables, which are essential for adding new header fields without upgrading the forwarding devices.

Figure 2.2 shows the relationship between P4 and the target device. Depicts how P4 is used to configure the target device, as well an API (such as OpenFlow or P4Runtime) that is designed to populate the fixed function switches.

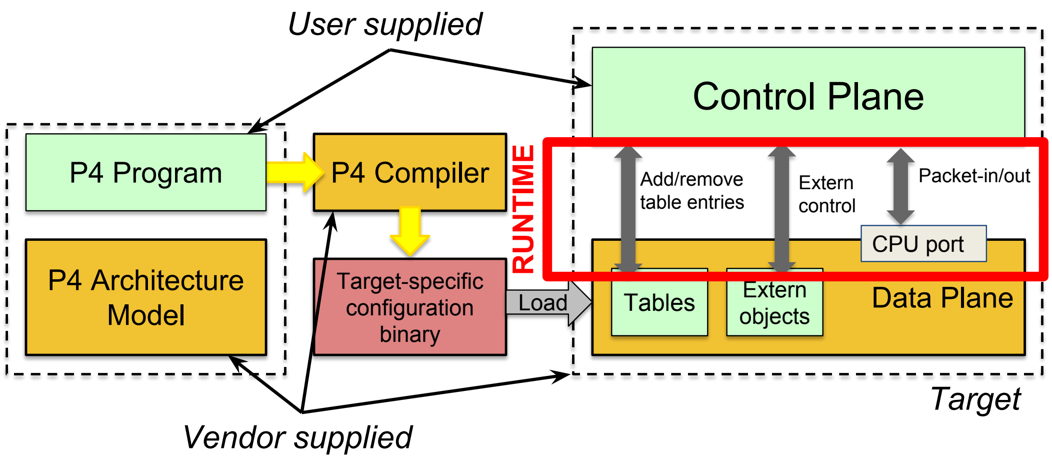


Figure 2.2: Programming a P4 switch
Source: P4.org, ONF

2.3.2 Advantages of P4

P4 provides a number of serious benefits compared to some state-of-the-art forwarding devices that can be adjustable (e.g managed switches)

- **Portability** P4 can construct refined packet processing algorithms using general-purpose operations and table look-ups that can be used across hardware targets that have the same architecture.
- **Flexibility** P4 has the ability to express the packet forwarding policies of the network as programs, compared to hardware-specific functions of traditional switches.
- **Further decoupling** P4 target devices can use abstract architectures to further decouple the low-level architectural details from the high-level processing.
- **Component libraries** Target manufacturers can supply component libraries to wrap hardware-specific functions into portable high-level P4 functions.
- **Debugging** Network administrators can write P4 programs based on a specific architecture to aid in the debugging process.

2.3.3 The design of P4

$P4_{16}$ language is the latest version that P4.org has published; it is a language with a syntax based on C, statically typed, and designed for packet processing. A P4 program consists of **parsers**, **deparsers**, and **control blocks** [19]. Parsers are the ones that receive packets in a byte format and translates them to packet headers. Deparsers, on the other side, transforms the headers into a complete packet in order to send it to the network. Control blocks guide the packets to the appropriate operations, like modify the headers of the packet. Furthermore, P4 does not support pointers, dynamic memory allocation, float numbers, and recursions. Loops are allowed only on the parsing stage.

2.3.4 $P4_{16}$ datatypes

In P4, all values are statically typed, and the programs that fail in type-checking are invalid. The main datatype in P4 is a bitstring of a specified width. For example, **bit**<64> declares a bitstring of 64 bits which is used in order to represent **integers**. Other data types that p4 includes are: **booleans**, **enumerators** including **error** type for express error codes. P4 is able to construct derived types such as tuples, structs, headers, arrays of headers, and unions of headers [19]. Structs and unions are inspired by C language.

Headers describe the format (the set of fields, their ordering and sizes) of each header within a network packet.

User-defined metadata are user-defined data structures associated with each packet.

Intrinsic metadata is information provided or consumed by the target, associated with each packet (e.g., the input port where a packet has been received, or the output port where a packet has to be forwarded).

Parsers describe the permitted header sequences within received packets, how to identify those header sequences, and the headers to extract from packets. Parsers are expressed as state-machines.

Actions are code fragments that describe how packet header fields and metadata are manipulated. Actions may include parameters supplied by the control-plane at run time (actions are closures created by the control-plane and executed by the data-plane).

Tables associate user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex decisions depending on many fields. At runtime tables behave as match-action units [8], processing data in three steps:

- Construct lookup keys from packet fields or computed metadata,
- Perform lookup in a table populated by the control-plane, using the constructed key, and retrieving an action (including the associated parameters),
- Finally, execute the obtained action, which can modify the headers or metadata.

Control blocks are imperative programs describing the data-dependent packet processing including the data-dependent sequence of table invocations.

Deparsing is the construction of outgoing packets from the computed headers.

Extern objects are library constructs that can be manipulated by a P4 program through well-defined APIs, but whose internal behavior is hardwired (e.g., checksum units) and hence not programmable using P4.

Architecture definition: a set of declarations that describes the programmable parts of a network processing device.

Figure 2.3: Core abstractions of the P416 programming language [1]

2.3.5 P4$_{16}$ architecture

P4$_{16}$ language is applicable in a wide range of target devices, which may differ in the type of processing or kind of capabilities. As far as the type of processing is concerned, some devices have to forward packets (e.g., switches), other devices have to receive/transmit packets (e.g., network cards), and others have to allow or block packets (e.g., firewalls). For the different kinds of capabilities, P4 is applicable in ASIC and FPGA devices. ASICs may have custom checksum hardware and FPGAs because they are programmable, may have, for example, custom queuing

[1]Fron P4 paper

mechanisms.

A hardware device represents the data plane, and the P4 language interacts with it. The data plane has some programmable blocks, and the P4 program is trying to manage the actions of each block. The manufacturer of each hardware device is responsible for providing the P4 architecture file, which contains the description of the control and parser blocks, type declarations, and constants that the programmer has to implement. [19].

The architecture model for the P4 language is as important as the C standard library is for C programming language [11]. In order to make P4 language applicable to a wide range of target devices, we need to have these targets conform to a specific model. This way, P4 programs can be portable across different devices as long as the latter support the same architecture file. The most known architecture model is the Programmable Switch Architecture (PSA). PSA specification is owned by P4.org Architecture working Group, and it is the most widespread architecture for multi-port Ethernet targets (e.g., switches). Another popular architecture is the V1Model which was created as an intermediate solution until PSA was properly defined.

2.3.6 PSA

The Portable Switch Architecture (PSA) specifies six programmable blocks and two fixed-function blocks, as depicted in Figure 2.4

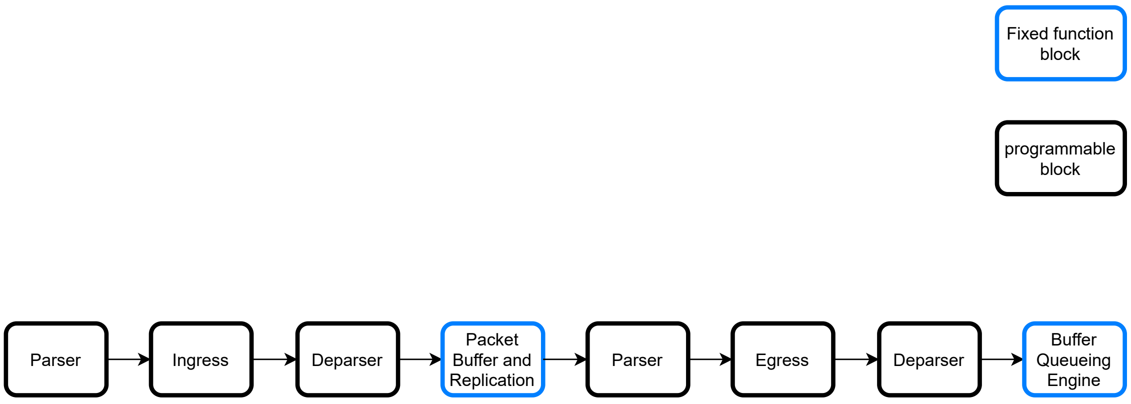


Figure 2.4: PSA blocks

The incoming packets are going through the **parser** block and then passed to the **ingress** block where a match-action pipeline decides where the packets go. After that, the packets enter the **deparser** block where the programmer specifies the metadata and the contents that each packet should carry to the **packet buffer**. The packet buffer might store the same packet multiple times if the programmer replicates it at the ingress block. The egress block uses a match action pipeline in order to send the packets to the appropriate egress port, where they will enter a queue to leave the pipeline. You can find more information about the PSA architecture at PSA architecture

2.3.7 V1Model

The V1Model architecture was designed based on the $P4_{14}$ switch architecture, so basically, V1Model is able to translate $P4_{14}$ programs to $P4_{16}$ programs. In this architecture, we have only six programmable blocks as depicted in figure 2.5. The main difference compared to PSA architecture is that the parsing process is done only once at the beginning and the deparsing process before the packet leaves the switch. Furthermore there are some extra programmable blocks, the *Verify Checksum* and *Compute Checksum*. The first block computes the checksum of the packet and compares the result with the checksum value attached to the header, and the *Compute Checksum* computes the checksum right before the deparser assembles the packet again and updates the checksum value on the header of the packet.

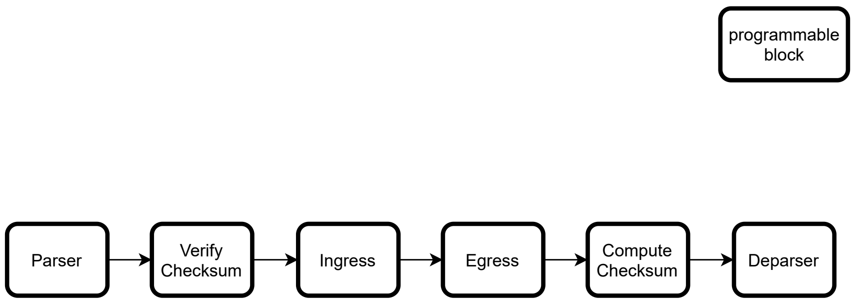


Figure 2.5: V1model blocks

2.3.8 Behavioral Model V2

Behavioral Model Version 2 (BMV2) is a framework that supports several software target switches such as *simple_switch*, *simple_switch_grpc* and *psa_switch* and has become very popular since many programmers use this framework to test P4 programs in an emulated environment such as mininet. It is written in C++11, and the most popular target device is the *simple_switch*. BMV2 is a developing tool for testing and debugging P4 data planes [1], so the throughput and latency of BMV2 are not comparable with a production-grade P4 switch like Barefoot Tofino's. In order to run P4 programs in BMV2, we need to compile the P4 code into a JSON format that the BMV2 switch can accept.

P4lang [8] provides three compilers for the P4 language, *p4c*, *p4Runtime* and *p4c-bm*. *P4c* and p4Runtime are the recommended compilers to use since they are the only ones that are currently maintained. Let us assume that we are compiling our p4 program with p4c compiler; it will provide us a JSON file that is going to be "fed" to the bmv2 switch binary [1]. In this thesis, we are using the V1model architecture since the target devices are BMv2 switches.

2.3.9 Apache Thrift RPC

In SDN architectures, we need a Southbound API in order to accomplish connectivity between the control plane interface and the forwarding devices in the data plane. The *simple_switch* target of the BMV2 architecture uses Thrift Remote Procedure Call (RPC) API in order to populate or change values to the tables of the running P4 program.

Apache Thrift is a library developed at Facebook in 2007 [15], which has as its purpose of deploying a reliable and efficient communication channel between different programming languages. In order to accomplish that, Apache Thrift uses the Thrift IDL (Interface Definition Language) file, which allows developers to define datatypes and service interfaces, and generate the necessary code by compiling that file in order to build RPC clients and servers that communicate as if it were local.

2.3.10 gRPC

Another RPC framework is gRPC which was introduced by Google in 1993 [40] and in the last years has become the most popular open-source RPC framework available. Like Thrift RPC, gRPC's primary goal is to create a communication channel between different applications, even if they use different programming languages. GRPC runs a server that can handle client calls and one or more clients in different languages, also called stubs.

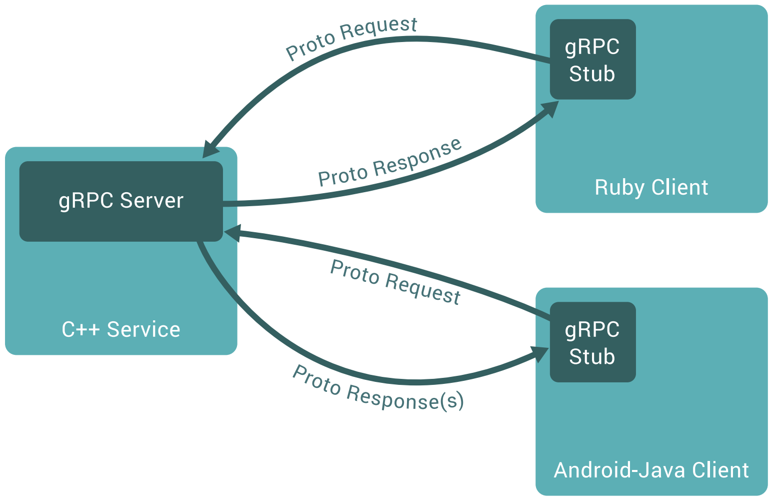


Figure 2.6: gRPC

Instead of using Thrift's IDL, gRPC uses Protocol Buffers, Google's open source mechanism for serializing structured data. In order to use Protocol Buffers, developers need to define the structure for the data they want to serialize in a proto file with a **.proto** extension. Then the proto file is passed to the protocol buffer compiler called **protoc** in order to generate data access classes to the preferred language [4].

Besides the usage of protocol buffers, gRPC owes its success to the adoption of the HTTP/2 protocol. With this change in 2015, gRPC gained some significant advantages compared to HTTP/1.1 that was using until then. Some of them are the **Binary framing layer** which makes sending and receiving messages compact and efficient by dividing them into smaller messages, **multiple parallel requests** which allows bidirectional communication and multiple requests at the same channel. Finally, **streaming** allows not only real-time communication but in high performance also thanks to the binary framing that we referred to previously.

Many big companies such as Cisco, Juniper, Netflix, etc., have adopted gRPC because of its high performance, flexibility, and very supportive community. So BMV2 provides the *simple_switch_grpc* switch target, which instead of communicating with the controller using Thrift RPC, uses gRPC. The compatible controller with the *simple_switch_grpc* target is P4Runtime API which we will briefly describe in the following chapter.

2.3.11 Mininet

Mininet is a network emulator written in python that allows the user to built a complete virtual network on a single computer. It supports virtual hosts, switches, controllers, and the links between them. All nodes in mininet run standard Linux network software, and the user is able to execute a wide range of commands through a Command Line Interface (CLI). Mininet provides an extensible Python API which allows the creation of custom network topologies and experiments with them.

The user can interact with the nodes, either from the CLI, where he can execute commands such as ping, traceroute and bring up/down links between nodes, or using *xterm* command, which opens a Linux shell for the device it refers to.

2.3.12 $P4_{16}$ example

In this section, we will demonstrate an example of a P4 program. This program implements a basic forwarding device for IPv4. The switch will perform the following actions for every packet: (i) update the source and destination MAC address, (ii) decrement the time-to-live (TTL) in the IP header, and (iii) forward the packet to the appropriate port.

A P4 program starts by including the standard $P4_{16}$ library (**core.p4**), and the library that describes the architecture of the device, in this case, we have the **v1model.p4** since the device is a bmv2 switch.

The following code is a fraction of the V1model.p4 architecture file and presents the programmable blocks that the user has to implement. The first block is a **Parser** block called *Parser* and his role is to identify the headers that are present in each incoming packet. We also have 5 **control** blocks which are the *VerifyChecksum*, *Ingress*, *Egress*, *ComputeChecksum*, and *Deparser*. All the above control blocks intend to further process the packet.

```
/*
 * Architecture.
 *
 * M must be a struct.
 *
 * H must be a struct where every one if its members is of
   type
 * header, header stack, or header_union.
 */

parser Parser<H, M>(packet_in b,
                    out H parsedHdr,
                    inout M meta,
                    inout standard_metadata_t
   standard_metadata);

/*
 * The only legal statements in the body of the
   VerifyChecksum control
 * are: block statements, calls to the verify_checksum and
 * verify_checksum_with_payload methods, and return
   statements.
 */
control VerifyChecksum<H, M>(inout H hdr,
                             inout M meta);
@pipeline
control Ingress<H, M>(inout H hdr,
                      inout M meta,
                      inout standard_metadata_t
   standard_metadata);
@pipeline
control Egress<H, M>(inout H hdr,
                     inout M meta,
                     inout standard_metadata_t
   standard_metadata);

/*
 * The only legal statements in the body of the
   ComputeChecksum
 * control are: block statements, calls to the
   update_checksum and
 * update_checksum_with_payload methods, and return
   statements.
 */
control ComputeChecksum<H, M>(inout H hdr,
                              inout M meta);

/*
 * The only legal statements in the body of the Deparser
   control are:
 * calls to the packet_out.emit() method.
 */
@deparser
control Deparser<H>(packet_out b, in H hdr);

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

The program usually continues with the type definitions, which can be used to create a new type providing the name followed by the size of it. Using the keyword *header*, the programmer is able to define new structures of header formats that the switch needs to recognize since P4 aims to be protocol independent language has to be flexible about the different headers of incoming packets.

For the specific example, we declare the headers that each packet carries, along with the size of each field. We developed this program to handle only the ethernet and IPv4 header.

```
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;


typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
```

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

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
struct metadata {
    /* empty */
}

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

Based on the V1model architecture, the user has to provide a **parser** block in the P4 program. The following block of code, which starts with the keyword *parser* uses a state machine and defines how the P4 program should parse the packets. The state machine returns a set of headers that it has extracted from the incoming packet according to the programmer's instructions.

The parsing process begins in the *start* state, and using *transition* keyword can switch from one state to another which in this case are the parse_ethernet and parse_ipv4 state, and finishes when the states *accept* or *reject* are reached.

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

    state start {
        transition parse_ethernet;
    }

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

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

}
```

This architecture, besides parser block, usually has many control blocks. A **control block** consists of **action** and **table** declarations. We can have multiple actions and tables, but only one **apply block** which is the one that triggers table lookups. As for the apply, it is a loop-free program that implies the order and under which circumstances the tables are *applied* to packets.

The action and table declarations are the primary methods that the programmer changes how the switch behaves to the incoming packets. The **actions**, are code fragments that operate similar to functions. They may include parameters given by the control plane that can be executed by the data plane. **Tables** consists of user-defined keys, and actions. You can refer to figure 2.3 to find out more about tables.

Ingress and egress blocks contain match action tables. Ingress blocks use them to determine the egress port(s) of the packet. Furthermore, in ingress processing, it is able to forward packets, replicate them (using multicast or send them to the control plane), drop them, and trigger a flow control.[18]

In our example, we have one table named *ipv4_lpm* which uses the *dstAddr* of IPv4 header for each packet and tries to match against using the least prefix match (lpm). This table can trigger either the *ipv4_forward*, *drop* or *NoAction* actions.

Ipv4_forward receives the destination Mac address and the port that the packet is supposed to leave from the switch and sets them to the appropriate fields of the packet. Drop action simply drops the packet.

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t
   standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }

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

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

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

The egress match/action table can recieve 4 types of packets. *Normal Unicast* (NU), *Normal Multicast* (NM), *Cloned from Ingress to Egress* (CI2E), and *Cloned from Egress to Egress* (CE2E). NU and NM packets come from the ingress control block, and in egress block can be recirculated using unicsat or multicast packet paths. For CI2E and CE2E, basically the packets cloned from the ingress and egress block respectively and reentered the egress block.

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

Finally the Deparser state is where the programmer declares how the output packet will look on the wire. In our case where we have two headers, we declare the order that the headers are going to be combined and sent out of the swtich.

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

2.4 P4Runtime

P4Runtime [9] is a vendor-independent, protocol-independent runtime API for P4-defined data planes. It was built to augment the programmatic API definition expressed in Protobuf format [10]. P4Runtime uses gRPC protocol 2.3.10 which allows the user to write the Controller in a wide range of programming languages and using protocol buffers to communicate with the target device.

2.4.1 Architecture

The architecture of P4Runtime is depicted in Figure 2.7. We observe that our target device, which is at the bottom, is connected with multiple controllers which are at the top. In order to have multiple controllers above one target device, we need a *P4 Master Controller* and as many *P4 Slave controllers* as we need. A multi-parser protocol ensures that only one controller has the right to access the target device.

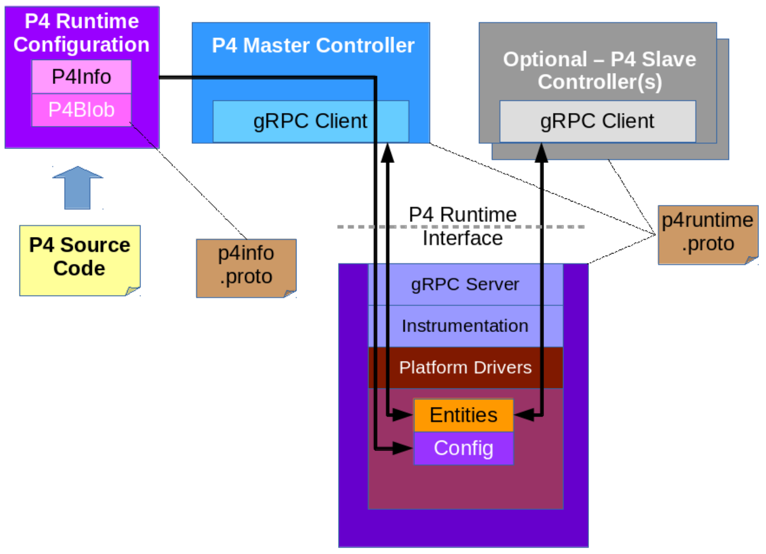


Figure 2.7: P4Runtime Architecture
Source: P4.org, ONF

P4Runtime API is able to construct messages between the interfaces of the controllers and the server(target device). For that purpose, the API is using a Protobuf file named *p4runtime.proto*. Another Protobuf file is the *P4info.proto* which describes the structure of P4Info metadata. The controller can access P4 entities which are declared in P4Info metadata using *p4info.proto*.

In a paradigmatic workflow, a P4 source program is compiled to produce both a P4 device configuration file and the P4Info metadata. The p4Info is designed to be target and architecture independent; however, the specific contents are architecture dependent due to the compiler, which rejects incompatible code.

2.4.2 Single Embedded Controller

A target device has an embedded controller that can communicate using P4Runtime. This is the simplest use-case of the P4Runtime controller and the most common in P4 examples. P4Runtime is designed to be an ideal RPC and an IPC (Inter-Process Communications). Figure 2.8 shows the architecture we mentioned above.

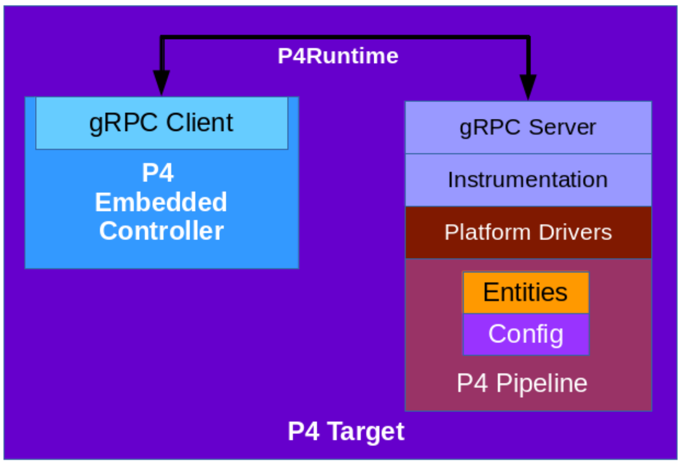


Figure 2.8: Embedded P4Runtime
Source: P4.org, ONF

Chapter 3

Related Work

In this chapter, we present previous work that is related to our research subject. The content of this chapter is based on the literature review that was conducted as part of my master thesis. We will refer to some notable tries to develop a tool that helps debug an SDN network and discuss the capabilities of each one, as well as their advantages and disadvantages.

3.1 Where is the Debugger for my Software-Defined Network?

Inspired from gdb [2], the authors of this paper [24] introduced *ndb*, a prototype network debugger for SDN which implements two major primitives which came up very helpful for its cause. The first one was *brakepoints*, and the second one was *packet backtraces.*

The authors take advantage of Software Defined Networks(SDNs) in order to debug the network control programs like gdb debugs software programs. The architecture of this tool is simple. Since traditional Openflow switches cannot stamp information in the packets that they forward, they have programmed the switches to send a "postcard" every time, a packet visits a switch. A postcard is a truncated copy of the packet's header, followed by the matching flow entry, switch id, and output port. Then these postcards for every switch are sent to a Collector, which stores them and is able to create backtraces for every packet using the information in the postcard alongside the given breakpoint. Furthermore, they have a Proxy unit to modify the messages from the controller and tell the switches to create the postcards. In order to find a postcard for a specific packet and inspects the IPID and TCP sequence number fields of the immutable header fields since forwarding rules do not modify them.

This debugger is able to backtrace the packets, identify the source of bugs, and verify data authentication. Based on *SDN Testing and Debugging Tools: A Survey* [35] the advantages of this debugger are that it does not have a framework restriction, does not need a specific SDN controller or specific programming language.

The disadvantages are that it can only identify bugs along the transmission path, and it has limitations if the packets of the network change their state dynamically.

3.2 Controller-agnostic SDN Debugging

Similar to ndb, the authors of this paper introduce *OFf* [20], a debugging and test environment for SDN developers which is built on top of the fs-sdn simulator [23]. For *OFf* to debug an application, the developer needs to include the corresponding library. Furthermore, it does not need any additional hardware to be deployed, and it does not affect the network's performance unless the developer issues a debugging command.

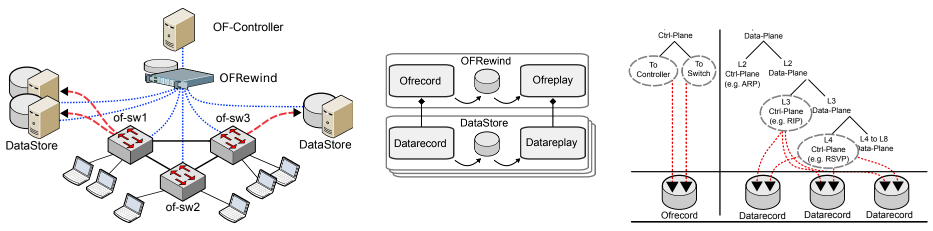
The architecture of *OFf* consists of two parts: the *OFf proxy* and *OFf Controller/Debugger Runtime Interfaces*. The proxy unit, similar to ndb, represents the communication channel between the simulated network in fs-sdn and the controller. Besides that, the proxy is based on four components, the *UI wrapper* which provides an interface for the developer to send commands to the other three components. The second component is the *Debugger*, which is responsible for several sub-modules aiming for more specific *OFf* commands from the controller API to control plane activity. The third component is the *Trace Replay* which can reproduce network activity, and the last one is the *Diff Report Generator* that detects changes in topology or the policies of the network and generates a report to help developers determine the effects of configuration changes. The *OFf Controller/Debugger Runtime Interface* is responsible for linking the *OFf* Proxy unit to a specific controller platform and language-level debugging environment.

The functionality that *OFf* debugger provides is that it can trace a packet and replay its route, alert when some configuration changes, and verify that packets passed through a specific set of switches. The advantages are that it is able to identify and eliminate bugs, detect security vulnerabilities. Finally, the disadvantages are that it is not applicable to all switch vendors.

3.3 OFRewind

OFRewind [41] is a tool that takes advantage of split forwarding architectures such as OpenFlow to improve the way that recording and replaying network domains are done. *OFRewind* enables *scalable, temporally consistent, centrally controlled* network recording and *coordinated* replay of traffic in an OpenFlow controller domain. Because of how flexible an OpenFlow controller may be, *OFRewind* can dynamically select data plane traffic for recording, which improves network scalability but also makes it possible to enable always-on recording for low-volume traffic such as forwarding control messages, which are more prone to bugs.

The main component of this system is the *OFRewind* which runs as a proxy between the switches and the actual controller. This way, it is able to communicate with every *Datastore* component that is locally attached at regular switch ports.



(a) System components (b) Software modules (c) Traffic strata

Figure 3.1: Overview of OFRewind

Both components *OFRewind* and *Datastore* can be broken down further as depicted in Figure 3.1

Although it is possible to enable always-on recording for every packet in the network, this will cause significant overhead problems in performance as well as storage. To counter this problem, the authors decided to classify the traffic and select for recording only the categories of traffic the network operators seem important. If this *selection* approach does not reduce the recorded traffic, they can apply *sampling* in packets or flows as a reduction strategy. Finally, the last data reduction approach is to record the first X bytes of each flow.

When the operators want to replay traffic, Ofreplay is responsible to re-inject the captured traces by Ofrecord into the network. There are different replay scenarios: replaying traffic towards the controller, replaying traffic towards the switches, and finally replaying traffic based on packet headers captured by the Datarecord which allows to re-generate exact flows that enable complete testing of the network. The packets may have a complete payload or dummy payload.

The advantages of this tool are that it can reproduce software errors, locate configuration errors, and replay only the desired part of network traffic. The disadvantages are that it needs a lot of memory space, depending on the device's behavior.

Chapter 4

P4Debugger

In this chapter, we present and describe P4Debugger, a tool that we created in order to help network operators debug an SDN exploiting P4 capabilities. The main functionality of this tool is to taint packets with the information of the switch that forwards them, keep a record of all flow table versions, visualize the topology and how the switches behave using a web app and monitor the network for a potential policy violation.

The Chapter describes the three parts that assemble the P4Debugger, *Data Plane*, *Control Plane* and *Visualizer*. First, we will present the workflow of P4Debugger and how the three parts interact with each other in order to inform the user about the network state. Then we will analyze the *Data Plane* and explain how we taint the packets. Then, we describe the *Control Plane* and how the P4Runtime controller interacts with the P4 switches. Finally, we will present the *Visualizer*, which is a web application that visualizes the network's behavior.

4.1 Workflow

In this section, we present the workflow of P4Debugger, which is depicted in Figure 4.1. We begin from the *Data Plane* where the P4 switches, before forward each packet, add information on the custom header we created. Furthermore, some switches are connected to monitors and have to duplicate a percentage of packets and forward it to them to check if the right policies are applied to the network.

Next, the *Control Plane* which in our case is a P4Runtime controller, interacts with the P4 switches updating their tables based on the instructions of the network administrator, is also responsible for pushing the switch's flow tables to a git repository for every packet out.

The *Visualizer* fetches the flow tables from the git repository, parses the essential data in order to visualize the topology as well as the flows that the topology has for a selected time and day.

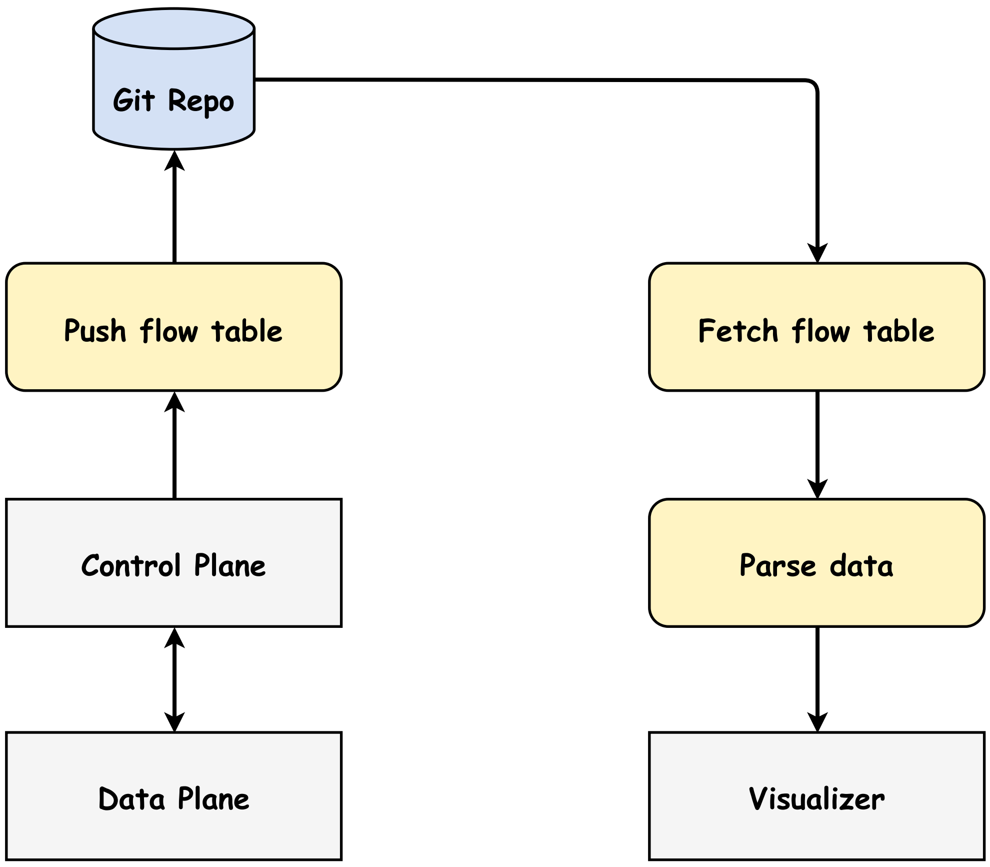


Figure 4.1: Workflow of P4Debugger

4.2 Data Plane

In an SDN, as we described in section 2.2, the data plane or Southbound interface is where the networking devices are. In our case, P4 switches are responsible for forwarding the traffic. Protocol independence is the feature of the P4 language that makes it so important in this thesis. A P4 switch has access to the whole packet header, not only to the link-layer as traditional switches. It can dissect the packet from the link-layer all the way to the payload; This is feasible because the programmer is responsible for declaring the headers and the order of them, how the parser is going to parse the packet and assign the bytes to the corresponding headers.

In this section, we will analyze the structure of a packet with emphasis on the IP layer, which is the one we used for assigning the custom header, Then we will describe the structure of the custom header as well as the workflow of the switch, and finally, we will describe how we managed a two-way communication between our switch and the controller.

4.2.1 IPv4 Header

In order to send some information over the network, we need to encapsulate the data in a header that complies with the protocol the network device has. Each network protocol, such as TCP, IP, or Ethernet has a header format with the appropriate fields and sizes. In Figure 4.2 we show how an IPv4 packet is structured.

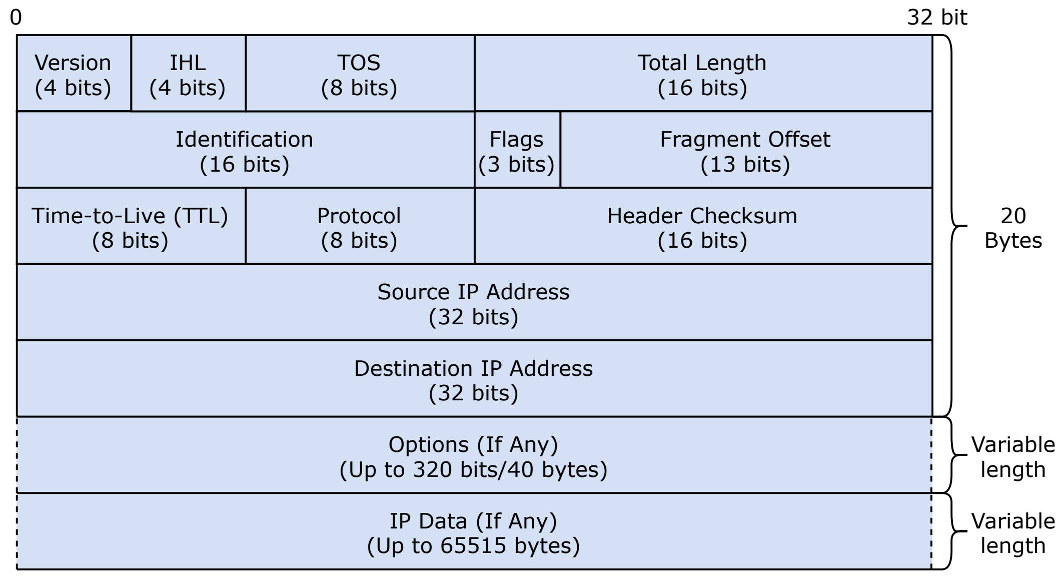

0
32 bit
Version (4 bits)
IHL (4 bits)
TOS (8 bits)
Total Length (16 bits)
Identification (16 bits)
Flags (3 bits)
Fragment Offset (13 bits)
Time-to-Live (TTL) (8 bits)
Protocol (8 bits)
Header Checksum (16 bits)
Source IP Address (32 bits)
Destination IP Address (32 bits)
Options (If Any) (Up to 320 bits/40 bytes)
IP Data (If Any) (Up to 65515 bytes)
20 Bytes
Variable length
Variable length

Figure 4.2: IPv4 Header

Here is the description of each field:

- **Version** : The version of the IP protocol, for IPv4 the value is 4.
- **IHL** : The length of the header, the minimum is 20 bytes (when Options filed does not exist) and maximum is 60 bytes (when Options field reaches the maximum length).
- **TOS** : Type of Service (TOS) uses 3 bits for IP precedence and 4 bits for TOS; The last bit is not used.
- **Total Length** : The length of the IP packet including IP header and data. The length field is 16 bits so 2^{16} - 1 = 65535 bytes.
- **Identification** : Differentiate fragmented packets from different datagrams.
- **Flags** : Control or Identify fragments.
- **Fragment Offset** : Used to fragment and reassembly the packet if needed.
- **TTL** : Limits datagram lifetime.

- **Protocol** : Describes the protocol used in the data portion. For TCP, the value is 6.
- **Header Checksum** : Value calculated based on IP header. The router drops the packet if does not verify that value.
- **Source IP** : IP address of the host that sent the packet.
- **Destination IP** : IP address of the host that should receive the packet.
- **Options** : Used for network testing, has variable length from 0 to 40 bytes and usually is empty.
- **IP Data** : The payload of the packet that may contain data and headers from higher level protocols. Has variable length from 0 to 65515 bytes.

We can observe that if we want to add an extra header in the IP layer, the only field that is not used often and has a variable length which is critical in our case, is the Options field. We need the custom header field to have a variable length because it will increase its size based on the number of switches that pass through.

4.2.2 Custom Header

As we described in this chapter, P4 allows us to declare our headers and, using the *Parser*, we can define which headers represent the appropriate bits of the packet. In this section, we will present the headers that we declare in our P4 program and the order we place them to structure a packet.

Usually, the network topology of switches can see only the link-layer (MAC) headers of every packet, except when dealing with layer-3 switches that can inspect IP-layer too. In our case, although we could forward the packets using only the link-layer, we also need to access the IP-layer to manipulate the extra header. Since we declare and use the IP header for the above reason, we decided to forward the packets based on IP tables that make the representation of use-case scenarios easier.

We created the IPv4 header exactly as described in RFC 791 [13], then we added a set of custom headers in the options field to tag each packet with the desired information. We calculated the size of each field very carefully in order to maximize the efficiency of our tool by allowing us to keep track of as many hops as possible. In Figure 4.3 we see how the Option field of IP header is structured.

The fields of the custom header are described bellow:

- **Option Type** : This 8-bit field is divided into three sub-fields:
  - **Copy Flag** (1 bit) : Declares if the options should copied to all fragments if the datagram is fragmented.
  - **Option Class** (2 bits) : Defines the category that options belong. 0 is for Control options and 2 for Debugging and Measurement.

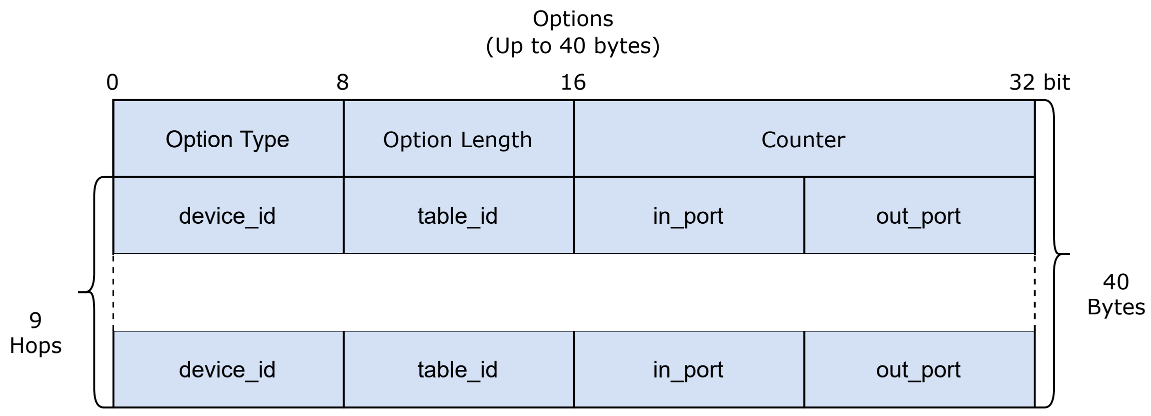


Figure 4.3: Custom Header

- **Option number** (5 bits) : Specifies the kind of option

- **Option Length** (8 bits) :Indicates the size of the entire option field.
- **Counter** (16 bits) : Specifies the number of P4 switches each packet passes through.

For each switch that a packet passes through (hop), we add a set of the following fields in order to record the information we want.

- **device_id** (8 bits) : Indicates the switch ID.
- **table_id** (8 bits) : Shows the ID of the table the packet passed through.
- **in_port** (8 bits) : Indicates the port number that the packet came from.
- **out_port** (8 bits) : Indicates the port number that the packet sent to.

IP option header has a maximum size of 40 bytes; So, assuming that each packet will have 4 bytes committed for *Option Type, Option Length* and *Counter*, we are left with 36 bytes of free space. Since the information we want from each switch sums up to 4 bytes, we are able to taint the packet up to 9 times.

4.2.3 Forwarding

Every forwarding device needs a forwarding table to send the packets to the appropriate address. In our case, P4 tables generalize traditional switch tables. P4 tables behave as match-action units with the following steps:

- Construction of a key.
- Look up for the key in a lookup table, and the result will be an "action".

- Execute the action over the input data e.g(change the source and destination MAC address of the packet).

In our implementation, the key is the destination IP address, and the table tries to match it with its entries using the longest prefix match algorithm (lpm). When the switch finds the match, it executes the corresponding action with the appropriate parameters.

4.2.4 Packet Counting Per Flow

The procedure until now in the data plane is that for every packet the switch forwards, adds some extra information in the custom header before the packet leaves the switch. This procedure allows us to have all the packets in our network topology tainted with the extra information.

The question is, how are we going to inspect this extra information? We can not expect the users to check them by themselves. The solution we came up with is to add a host for each P4 switch that will act as a monitor. Then the P4 switch will duplicate a percentage of every packet that forwards for each flow and send it to the monitor for inspection. We have to send only a portion of the packets per flow because otherwise, we were going to overwhelm the monitor with too many packets, as well as reduce the performance of the switch by half.

The challenging part of this action is that in order to count the packets and check if the specified value has been reached, we need to write and read the value. V1Model has the *counters* that are able to count quantities of packets or bytes, and *direct counters* that are increased only if the table entry that the direct counter is associated with is matched. So *direct counters* can count packets per flow, but unfortunately, only the p4 controller is able to read the value. *Registers* are stateful memories whose value can be read and written during packet forwarding under the control of the P4 program. Although some P4 switch manufacturers like Tofino have implemented a *direct register* which can be associated with table match keys and be increased every time a table entry is matched, V1Model architecture as well PSA do not support it yet.

So we created an algorithm inside the P4 switch in order to count packets per flow. The procedure we follow is not easy and demands resources, but we can avoid it in a real case scenario by using a physical P4 switch with implemented direct registers.

The procedure of counting the packets starts at the egress stage of the switch. As *flow* we defined the source and destination IP of the packet. The main idea is that we insert the source and destination IP of the packet in a hash function. The hashed value then operates as an index for storing the *flow* and the *counter* of the flow in two hash tables, the *id_ table* and *counter_ table* respectively.

For every packet that is not cloned, we check if its *flow* is already in the *id_ table*. If it does, we increase the counter at the same position in the *counter_ table* by one. In case that the specific *flow* does not exist in the *id_ table*, we assign its flow in the empty cell that its hashed value indicated and assign the number 1 at the *count_table*. However, we most probably will encounter the big problem of collision.

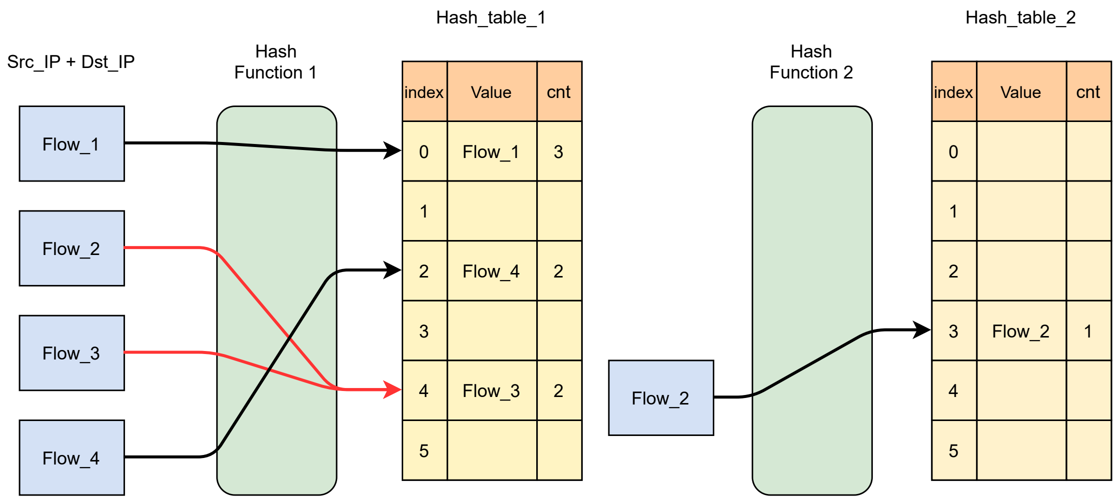


Figure 4.4: Counting Packets per Flow

Collisions will happen when the hash function returns the same hash value for two different *flows*. So for a packet from a new *flow*, we may find that the *id_table* at the given index is already filled with another flow value. The solution to this problem is to add more hash functions and tables. The same technique was proposed in the paper *Designing Heavy-Hitter Detection Algorithms for Programmable Switches* [16]. If there is a collision for a specific flow, we use the second hash function to produce the index for the second *id_table* and *counter_table* and insert the flow there.

The amount of hash functions and tables is proportional to the number of different flows we are going to have in our network. However, there is always the possibility that a flow may not be able to find a position in the *id_table*. In that case, we need to recirculate the packet and try again with a small change in the flow's data. Nevertheless, since we are using this technique only for demonstration, if we want to apply this tool in a real network, we will use hardware P4 switches that can count the packets without all this overhead. In our implementation, we chose to use three hash functions and six tables (2 for each function) to ensure that all our flows can match a table entry.

In Figure 4.4 we depict how our algorithm works for two hash function tables. We show that if *Flow_2* is guided to the index in the *hash_table_1* where we already have inserted *Flow_3*, then we pass *Flow_2* to the second hash function and try to insert it to the *hash_table_2*. We increase the counter value of each field based on how many times we meet the same flow on the packets.

4.2.5 Data Plane Workflow

In this section, we are going to describe the workflow of the data plane as depicted in Figure 4.5. We know from 2.3.7 that P4 switches have an Ingress and an Egress programmable block. We divide our actions into these two different blocks as they are the ones that we apply our data plane's logic.

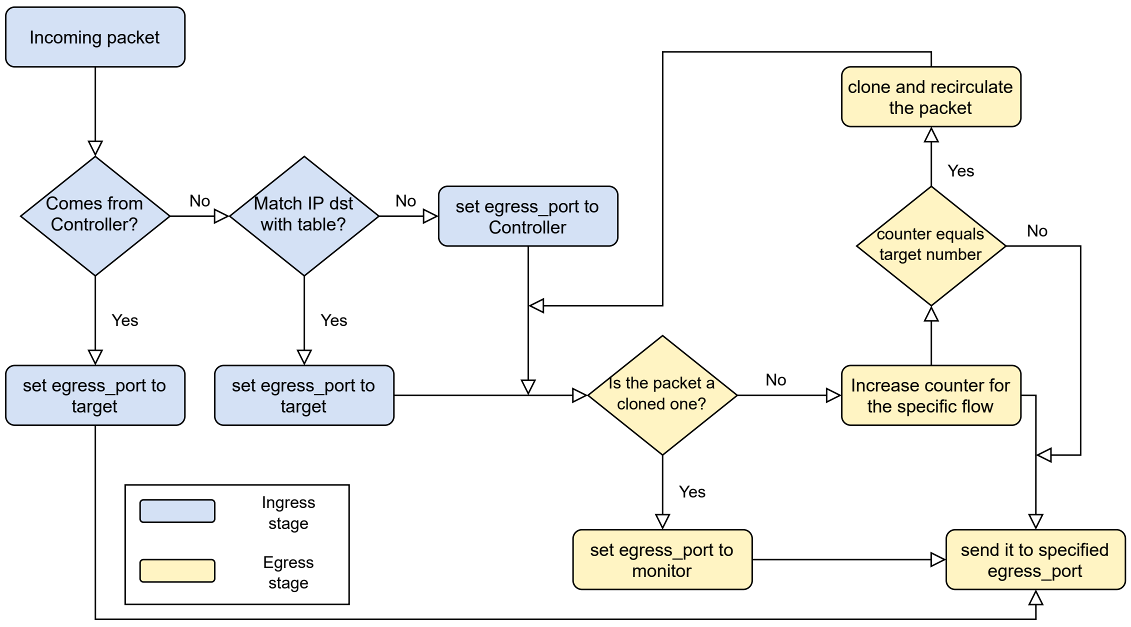

Incoming packet
Comes from Controller?
No
Match IP dst with table?
No
set egress_port to Controller
Yes
Yes
set egress_port to target
set egress_port to target
clone and recirculate the packet
Yes
counter equals target number
No
Is the packet a cloned one?
No
Increase counter for the specific flow
Yes
set egress_port to monitor
send it to specified egress_port
Ingress stage
Egress stage

Figure 4.5: Data Plane Workflow

- **Comes from Controller**: If the incoming packet comes from the controller (Packet out), it should not pass to match/action table, instead we set the egress_port of that packet equal to the port that the controller has instructed. This information is stored to a custom header of the packet that we will discuss at 4.3. If the incoming packet is not from the controller then we pass it to the match/action table.

- **Match IP Table**: Now the incoming packet passes through the match table. If there is a match we set the egress_port of the packet according to the table. Else we send it to controller by setting the egress_port number to controller's one.

- **Is the packet a cloned one?**: This state is located at egress block, If the packet is a cloned one, we have to forward it to the monitor, so we set the egress_port equal to monitor's port. Otherwise we increase the counter for the specific flow.

- **Counter equals target number**: After we increase the counter for the specific flow, we check if we have reached the desired number of packets. If the counter is equal to the target number, then we clone the packet and recirculate it to enter again the egress block. Otherwise we continue by sending the packet to the specified egress_port.

4.3 Control Plane

As described in 2.4, P4Runtime aims to provide target-independent and protocol-independent API to the control dataplane. Furthermore provides a runtime control for P4 targets, and using Protobuf allows easy serialization and supports a wide range of programming languages. Hence we decided to use P4Runtime API for our implementation.

There are two reasons we need a controller for our implementation. The first is to present how we can populate the forwarding tables of the switches dynamically, and the second is to install a two-way communication with our target devices. The population of the forwarding tables is a procedure that requires constructing the table entry in a protocol buffer representation and sent this entry to the target device using gRPC protocol.

Achieve two-way communication with our target device was a challenging task. The P4Runtime specification [9] had only a reference of how the switch can initiate Packet In with the Controller. Eventually, we managed to create Packet in and Packet out messages with the following actions.

In our P4 program, we define the Ethernet and IPv4 headers as well as two extra headers that we need in order to communicate with the controller. The structure of the packet is depicted in Figure 4.6.

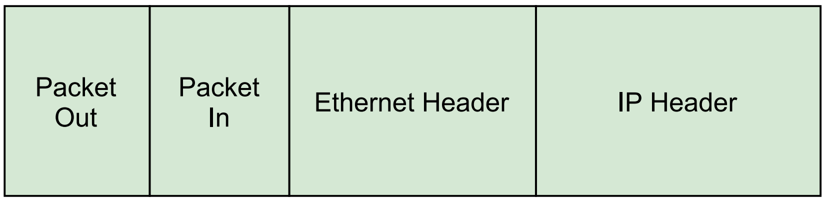


Figure 4.6: P4 program packet structure

- **Packet Out**: We declared a field named egress_port of 16 bits that indicates the port number the Controller wishes to forward the packet during the Packet Out message.

- **Packet In**: In this header, we declare a field named ingress_port where we store the number of the port that the packet entered the switch, and we forward the packet to the controller with a Packet In message.

The Packet-In message serves the purpose that the Controller is notified to populate another table in the target device. This allows us to initiate the second part of P4Debugger's implementation, which is to version the flow tables. When the Controller receives a Packet-In message, it executes a script that pushes all current forwarding tables of the switch in a git repository. After that, it populates the new entries at the target device. This way, we are able to have all table versions available.

We are using a single embedded controller in every switch as we described in 2.4.2, because it is easier for every controller to handle his Packet-In and Packet-Out messages since we have to upload the tables of each switch to the git repository. This will decrease the possibility of conflicts when we have Packet-In messages from multiple target devices. Furthermore, we could still use a master controller to manage all the embedded ones if we desire a centralized programming network.

4.4 Visualizer

In this section, we are introducing Visualizer, which is a web application we created in order to manage the different versions of tables from the git repository and provide a practical way to preview the flow rules. Figure 4.7 describes the workflow of the web application.

- **Topology**: The user uploads the topology file (JSON) to the web application, then we send the JSON file to the server. The server parses the topology file and separates it in three different tables, the *switches*, *hosts*, and the *links*. After that, it returns it to the web application where we visualize the topology using Vis.js library [12].

- **Date/Time**: The user selects a date and a time from the web page, and we forward the selected info to the server. Then the server fetches the forwarding tables for each switch, from the git repository, for the specified date and time. The switches are known from the topology file that the server parsed in the previous action.

- **Route**: The user selects a host or a switch from the web application, and inputs an IP address. After that we send these two information to the server, and using the already fetched flow tables, it can export the path from the switch/host to the given IP address. The server returns the path to the web page, and highlights the path in the topology.

4.4.1 Back-End

In order to process the data, we selected Node.js [7] which is an open-source, cross-platform, back-end JavaScript runtime environment. We chose this back-end

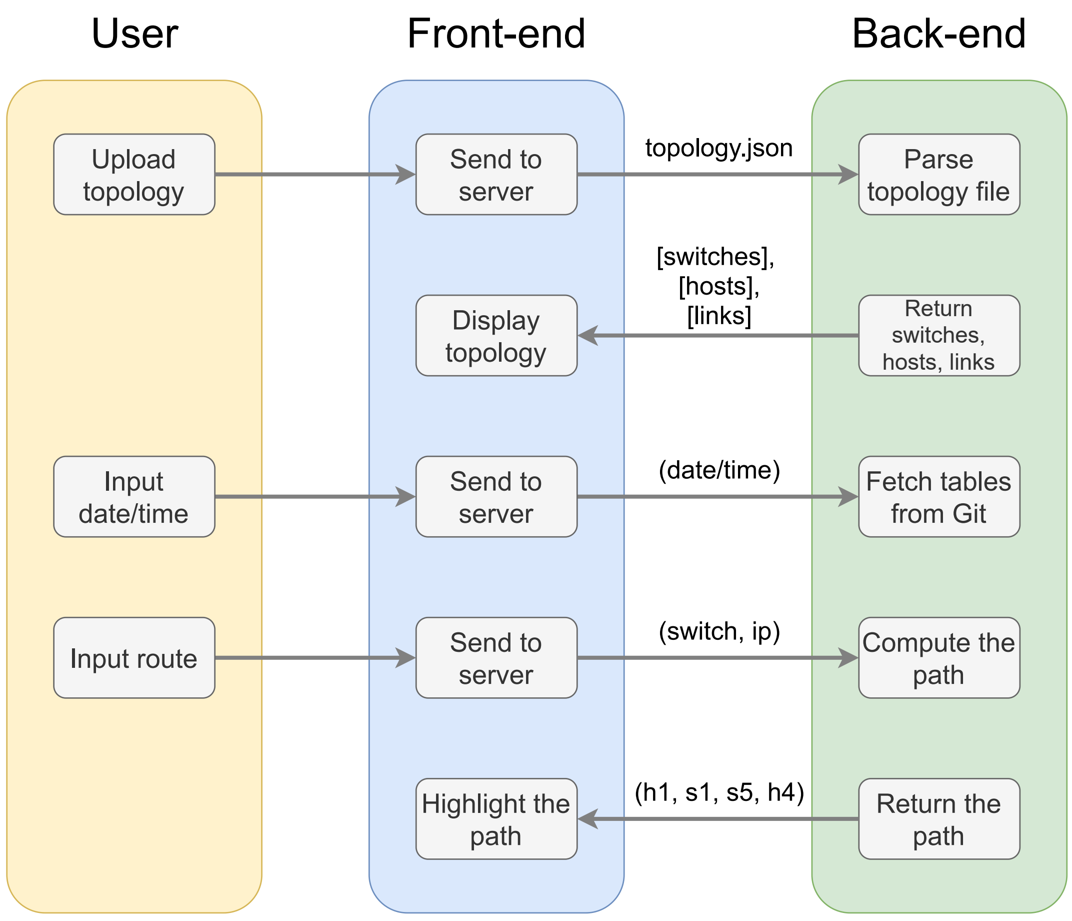


Figure 4.7: Visualizer Workflow

solution because it has a very active community with almost guaranteed maintenance. Furthermore, we would like to use a JavaScript based back-end solution because the library for visualizing the topology is based on JavaScript as well.

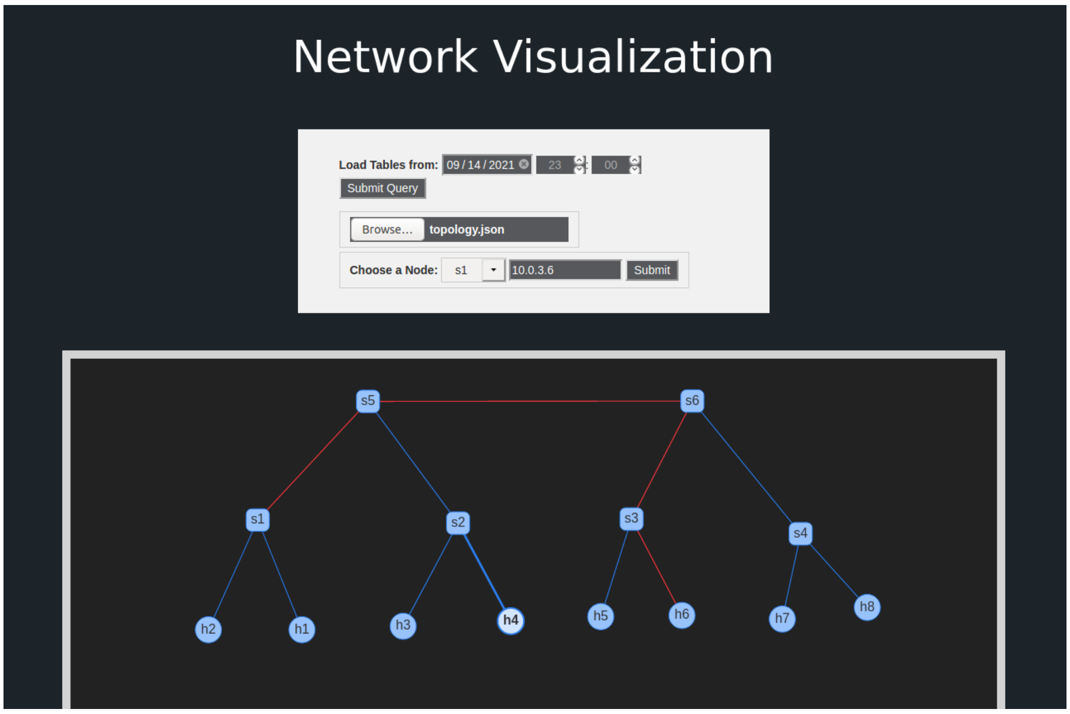
The most trivial part in the back-end implementation was to find the path in the topology, given a switch and an IP address. As information, we had the flow tables of each switch and the topology. We linked each port from every flow table with the appropriate switch based on the topology, and then, using a recursive function, we found the path that a packet should follow. We describe below the structure of a flow table, along with some values as an example.

- **S1** Flow table for the switch 1
  - **match_field**: 'hdr.ipv4.dstAddr',
  - **mac**: '00:00:00:00:11:04',
  - **table_name**: 'basic_tutorial_ingress.ipv4_lpm',

- **subnet**: '32',
- **action_param**: 'port',
- **ip**: '10.0.0.4',
- **port**: '2',
- **action_name**: 'basic_tutorial_ingress.ipv4_forward',

4.4.2 Front-End

We created a web application that allows network administrators to load any version of the flow tables to a simulated topology and preview each note's paths. This provides a reachability test for the network topology at any point in time as it can be a helpful tool to observe loops. For Visualizing the topology, we used Vis.js, which is a dynamic browser based visualization library designed to handle large amounts of dynamic data. In Figure 4.8 we present a Screenshot of our Web application after successfully found the path from switch 1 (s1) to IP address 10.0.3.6, which belongs to host_6.


Network Visualization

Load Tables from: 09 / 14 / 2021 23 00

Choose a Node: s1 10.0.3.6

Figure 4.8: Web app

Chapter 5

Evaluation

In this chapter, we will introduce and analyze the results produced from P4Debugger. First, we are going to present to you some errors that are commonly seen by SDN programmers, and then we will present some measurements that we have conducted that represent the overhead of our implementation in a network topology.

5.1 P4Debugger in action

The first error we are going to analyze is a network loop. In figure 5.1 we illustrate a network loop between three switches. These kinds of loops are very hard to detect, especially in SDN. The most common solution to this problem is to use Spanning Tree Protocol [33], which basically forms the topology in a way that there is no way to create a loop. But in SDN, Spanning Tree Protocols may not be applicable due to the dynamic programming. So the only way to detect that there is a loop in the topology is by noticing performance issues on the network.

With P4Debugger, these kinds of problems are easy to be traced. The monitors that can be attached to any P4 switch receive a percentage of the packets and can analyze the custom header to find the loop. In figure 5.2 we illustrate the fields that the custom header of the packet will have after completing the first loop.

The second error we are going to investigate is when two hosts (h1,h2) could not connect. Similar to the above solution, we inspect the monitor, which is attached to any of the switches that connect the two hosts. Applying backtrace to a cloned packet shows that the packet is reaching the host2, but one switch was corrupting the source and destination MAC, so the host2 rejects it.

The third error is a trivial one. Let us assume that we have programmed our SDN controller to change the flow route if a target device goes offline. This means that we will experience the new flow rules only when the specific device for any reason disconnects from the network. As long as the switch is online, there are no problems. Let us now assume that the target device goes offline on a Sunday night, and the Controller initiates the new flow rules. Due to a misconfiguration at the new flow rules, a part of the network is down until the specific target device

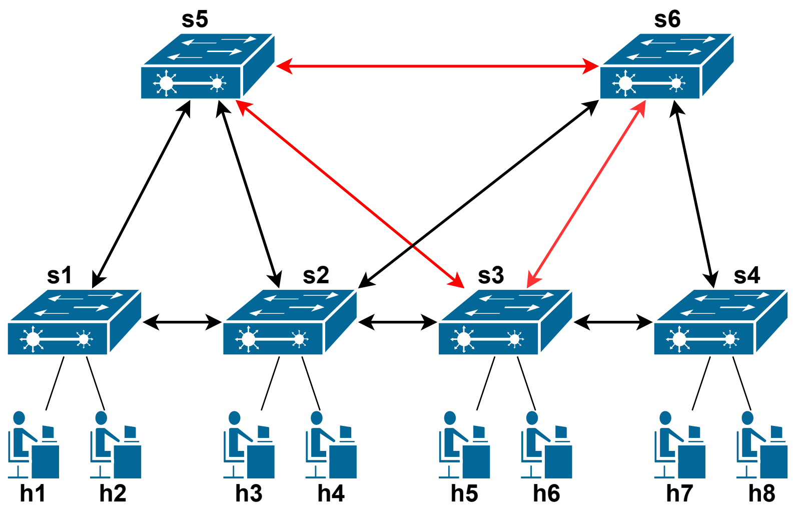


Figure 5.1: Network Loop example

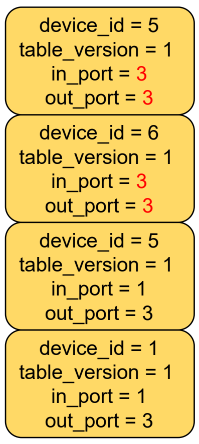


Figure 5.2: Custom header loop example

goes online again and the Controller applies back the standard flow rules. It is challenging for a network administrator to trace this kind of problem. The main reason is that they appear dynamically, so either they need to get the network to the same state as Sunday night, which means reproduce the error, or searching thousands of lines of log files to trace the bug.

Using the Visualizer, which is depicted in Figure 4.8 we can look back at the flow tables that the network topology had on Sunday night. Then we initiate the topology and start to emulate some flows until we find which switch is not responding or a flow that violates the policies.

5.2 Overhead of P4Debugger

In this section we are going to discuss the overhead that our P4program applies to the target devices. The switches that we use to deploy our debugger are bmv2 which are not meant to be a production-grade software switch. Instead, they are supposed to be used as a tool for developing, testing, and debugging P4 data planes and control plane software written for them. Because of that, the performance of bmv2 in terms of throughput and latency is significantly less than the performance of a production-grade software switch like Open vSwitch.

Furthermore the performance of the bmv2 switch depends on a variety of factors:

- which version of bmv2 code we are running.
- which flags were used to build bmv2 : This is I think the most important factor because some flags have a huge impact in the performance.
- the options we give to start *simple_ switch*.
- the performance of our hardware: how many cores and memory does our system have?
- If we are running in a physical Linux machine, or a Linux VM.

Considering all the above, unfortunately, we cannot provide a direct answer about the performance of our P4Debugger. We have decided to compare our P4 switches' bandwidth with the bandwidth that a standard P4 switch uses, without the custom header and tagging procedure. This will give us some information about the computational power our P4 program needs.

We have also added some measurements with the counting procedure implemented on our switches. As we mentioned in 4.2.4, the counting per packet flow procedure is not going to be implemented in a production grade switch because most of them have a dedicated register for that purpose. However, it highlights the difference in performance between a simple forwarding procedure, a forwarding and packet tainting procedure, and a forwarding packet tainting and counting per-flow procedure.

5.2.1 Measurements

We used iperf [6] to compute the bandwidth and Gnuplot [3] to graphically represent the measurements. Furthermore, the P4 program runs in a Linux Virtual Machine, with four cores (intel i7 6700HQ) CPU and 16 GBytes of RAM. In figure 5.3 we observe that the P4 switch that was programmed only to forward the packets, and the P4 switch that was creating a custom header and adding information on it for every packet, have almost the same performance. However, when we add the counting procedure above that, we experience a severe bandwidth drop. Finally, in Table 5.1 we illustrate the percentage difference between the actions of tainting the packets, and tainting and counting them, and the Simple forwarding action. We observe that the tainting actions has no impact at all in the performance, compared to the simple forwarding action. But when we add the counting process, the impact in performance is approximately 100 % worse.

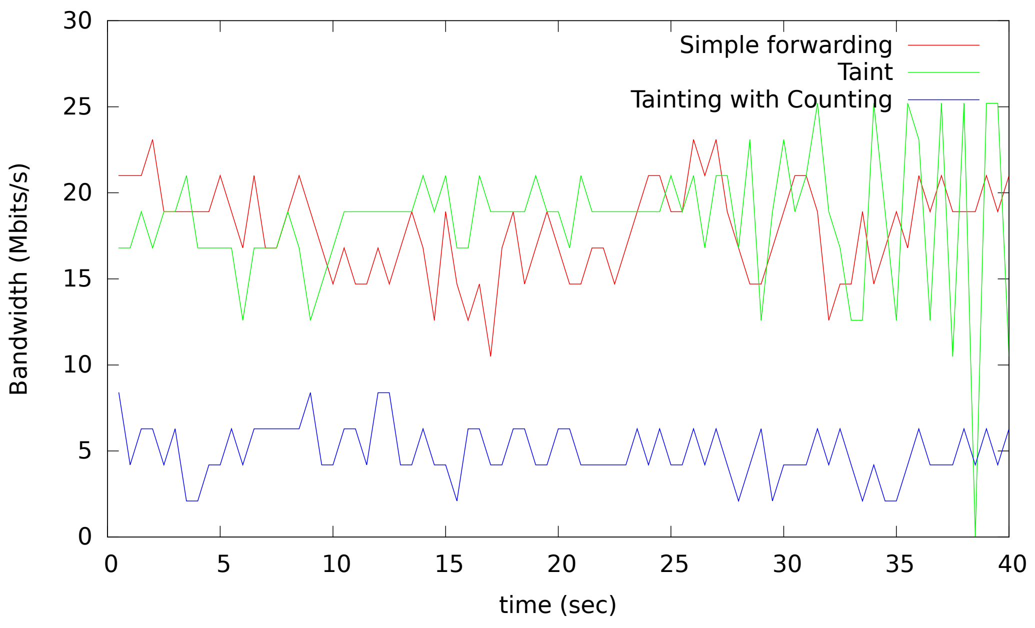


Figure 5.3: Default TCP measurement for 1 hop

Table 5.1: Default TCP measurement for 1 hop

	Mean	Median
Tainting	-2.4 %	0 %
Tainting and Counting	113 %	127.4 %

In order to distinguish the performance between our P4 switch and a basic forward P4 switch, in figure 5.4 we used iperf to send tcp packets with a maximum buffer size of 150 bytes. This means that the target device will have to deal with a much larger amount of packets for the same amount of time, considering that the default TCP buffer size is 128 KBytes in iperf. The figure 5.4 confirms our thought, and now we can clearly observe the impact in performance that our P4Debugger has. From the Table 5.2, we observe that the process of tainting the packets, decreases the bandwidth by approximately 40% based on simple forwarding action, while the counting process, by 135 %. Nevertheless, it is highly impossible to have a consistent amount of that small packets. Finally, the p4 program that taints the packets and counts them based on their flow, we see that the performance is significant lower.

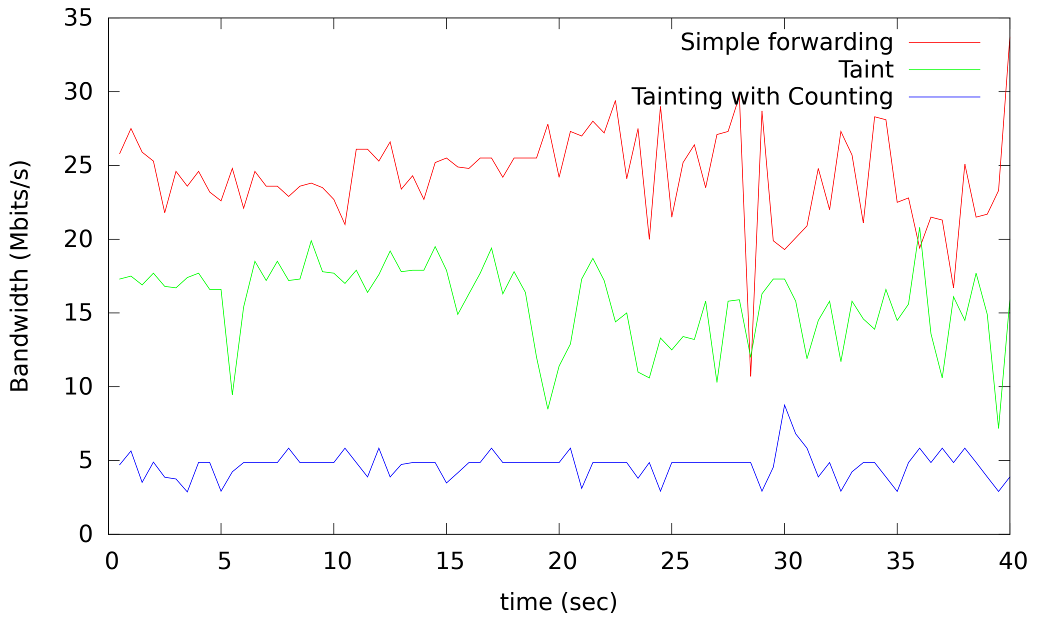


Figure 5.4: TCP measurement for 1 hop and 150 bytes of packet size

Table 5.2: TCP measurement for 1 hop and 150 bytes of packet size

	Mean	Median
Tainting	43.3 %	40.3 %
Tainting and Counting	135 %	134 %

The last two figures have been produced from the same iperf instruction as the previous ones, but in a different topology. This time we measure the bandwidth between 2 hosts that are four switches (hops) away. This will give us a more solid sample of the actual overhead compared to the basic forwarding switch. In figure 5.5 we observe that even though the measurement is performed in 4 P4 switches that run our packet tainting procedure, we have almost the same performance in bandwidth as the simple forwarding switch. The Median difference of the process of tainting based on simple forwarding, is 0 %, and the Median difference is 13%

as shown in Table 5.3.

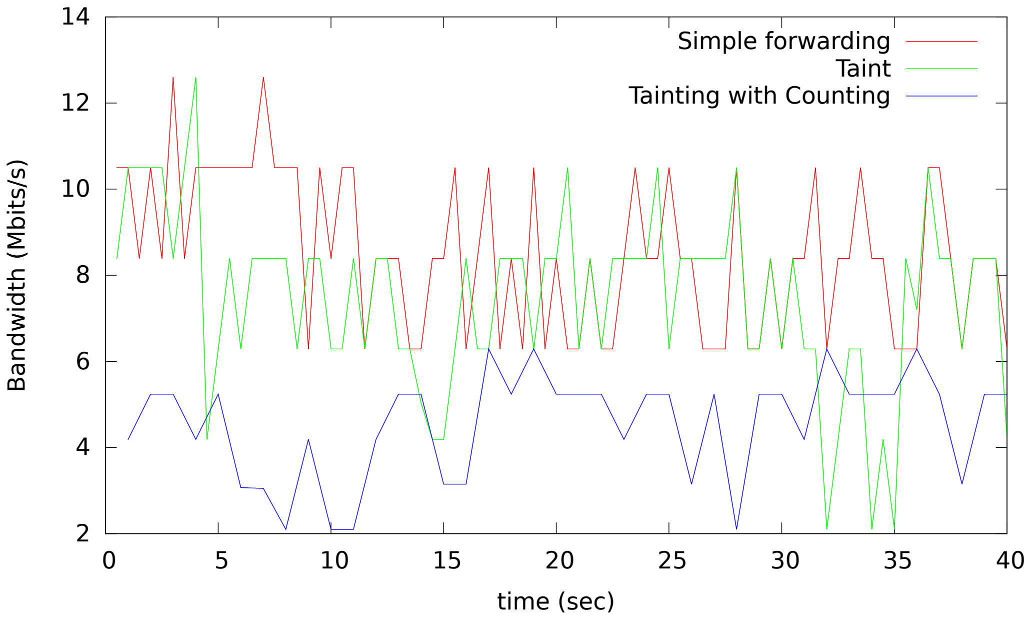


Figure 5.5: Default TCP measurement for 4 hops

Table 5.3: Default TCP measurement for 4 hops

	Mean	Median
Tainting	13 %	0 %
Tainting and Counting	60.5 %	46.2 %

The last figure 5.6 confirms once again that if we try a very small size of packets, we end up stressing out our implementation, which clearly has a difference in performance compared to the simple forwarding one. The packet tainting and counting per flow program preview a worse performance in all scenarios. Table 5.4 informs us that we have an impact in performance of 50 % if we use the tainting process, and approximately 150 % if we also count the packets.

In Conclusion, the implementation of the packet tainting process performs very well in normal size packets, although we have to understand that BMv2 switches are not suitable to provide reliable performance tests. As we expected, when we tried to send a very small size of packets, the performance decreased. The counting process was a heavy task, and the impact on performance is obvious in all scenarios.

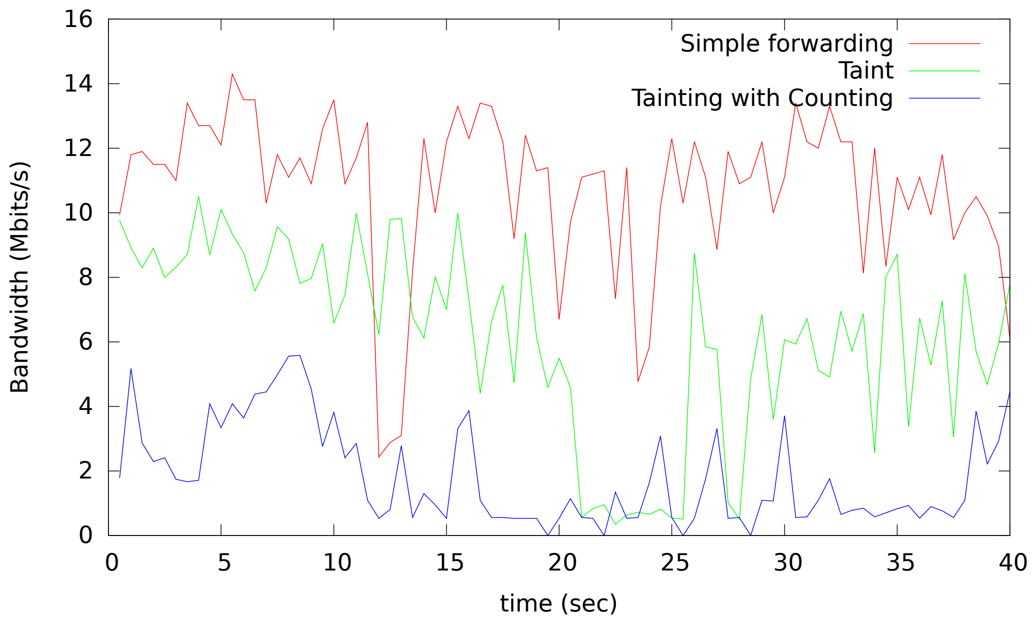


Figure 5.6: TCP measurement for 4 hop and 150 bytes of packet size

Table 5.4: TCP measurement for 4 hop and 150 bytes of packet size

	Mean	Median
Tainting	53.7 %	50.4 %
Tainting and Counting	142 %	164 %

Chapter 6

Conclusions & Future Work

6.1 Conclusions

SDN aims to make networks easier to manage. However, this is done by pushing complexity into SDN control software itself. Just as sophisticated compilers are hard to write but make programming easy, SDN control software makes network management easier, but only by forcing the developers of SDN control software to confront the challenges of asynchrony, partial failure, and other notoriously hard problems inherent to all distributed systems.

The techniques for troubleshooting SDN control software are very complex and may include inspection of logs in the hope of finding the triggering input. In this master thesis, we implement a tool that will help SDN programmers trace possible errors in their network with ease and reliability. P4Debugger applies to all the P4 programmable networks, and we showed that it is capable of quickly revealing network errors, that in other cases, it would take time and probably many resources to find.

Our method enables packet backtracing through the custom header we have implemented, which helps network programmers detect and resolve logic bugs, such as network loops and protocol compliance errors. Furthermore, we are able to version the forwarding tables of the target devices, which enables us to detect bugs and possible misconfigurations that may dynamically occur from migration events or any other dynamic programming instruction.

Based on the Results 5.2, although we cannot completely evaluate our tool because of the constraints of the bmv2 switch, we proved that despite all these constraints and limitations, our implementation performs almost the same with a P4 program that simply forwards packets. So we expect that a production-grade P4 switch like Tofino is able to run our P4 program with zero constraints. That said, if we try to forward a stream of very small packets, then we experience some performance drop. P4Debugger is a valuable contribution to SDN programmers because, with a very small trade off in performance (it may be almost no-existent in a production P4 switch), we solve the problem that SDN networks created.

6.2 Future Work

We see the following research and engineering directions as interesting future work.

- **TraceMac** : Using an extra field in the custom header of the packet, we could program the switch to perform traceroute, but in the Link Layer. In more detail, every time the switch receives a packet with the raised flag in the specified field, it will send back a packet from where it came from, and eventually, it will reach the host who initiated TraceMac, and inside the packet, there are going to be all the information about the switches and ports the packet came through. The difference with our implementation is that with TraceMac, we can make an active path exploration of our topology.

- **Policy analysis tool** : The current approach does not allow us to insert policies that our network must comply with. We can only backtrace a packet to see in which switch the error occurs. We need to implement a framework where the network administrator can input the desired policies, and the framework will check the cloned packets for any violation. This way, the Network operator will react instantly and mitigate the problem.

- **Open Source P4Debugger** : The current version of the framework is in the stage of prototyping. We have to test our P4 program to a production grade P4 switch in order to validate the produced measurements. After that, we can open source the code of the tool in order to attract more developers to maintain and supporting the tool.

- **All-in-one API** : We plan to combine the Policy analysis tool, along with the TraceMac, and the Visualizer in an API where the Network operator will have a suite of active and passive monitoring tools for debugging and policing the network.

Bibliography

[1] Behavioral-Model. https://github.com/p4lang/behavioral-model.

[2] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/documentation/.

[3] Gnuplot. http://www.gnuplot.info/.

[4] gRPC official page. https://grpc.io/docs/what-is-grpc/introduction/.

[5] Intel Tofno 2 P4 swtich. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[6] Iperf. https://iperf.fr/iperf-doc.php.

[7] Node.js. https://nodejs.org/en/.

[8] P4 language. https://github.com/p4lang.

[9] P4Runtime. https://p4.org/p4-spec/p4runtime/v1.0.0/P4Runtime-Spec.pdf.

[10] P4Runtime Protobuf. https://github.com/p4lang/p4runtime/tree/v1.0.0/proto.

[11] $P4_{16}$ Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA.html.

[12] Vis.js. https://visjs.org/.

[13] Internet Protocol RFC 791. https://rfc-editor.org/rfc/rfc791.txt, September 1981.

[14] Ieee standard for local area network mac (media access control) bridges. *ANSI/IEEE Std 802.1D, 1998 Edition*, pages 1–373, 1998.

[15] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 4 2007.

[16] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking*, 28(3):1172–1185, 2020.

[17] Nikos Bizanis and Fernando A. Kuipers. Sdn and virtualization solutions for the internet of things: A survey. *IEEE Access*, 4:5591–5606, 2016.

[18] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[19] Mihai Budiu and Chris Dodd. The p416 programming language. *SIGOPS Oper. Syst. Rev.*, 51(1):5–14, September 2017.

[20] Ramakrishnan Durairajan, Joel Sommers, and Paul Barford. Controller-agnostic sdn debugging. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 227–234, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014.

[22] Mark Fedor, Martin Lee Schoffstall, Dr. Jeff D. Case, and James R. Davin. Simple Network Management Protocol (SNMP). RFC 1098, April 1989.

[23] Mukta Gupta, Joel Sommers, and Paul Barford. Fast, accurate simulation for sdn prototyping. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 31–36, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the debugger for my software-defined network? In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, page 55–60, New York, NY, USA, 2012. Association for Computing Machinery.

[25] Bruce Hartpence. *Packet Guide to Routing and Switching.* O'Reilly Media, Inc., 2011.

[26] Raj Jain and Subharthi Paul. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, 51(11):24–31, 2013.

[27] Rowan Klöti, Vasileios Kotronis, and Paul Smith. Openflow: A security analysis. page 1, 2013.

[28] Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey, 2014.

[29] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson, Boston, MA, 7 edition, 2016.

[30] Yong Li and Min Chen. Software-defined network function virtualization: A survey. *IEEE Access*, 3:2542–2553, 2015.

[31] Kshiteej Mahajan, Rishabh Poddar, Mohan Dhawan, and Vijay Mann. Jury: Validating controller actions in software-defined networks. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–120, 2016.

[32] Subhasree Mandal. Experience with b4: Google's private SDN backbone. Santa Clara, CA, July 2015. USENIX Association.

[33] M. Marchese and M. Mongelli. Simple protocol enhancements of rapid spanning tree protocol over ring topologies. *Comput. Netw.*, 56(4):1131–1151, March 2012.

[34] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[35] Gilbert N. Nde and Rahamatullah Khondoker. Sdn testing and debugging tools: A survey. In *2016 5th International Conference on Informatics, Electronics and Vision (ICIEV)*, pages 631–635, 2016.

[36] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. *SIGCOMM Comput. Commun. Rev.*, 43(4):207–218, August 2013.

[37] Peter Perešíni, Maciej Kuźniar, and Dejan Kostić. Dynamic, fine-grained data plane monitoring with monocle. *IEEE/ACM Transactions on Networking*, 26(1):534–547, 2018.

[38] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 418–431, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting blackbox sdn control software with minimal causal sequences. *SIGCOMM Comput. Commun. Rev.*, 44(4):395–406, August 2014.

[40] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. Grpc: A communication cooperation mechanism in distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(3):75–86, July 1993.

[41] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. Ofrewind: Enabling record and replay troubleshooting for networks. 06 2011.

[42] Yinbo Yu, Xing Li, Xue Leng, Libin Song, Kai Bu, Yan Chen, Jianfeng Yang, Liang Zhang, Kang Cheng, and Xin Xiao. Fault management in software-defined networking: A survey. *IEEE Communications Surveys Tutorials*, 21(1):349–392, 2019.