Juan Sebastian Mejia Vallejo

Architecture and implementation of a Broadband Network Gateway using a programmable dataplane processor

Campinas

2018

UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Juan Sebastian Mejia Vallejo

Architecture and implementation of a Broadband Network Gateway using a programmable dataplane processor

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação.

Supervisor: Prof. Dr. Christian Rodolfo Esteve Rothenberg

Este exemplar corresponde à versão final da tese defendida pelo aluno Juan Sebastian Mejia Vallejo, e orientada pelo Prof. Dr. Christian Rodolfo Esteve Rothenberg

Campinas

2018

Inclua aqui a folha de assinaturas.

Abstract

Broadband Network Gateways (BNGs) play a crucial role in today's networks, it hands all access network traffic (e.g., DSL traffic), provisioning of such policies like rate limiter, tunneling, and complex network management mechanisms that a Internet Service Providers (ISPs) needs to implement his services. These network devices are expensive, proprietary, limited and slow device upgrading, become a point of failure to deploy new features, add functionalities and correct issues on the network without disrupting the normal service operation.

In this work, we propose a software-based virtualized BNG device running on inexpensive commodity hardware using high-level language for programming protocol-independent packet processors over a Multi-Architecture Compiler System to remove some of these barriers to innovation.

Keywords: Computer Networks; Software Defined Networking; OpenFlow; Future Internet.

Resumo

Adicionar resumo em portugues...

Palavras-chaves: Redes de Computadores, Software Defined Networking, OpenFlow, Internet do Futuro.

Contents

The road for a dream is not a solitary path. During this arduous and long walk, some faces keep up by your side until the end, while others come and go in the middle of the way. Regardless the time spent with you, everyone leaves marks and contribute, for the good and for the bad, with your personal growth. For this reason, I would like to dedicate this work to everyone that at some point in my life, helped me to go through this process and reach my aspirations.

Acknowledgements

My thanks to my parents, whoever has motivated me to go further and give me the best example of dedication.
I am grateful for my family, for his support in the distance and send me his best wishes and prays.

Christian Esteve Rothenberg, my supervisor, and friend. Many thanks to INTRIG group for bringing me support many times and even for its friendship outside of the university.

Ericsson Innovation Center Brazil, for sponsoring this work, and Ericsson Traffic Lab in Hungary, for technical support.

"The path to OpenFlow is not a four lane highway of joy and freedom with a six pack and a girl in the seat next to you, it's a bit more complex and a little hard to say how it will work out, but I'd be backing OpenFlow in my view"

Greg Ferro

List of Figures

List of Tables

1 Introduction

The advent of new services like Video-on-Demand (Vod), video conference, Virtual Private Network (VPN) and cloud-based new services, has increased the demand for access to broadband services (The Organization for Economic Co-operation and Development (OECD), 2016). In addition, many further access technologies such as xDSL, optical access and wireless technologies such as WiMAX and LTE require rapid deployment of services and devices guaranteeing performance in Internet Services provider (ISP) networks.
In an Internet service provider (ISP) network the Broadband Network Gateway (BNG) has the function of managing all access network traffic (e.g., DSL traffic) and other critical functions like to allow access and authentication for thousands of subscribers, monitoring, establishing sessions, tunnels and controls the user line rate. The fact that all sessions tunnels (e.g.., PPPoE, GRE) are terminated at the BNG means that is aggregated in a single point causing poor performance and quality of service. Therefore is no surprise that this device becomes expensive and hardware proprietary boxes, often the operator pay for some functionality that won't be used and the hardware boxes upgrade require a long wait until the next version available.
In recent years, in order to resolve this problem to turn this rigid hardware device into a software-based network and reduce time to market of new services and those functionalities have been decomposed and dynamically instantiated at different points of the network. This concept is following the trend of Software-Defined Networking (SDN) and Network functions virtualization (NFV) that turn some network functionalities into virtualized software processing running on a server (e.g., off-the-rack x86 servers), switches or even cloud computing infrastructure (HAN *et al.*, 2015).
Our approach to create an fully open and programmable BNG data plane is using a high-level language for programming protocol-independent packet processors (P4), this is a Domain-specific language (DSL) with a number of functions optimized around network data forwarding. Such a DSL can support customizing the forwarding behavior of the switch and may also be ported to other hardware or software switches that support the same language. In this work, we present an approach to implementing a BNG software switch implementation, and it is built on top of the MACSAD Switch target and P4 language that can provide dynamic and flexibility to the service provider to optimize the traffic on the network.
We discuss the architecture and protocols to deploy and concepts around the BNG software switch.
The rest of this document is structured as follows: Section II provides the background and

related work. Section III The problem statement and objectives, Section IV and Introduces the proposed design and some results, Section V Working plan and Execution schedule.

Objectives

To address the identified issues, the main objective of this work is to design, implement and evaluate the BNG software switch in a Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD). To this end, the following specific objectives are proposed:

For each of the issues mentioned in section **??**, we have some specific objectives in order to take action to those problems:

- Designing the architecture and functional implementation of a BNG software switch. Adding the data plane functionalities with the latest version of P4 lang and compiling our BNG p4 program in an emulation environment in order to test our P4 implementation.
- Defining and implementing P4_16 support for MACSAD. The MACSAD core works with C code in order to compile and run the software switch at execution time over a commodity server. According to the above, we need to implement the HLIR16 project and his dependencies such as P4C compiler to create an intermediate representation that brings data structure used internally in the MACSAD core to translate P4 into C.
- Integrating the BNG implementation generated with MACSAD to support the Traffic Manager ODP API. It adds QoS support to the BNG software switch. On the other hand, the controller has to interact either with MACSAD to manage table actions management as with the Traffic Manager block hence; this interface will be added as well.
- Performance evaluation with NFPA tool. Configuring a real network testing environment, software and hardware such as the described in section 5 in order to evaluate the packet processing performance with some specific use cases recommended in (NEMETH *et al.*, 2015):
  - Port Forwarding/L2 forwarding
  - L2 forwarding
  - GRE Encap/Decap

  Evaluating with unidirectional traffic and Small (64B) to large (1518B) packet sizes and with fixed destination and source MAC addresses, IP addresses and ports.

Text Structure

In this Introduction we explained the motivational aspects that justify this work. Also, we give a clear explanation for the objectives of this project.

In Chapter 2 we present a Literature Review. Related toprogramming protocol-independent packet processors (P4) language, OpenDataPlane (ODP) SDK and MACSAD framework architecture are described in the context of our implementation requisites.

In Chapter 3 we take a look at the architecture of the BNG software switch which we explain all the modules an his relationship between. Furthermore, we describe the tables associated to BNG dataplane.

In Chapter 4 we explain the BNG functionalities and features in detail.

In Chapter 5 we show the performance results and compare with related work.

Finally, in Chapter 6 we highlight the results, issues in the implementation process and future work.

2 Literature Review

This section defines three main concepts in our research work: BNG Protocol, another is the P4 language, the third MACSAD compiler and finally an ODP framework description.

Broadband Access Networks

The first generation network based on centralizes BRAS routers was driven for the customer demand for High-speed Internet (HSI) the second generation Ethernet-based Broadband Network Gateway (BNG) routers was driven by subscriber demand for a linear TV service (content broadcast at specific times, e.g., Netflix) delivered in conjunction with voice and HSI services (ALCATEL-LUCENT, 2010).
The Customer Premise Equipment (CPE) represents the triple play communications devices: Telephone (Voice), PC (Internet), Set-top box (TV) however all these devices are connected to a Home Gateway (HG) that brings the interface with the network through any access technology like Digital Subscriber Line (DSL).
One or more HG can be connected to a single DSLAM that sends the traffic to the BNG device and it route in a core IP network and the edge routers to provide connectivity to the Internet (See Figure 1).
The network operator provides connectivity, authentication, applications and service network

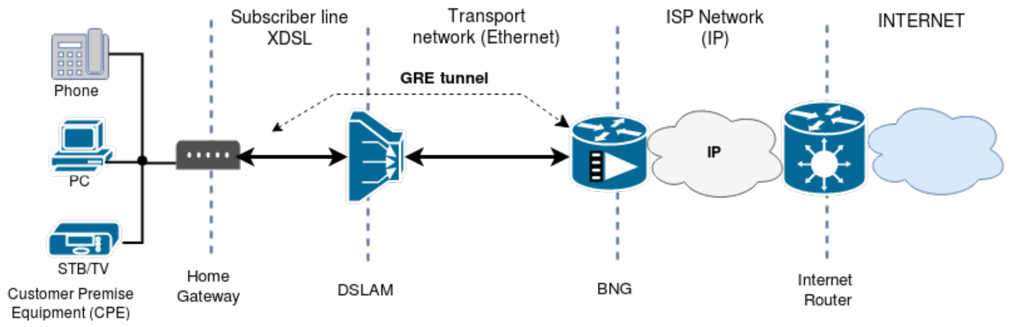


Figure 1 – Access Network Provider Model.

policies to his users, therefore, these procedures involve the premise of session establishment using access communication protocols which are managed in the BNG, the most common protocols to establish session are:

- The PPP over Ethernet (PPPoE): Use the point-to-point (PPP) protocol.

- The IP over Ethernet (IPoE): Use IP protocol that runs between CPE and BNG.
- Generic Routing Encapsulation (GRE V2), encapsulation protocol brings virtual Point-to-Point connections through IP network.

In our implementation, the packet sent or received by the HW are encapsulated with GRE headers, creating a point-to-point link with the BNG, but is just one of the options for packets encapsulation.
Since the BNG centralizes all the functions simplify the management functions like:

- Session management and header cap/decapsulation.
- Interface to Authorization, Authentication and accounting services.
- ARP proxy to manage the requests from the network interface on the BNG side.
- Network Address translation to route the packets towards the operator's core Network.
- Interface to assignment queues and line rate to subscribers.

On the other hand, all these tasks make the structure of network rigid and become difficult to support the protocols and architectures in the current ISP. In (ROBERTO *et al.*, 2013) we found some similar approach to virtualize a Broadband Remote Access Server (BRAS) based on Click OS, a tiny Xen virtual machine designed specifically for network processing it can achieve line rate of 10Gbps and it composes of netmap and VALE as the packet I/O framework.
Our architecture approach goes with the same trend of network function virtualization using Macsad framework that compiles P4 code to bring more flexibility to the data plane and adding support for Hardware Abstraction Layer (HAL).
In the next session, we will describe our architecture design to be rapidly reconfigured using a "programming protocol-independent packet processors" (P4) and MACSAD to generate the datapath Logic codes.

Programming Protocol-Independent Packet Processors (P4)

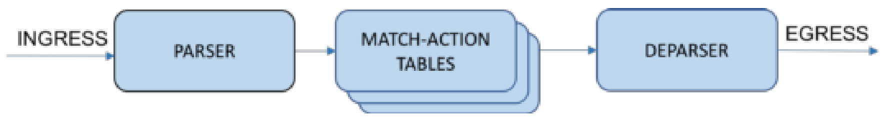


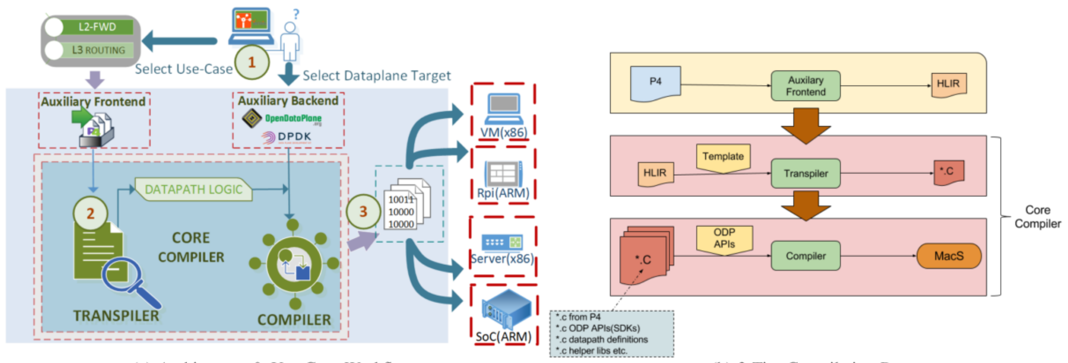
Figure 2 – P4 Abstract Forwarding Model. Source: Adapted from (P *et al.*, 2013).

P4 is a high-level language for programming protocol-independent packet processors that define how the pipeline of a network forwarding device should process the packets using the abstract forwarding model (See figure 2). P4 define the header structures and use the parser to extracts the header fields. The pipeline is defined through a series of match-action tables, which execute one or more actions like packet forwarding, drop and so on. This tables can be changed and accessed at "runtime" through a controller software to add, remove and modify table entries and finally, de-parser writes the header fields back before sending the packets to the output port. The three main advantages of P4 are:

1. Reconfigurability in the field: Programmers should be able to change the way how the switch process the packets once that was deployed.
2. Protocol independence: Capability to deploy any protocol in a switch.
3. Target independence: To describe packet-processing functionality independent of the hardware where it has been deployed (P *et al.*, 2013).

Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD)

This tool brings an environment to compile and deploy switch for L2/L3 applications automatically generating the datapath code for heterogeneous targets over 10Gbps network interfaces setup (PATRA *et al.*, 2017).
The MACSAD architecture is designed around the following three modules: (See the figure 2a)



(a) Architecture & Use Case Workflow (b) 3-Tier Compilation Process

Figure 3 – Macsad architecture. Source:(PATRA *et al.*, 2017)

- Auxiliary Frontend: The P4 code is the MACSAD input that will be converted to an IR representation using the p4-hlir framework that supports different Domain Specific Language (DSL) to generate a High-Level Intermediate Representation (HLIR) to be used in the Macsad core compiler.
- Auxiliary Backend: this module provides a common SDK for the Compiler incorporating the ODP APIs auto-generating support for different platforms with ODP supporting various control protocols like Switch Abstraction Interface (SAI), OpenFlow (OF), and other useful events for buffer like queueing, scheduling, etc.
- Core Compiler: Composed of a Transpiler and a Compiler submodule (see the figure 3b), transforms the IR generated by the frontend into the target imaged in association with the auxiliary backend.
The transpiler takes the HLIR input and auto-generates the Datapath Logic codes. Some critical decision is taken in this sub-module like to optimize the 'Dead Code Elimination' by identifying reachability in a graph, decides the type of look-up mechanism to be used, and size and type of tables to be created by. The Macsad compiler creates a set of libraries in 'C' codes for the desired target (x86, x86+DPDK, ARM-SoC).
The BNG P4 code will be added to the system as an input to create the BNG router. Also, Macsad tool brings the generation of high-level ODP APIs to deliver platform abstraction with high performance and hardware-acceleration options.

Open Data Plane (ODP)

Open Data Plane project provides an application programming environment for data plane applications with high performance and portable across a lot of HW platform. The aim of ODP is separate application design from the functional implementation of that design, this because historically it requires that data plane application must be redesigned on changes in network speed and capacity because the applications need to be very integrated with specialized hardware to achieve acceptable performance levels. ODP relies on the Linux kernel itself; it defines some ODP API set to rapid porting ODP application to any platform (See Figure 4), it can define functions and limits like a number of queues or used cores processor.
ODP applications can run in parallel with full Linux user processes that implement control and management functions since that these typically do not have critical performance and latency requirements. In that sent, this tool calls the Software Development Kit (SDK) and optimize the features and functionalities for the particular hardware platform (SoC or Server). The ODP APIs is written in the C programing language and are optimized to control related SoC resources such as:

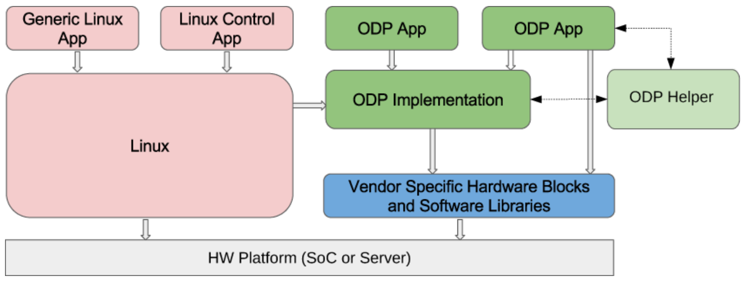

Generic Linux App
Linux Control App
ODP App
ODP App
Linux
ODP Implementation
ODP Helper
Vendor Specific Hardware Blocks and Software Libraries
HW Platform (SoC or Server)

Figure 4 – ODP system architecture. Source:....

- CPUs (or hardware threads)
- Main memory
- Huge page mappings (how many, what sizes)
- Physical and virtual ports/interfaces
- Packet classification rules
- Scheduler (core groups, algorithms, ordering)
- Hardware queues
- Output traffic management
- hardware Quality of Service (QoS) support.

3 Architecture

The Architecture overview is shown in Figure 5. We have main tree processes: (i) Creation of BNG data plane, (ii) Macsad P4_16 support and (iii) Integration with OPD APIs. Finally, this network virtualized device can be used in a commodity server such as executable program.

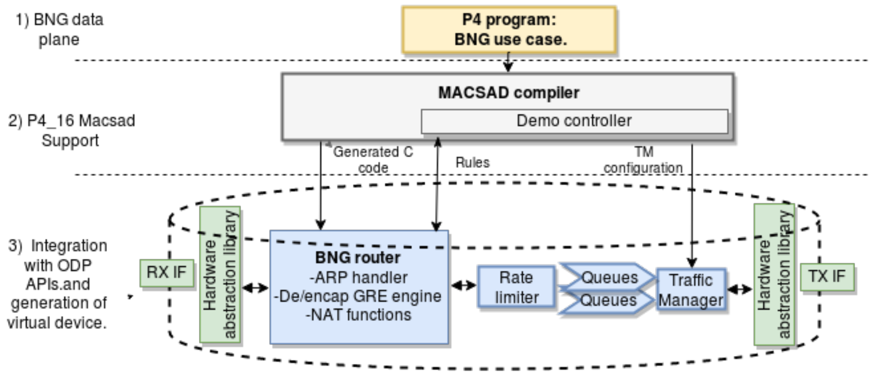


Figure 5 – BNG software architecture.

1. Create a P4 program in order to set the data plane functionalities of the BNG device (BNG data plane).

2. Adds support the the current MACSAD implementation in order to compile the last version of P4 (P4_16).

3. Integration of the BNG compiled and generated version by MACSAD with the ODP APIs, to add others features like Traffic management, Scheduling, and Rate limiter.

Details of processes are explained in the next sections. It is important to highlight that the first process was completed, the second and third are currently in progress and the latter process is included as a scheduled task in Working Plan and Execution Schedule chapter.

Creation of BNG data plane

To support the workload multiples lookup tables are supported either Upload link (UL) and Download Link (DL): a table to store MAC address, a table of CPE based on their IP address routing table, encap/decap GRE, both tables to block any port or IP using a

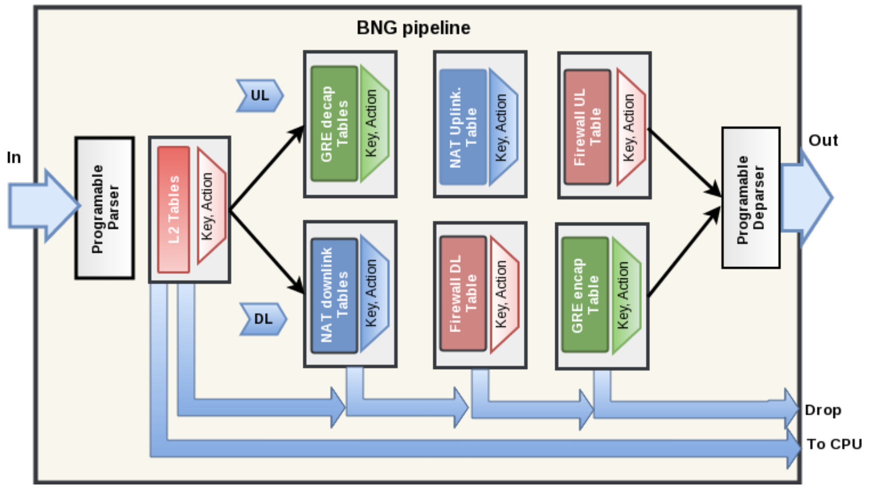


Figure 6 – BNG data plane.

Firewall for UL and DL, both table to Network Address translation (NAT) for UL and DL, the table to limiter the user rate either UL and DL. They are briefly described below.

L2 mac table

When the L2 packet is coming to the BNG switch, it uses MAC learning. Hence, when an ARP packet is received from the CPE, an entry is created (or updated) in a MAC address table swap the source and destination MAC address in the Ethernet header.
Ethernet is the protocol used in the Transport layer. Therefore, the packet header fields are read and rewrite with the new fields in this table in order to forward the packet out by the appropriate port.

Nat table

Basic full-connect NAT for TCP traffic (over IPv4) do both main functions: Translates IPv4 addresses and TCP port numbers of request packets originating from a client on a private network (iAddr: iPort) is mapped to an external address (eAddr: ePort), any packets from iAddr: iPort is sent through eAddr: ePort, the packets that his header fields matching in NAT table are processed in order to rewrite the new header and the packet is forwarded appropriately otherwise when a TCP packet is received on the external interface, for which there is no mapping, the packet is dropped.

GRE tables

The BNG will be able the forward based on the contents of a custom encapsulation header as well as perform normal IP forwarding if the encapsulation header does not exist in the packet. Our BNG is enabled to create point-to-point tunnel mechanism in the intern network to encapsulate the packets with standard GRE packet header structure, as defined by RFC 2784 and RFC 2890 and save the user ID in order to establish the user session. The BNG de-encapsulate the packets originated from CPE to an external address as well as the reverse process when the packet incoming from an external network. The GRE table encap has the outer IP address header field in order add it as new IPv4 header and the GRE header fields in order to enable the new packet.

IPV4 table

The routing table stores the next hop based on the IP address. It's based on the LPM (longest prefix match) implementation. Another table stores information related to the next hop (IP address, port index, etc.). With IPv4 forwarding, the switch must perform the following actions for every packet: (i) update the source and destination MAC addresses, (ii) decrement the time-to-live (TTL) in the IP header, and (iii) forward the packet out the appropriate port. Our BNG device will populate with static rules using a basic controller. Each rule will map an IP address to the MAC address and output port for the next hop.

Parser/Deparser

Our parser will detect the different headers used in the BNG device. It extracts headers from the packet at the current offset into per-packet header instances and marks those instances valid, updating the Parsed Representation of the packet. The parser then indicates as valid byte the correct header and makes a state transition to the next header. The de-parser reverses the process of parsing that emits headers in the proper order. Only headers which are valid are serialized, in our case, the output packet is serialized with the same protocols of the input parser block.

Rate limiter

Some research works [i.e (RODRIGUES *et al.*, 2011),(POPA *et al.*, 2013),(BALLANI *et al.*, 2011)] based on end host-based rate enforcement for fair and guaranteed bandwidth allocation in a multi-tenant datacenter. A datacenter requires a large number of rate limiters, far of the capacity of commodity NICs (e.g., Intel's 82599 10G NIC supports up to 128 rate limiters). Software rate limiters in host network stacks can scale, but this approach implies high CPU overheads hinder support for high-speed links (RADHAKRISHNAN *et al.*, 2014) We implemented a basic Rate limiter in P4 code without slow down performance for every packet to inspect such as defined in RFC 2698 (A Two Rate Three Color Marker - IETF) standard; these describe traffic policing elements like a meter and a dropper. The meter measures the traffic and determines whether or not it exceeds the rate limit, in case that it exceeds the limit the packets could be dropped.

Macsad P4_16 support

Macsad is in charge of converting our BNG P4 program into C code, specifically, it brings support to the version P4_14 (v1.0/v1.1). However, since may of 2017 has been launched the new version P4_16 (v1.0) that compared with the early version doing larges changes regarding syntax and semantics of the language. The new P4_16 specification[1] shows in figure 7 how this has been transformed from a complex language (more than 70 keywords) into a more compact language (less than 40 keywords) accompanied by a library of fundamental constructs that are needed for writing most of the P4 programs.

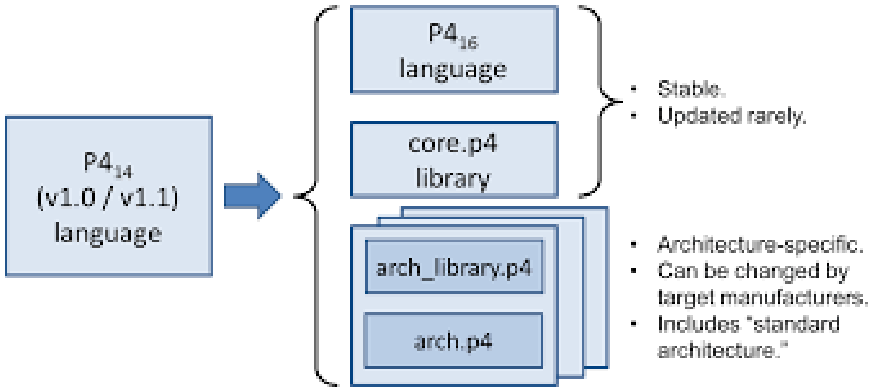


Figure 7 – Evolution of the language between versions P4_14 (v 1.0 and 1.1) and P4_16.

As we can see the section in the Frontend layer, MACSAD is using p4-hlir project which translates P4 programs into High-Level Intermediate Representation (HLIR) that creates a suitable P4 program representation (See figure 3b), that can be consumed by multiple back-ends, in our case, MACSAD will be used this tool in "Transpiler module", where is access the different P4 top-level objects using these Python ordered dictionary such as p4_parser, p4_tables, p4_actions, p4_headers and so on.
Therefore, our approach for MACSAD update to the new version is using HLIR16 project

[1] https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf

that create a convenient Python representation from a JSON file compiled in P4C compiler from a .p4 source file.

Integration with OPD API's

In order to apply some quality-of-services options for the ISP users, exist some mechanism for achieving these kinds of traffic-management goals in a shared network is through queuing and scheduling. This below features working together, deciding what packets get sent and when; thus scheduling is in charge of sending someone else's packets right now or delaying packets that are arriving too fast. In the following subsection, we will take a look at so-called Weighted fair queuing (WFQ) that provides a straightforward strategy for dividing bandwidth among multiple senders according to preset percentages.

Weighted fair queuing (WFQ)

This is a data packet scheduling algorithm based on both Fair Queueing (FQ) and the generalized processor sharing policy (GPS), where instead of giving each class an equal share, we assign each class a different percentage. For example, If all four input classes A,B,C,D are active, then each gets 25% of the total bandwidth (see figure **??**), just as for a flat four-input-class fair queuing structure. However, when only A, B and C are active, and D is idle. A, B and C each get 33%, hence it lets to allocate bandwidth according to administratively determined percentages.
ODP provides a suite of APIs that control traffic shaping and Quality of Service (QoS). Tx processing using Traffic manager and RX processing involves the use of Scheduler, both processes are described around in the next subsections.

Scheduled RX Processing

Scheduled RX processing is performed via the Scheduler and is requested when a PktIO is opened with an in_mode of ODP_PKTIN_MODE_SCHED. In a basic use, it associates the PktIO input event queues created by odp_pktin_queue_config() with the scheduler. The hash function could be employed to distribute input packets among multiple input queues and instead of these being plain queues they have scheduled queues and have associated scheduling attributes like the priority, scheduler group, and synchronization mode (parallel, atomic, ordered).

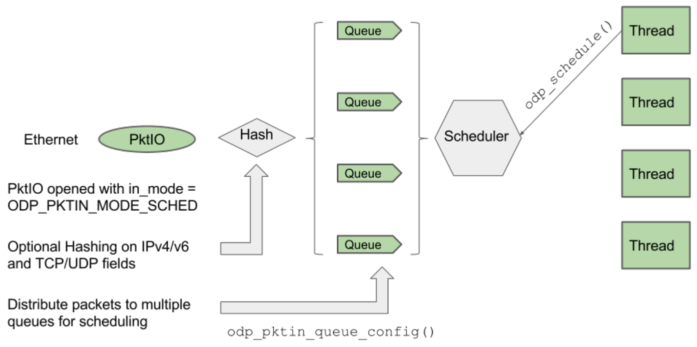

Ethernet
PktIO
Hash
Queue
Queue
Queue
Queue
Scheduler
odp_schedule()
Thread
Thread
Thread
Thread
PktIO opened with in_mode = ODP_PKTIN_MODE_SCHED
Optional Hashing on IPv4/v6 and TCP/UDP fields
Distribute packets to multiple queues for scheduling
odp_pktin_queue_config()

Figure 8 – PktIO SCHED Mode Receive Processing. Source

Inbound packets are classified and put on queues associated with their target class of service which are themselves scheduled to threads (see figure 8).

Scheduled TX Processing

Scheduled TX processing is performed via the ODP Traffic Manager and is requested when a PktIO is opened with an out_mode of ODP_PKTOUT_MODE_TM. The ODP Traffic Manager accepts packets from input queues and applies strict priority scheduling such as weighted fair queueing scheduling and bandwidth controls to decide which input packet should be chosen as the next output packet and when this output packet can be sent onwards. Weighted Fair Queuing (WFQ) is used for priority assign from multiple input packets with the same priority. Each input can be assigned a weight in the range MIN_WFQ_WEIGHT to MAX_WFQ_WEIGHT (nominally 1..255) that affects the way the algorithm chooses the next packet. If all of the weights are equal AND all of the input packets are the same length then the algorithm is equivalent to a round-robin scheduling.

A TM system is composed of Tm_nodes are "entity"/object. It lets interconnect, and interplay of a multi-level "tree" of tm_nodes can allow the user to specify some very sophisticated behaviors. Each tm_node can contain a set of WFQ scheduler (see figure **??**). Each node contains a set of "fan-in" connections to preceding tm_queues or tm_nodes.

4 Development

This chapter covers the software switch development. In section 4.1, we give details about the implementation of the most important OpenFlow 1.3 features, in section **??**, we describe the open source model adopted for the software switch development.

Software switch implementation

Implementation of an OpenFlow switch depends on the platform for which it is designed. OpenFlow hardware implementation on traditional Application Specific Integrated Circuit (ASIC) chips usually suffer from limitations, like small capacity in the total number of flows and not real support for multiple tables. Unlike hardware, software implementations offer greater flexibility in the implementation of OpenFlow features. In environments where high throughput is not the biggest concern, software switches running on commodity servers can be a low cost replacement option for traditional network switches.

The OpenFlow specification describes OpenFlow switches pipeline and the required and optional building blocks. It does not gives low level details about how these components should be implemented. As long as it works how the specification dictates, switch designers are free to use any data structures and algorithms in order to implement OpenFlow. When defining implementation details, we explored the software implementation freedom to meet the requisites defined on section 1.1. At the same time, we came up with innovative design decisions towards future extensions of the OpenFlow match field support.

In this section we discuss how we implemented the OpenFlow 1.3 software switch adding several changes to the base switch - using C [1], the switch native programming language, and C++ - in order to support all features and keep it as simple as possible. The next subsections describe this new functionalities in the context of the architecture of the software switch presented in chapter 3.

[1] In this chapter there will be two common words: struct and structure. While struct is a C language keyword and structure is a more generic word for a collection of data variables, both will be used to denote a C struct.

Oflib

The software switch architecture Marshaling/Unmarshaling library, presented in section **??**, is called Oflib. Although already present in the software switch base code, the library underwent several modifications in old messages and grew with the addition of OpenFlow 1.3 messages.

Every OpenFlow message represented by the Oflib has a common header. This header struct contains only one member, which is the message type information. Using the same initial struct for every message struct allows the implementation of two general functions that abstract marshaling and unmarshaling. In the Listing 4.1, we show the definition of these functions. Marshaling, also known as packing, is done by *ofl_msg_pack*. By passing a pointer to the struct *ofl_msg_header* for the function, we can check the message type and apply the message respective marshaling function. Unmarshaling, also known as unpacking, is performed by *ofl_msg_unpack*. In this function, the first bytes of the OpenFlow messages, the *buf* parameter, reveal their types. With this information the function calls the proper function to convert the message for the Oflib format.

```c
int ofl_msg_pack(struct ofl_msg_header *msg, uint32_t xid, uint8_t **buf,
    size_t *buf_len, struct ofl_exp *exp);

ofl_err ofl_msg_unpack(uint8_t *buf, size_t buf_len, struct ofl_msg_header **
    msg, uint32_t *xid, struct ofl_exp *exp);
```

Listing 4.1 – Oflib: message pack and unpack base functions

Another Oflib task, discussed in the section **??**, is message error handling. It checks for bad requests from the controller, for example messages with unknown types and wrong sizes are performed by every unpacking function. In case of error, the function returns the OpenFlow error code for the Datapath, which creates an error message and sends it for the controller, through the Communication Channel.

Addition of new OpenFlow messages in the Oflib is a trivial task. Firstly, the developer needs to define a C struct, with *struct ofl_msg_header* as the first member. Then, write a pack and unpack function. Finally, add the new message type for *ofl_msg_pack* and *ofl_msg_unpack*. Listing 4.2 illustrates the OpenFlow 1.3 *Role Request*, implemented during our work.

```
struct ofl_msg_role_request {
  struct ofl_msg_header header; /* Type OFPT_ROLE_REQUEST/OFPT_ROLE_REPLY. */
  uint32_t role;                /* One of OFPCR_ROLE_*. */
  uint64_t generation_id;       /* Master Election Generation Id */
};

static ofl_err
ofl_msg_unpack_role_request(struct ofp_header *src, size_t *len, struct
    ofl_msg_header **msg)

static int
ofl_msg_pack_role_request(struct ofl_msg_role_request *msg, uint8_t **buf,
    size_t *buf_len)
};
```

Listing 4.2 – Oflib message Role request struct and function definition

Additionally, the Oflib also has printing functions. This is helpful for logging and debugging in the software switch.

OpenFlow Extended Match

When compared to OpenFlow 1.1, in the number of supported match fields, the version 1.3 of the OpenFlow protocol supports nearly twice as much fields as the former version. This growth was only possible due to changes in the match structure specification. A match structure from OpenFlow 1.1 was a fixed number of fields, carrying 88 bits of information in every message carrying a new flow. Match fields not set in the message were sent, adding unnecessary space overhead. In order to keep the protocol evolution and to support more fields, the OpenFlow Extended Match (OXM) was introduced by the OpenFlow 1.2 specification. The OXM format is Type-Lenght-Value (TLV) based and replaces the old fixed match structure. A less restricted definition of the match struct adds more flexibility for the insertion of new match fields. Figure 9 shows an example of how a field is defined by an OXM field and the TLV respective sizes in bits. The Type of a match field is formed by the OXM Class and OXM Field. An OXM class represents a vendor number, where 0x8000 is the basic class for the specification of the match fields. OXM Field defines the match field. In the example, the field number 15 represents the UDP protocol in OpenFlow. The last bit of the Type is left for the Has Mask field, which indicates if the match is masked or not. Finally, the Length field is the value size.

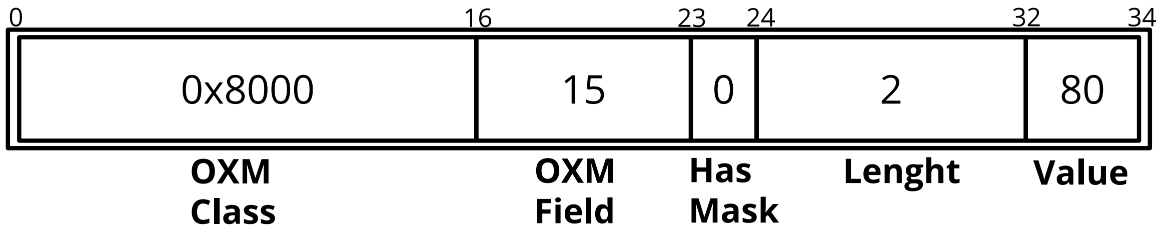


Figure 9 – OXM field example

Some challenges arise with the OXM introduction. Whereas extension of match support for messages is solved, there is nothing concerning the packet parsing in the Datapath. The next subsections discuss how our implementation deals with protocol fields extensibility in the software switch.

Packet Parser

Each new protocol added for the OpenFlow specification demands the addition of an specific code to extract the new fields. Distinct protocols may have singular and complex parsing methods. For instance, variable fields such as IP options can require cumbersome deep packet inspection. For this reason, the Packet Parser implementation needs to be flexible and easy to extend. Also, the idea of simple insertion of new match fields meets the ease of extension requisite.

As a means to achieve a Packet Parser implementation featuring the mentioned characteristics, we have come up with a design which uses a packet description language to assist the parsing. Figure 10 shows the Packet Parser model implemented on the switch Datapath. Each module is described as follows:

- **NetPDL**. The Network Protocol Description Language (NetPDL)(RISSO; BALDI, 2006) is the packet description language. It is a XML-based language and has a large number of protocols and specified encapsulations. In addition, the simple language definition allows easy and fast addition of non available protocol description. An example of how the UDP protocol is described using NetPDL can be foung in the Annex B. In the Figure 10 the NetPDL module feeds the parsing library with the description of the OpenFlow 1.3 supported match fields.
- **NetBee library Parser**. Netbee is as library for packet processing (NBEE, 2012). It is composed by several modules for different types of network application, such as packet filtering and sniffing. For our Packet Parser implementation, we use the Netbee library decoding objects. These objects come from a C++ set of classes and methods that

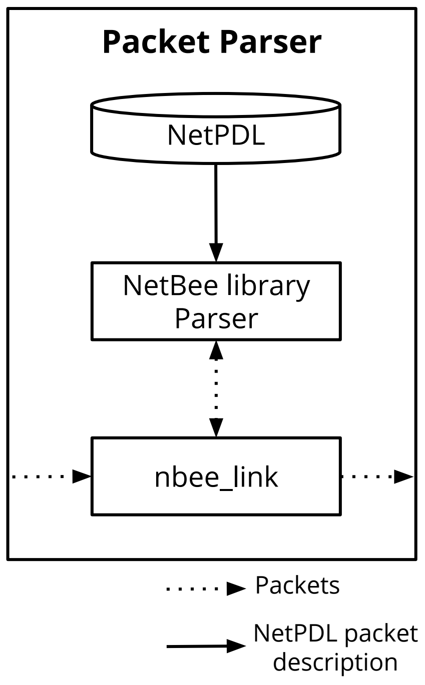


Figure 10 – Packet Parser components

ease packet decoding. To accomplish this, firstly Netbee loads the NetPDL protocols specification into the machine Random Access memory (RAM) and on a packet. Then, received packets are decoded according to the NetPDL description and the extracted information is stored in a protocol tree. Finally, packet field values can be retrieved from the tree using specific methods of the library.

- **nbee_link**. This module is where packets are converted in the flow match structure. Arriving packets are sent for the Netbee library for decoding. From the protocols tree generated by Netbee, the nbee_link module extracts the field values and builds the packet match structure that will be sent to the Flow Table look up. The code to extract a protocol is shown by Listing 4.3. Using the Netbee method *GetPDMLField*, we get the three ethernet protocol supported fields in OpenFlow 1.3. The second argument of *GetPDMLField* reflects the field name defined in the NetPDL specification. The function *nblink_extract_proto_fields* receives the extracted field value and type and inserts this into the match structure. Another important piece of code is present in the third line. For possible further processing, for instance, the application of a *set field* action, a reference to the protocol position needs to be stored.

```
if (protocol_Name.compare("ethernet") == 0 && pkt_proto->eth == NULL)
{
    pkt_proto->eth = (struct eth_header *) ( (uint8_t*) pktin->data +
         proto->Position);
    PDMLReader->GetPDMLField(proto->Name, (char*) "dst", proto->
        FirstField, &field);
    nblink_extract_proto_fields(pktin, field, pktout, OXM_OF_ETH_DST)
        ;
    PDMLReader->GetPDMLField(proto->Name, (char*) "src", proto->
        FirstField, &field);
    nblink_extract_proto_fields(pktin, field, pktout, OXM_OF_ETH_SRC)
        ;
    PDMLReader->GetPDMLField(proto->Name, (char*) "type", proto->
        FirstField, &field);
    nblink_extract_proto_fields(pktin, field, pktout, OXM_OF_ETH_TYPE
        );
}
```

Listing 4.3 – Ethernet parsing in the nbee_link module

An example of how helpful is a flexible design for the Packer Parser is on the support for IPv6 Extension Headers (EH) (DEERING; HINDEN, 1998). EHs parsing execution is not a trivial task, as there are different types and formats. What is more, IPv6 packets may present complex combinations of headers. In OpenFlow 1.3 support for IPv6 EHs is not based on values, but on a special bitmap that matches in the presence of EHs. Besides, a bit field matches an IPv6 packet only if their EHs are in the recommended order. All of these details would result in a large ammount of code to parse EHs correctly. However, this is done in few lines due to our extensible implementation and the NetPDL language.

Flow Match Prerequisites

Another change brought by OXMs is the introduction of flow match fields prerequisites. In order to obtain flow match consistency, some match fields require the presence of other fields. For example, matching any ARP protocol field requires the ethertype field having the correct value for an ARP packet. Thereby, inconsistent flows are denied by the Flow Table.

To map OXM fields prerequisites, a file [2], with several C Preprocessor macros, was created. The macros map each field with their respective network layer 2, layer 3 or upper

[2] This file was inpired by the old way that OVS handled the Nicira Extended Match (NXM) format. NXM is the format that gave origin to OXM.

level requisite. In addition there is a field that tells if a field is maskable or not. Listing 4.4 shows prerequisites and fields macros definition. Also, it gives an example of a field created by the *DEFINE_FIELD* macro.

```

#define OXM_DL_NONE     (0, 0)
#define OXM_DL_ARP      (ETH_TYPE_ARP, 0)
#define OXM_DL_PBB      (ETH_TYPE_PBB,0)
#define OXM_DL_IP       (ETH_TYPE_IP, 0)
#define OXM_DL_MPLS     (ETH_TYPE_MPLS, ETH_TYPE_MPLS_MCAST)
#define OXM_DL_IPV6     (ETH_TYPE_IPV6, 0)
#define OXM_DL_IP_ANY   (ETH_TYPE_IP, ETH_TYPE_IPV6)

#define DEFINE_FIELD_M(HEADER, DL_TYPES, NW_PROTO, MASKABLE) \
    DEFINE_FIELD(HEADER, DL_TYPES, NW_PROTO, MASKABLE)       \
    DEFINE_FIELD(HEADER##_W, DL_TYPES, NW_PROTO, true)

DEFINE_FIELD    (OF_TCP_SRC,      OXM_DL_IP_ANY,   IPPROTO_TCP,    false)
```

Listing 4.4 – Ethernet parsing in the nbee_link module

OXMs matches definitions are loaded by the Oflib, and used in the function *oxm_pull_match*, which is called during the match unpack. Among the tests performed to detect invalid OXM fields are: bad prerequisite, duplicate fields, wrongly masked and nonexistent field.

Flow Matching

In the pursuit for the best way to perform flow matching inside the Flow Table, developers might want to try different algorithms and data structures. For this reason, the switch implements a flexible and easy interface to change the way packets are matched.

Match fields are part of the software switch *flow_entry* struct. Instead of defining a fixed match as one of the *flow_entry* member, a pointer to Oflib *struct ofl_match_header* is left as a reference for the entry match fields. Therefore, if a developer wants to experiment his own match structure, there is only the need to make it start with an *ofl_match_header*.

This work presents a default flow matching using the Oflib match structure called *ofl_match*. Besides the match header, the struct includes a Hash Map structure to store OXM TLVs. Each OXM entry in the Hash Map has an exclusive key, created by the combination of the field Type and Length information. Storing only flow specified fields saves memory space, at a small cost of the pointers created to mantain the data structure. Another advantage in the Hash Map use in the match structure is the constant access time for the OXMs. Fast element access is very important for two of the most common operations:

- **Check packet matching**. Packet fields are extracted and matched against the flows. Matching is performed by look ups of the packet fields in the Hash Map.
- **Check flow collision**. Flows collide when a new flow is installed and the Flow Table contains a flow with the same match fields and priority. In this case the old one is replaced by the new one. The Hash Map allows a direct comparison of fields.

Another detail about flow matching in the software switch is about the linear behavior of Flow Table look up. The Flow Table stores flows in a list ordered by priority. When a packet is sent to the flow match it loops through the flow list until it finds a matching rule or it reaches its end. This is the most simple approach for the flow match and was chosen for its simplicity. Developers who might want to modify the behaviour of Flow Table look up just need to add their own code for the function *flow_table_lookup*.

Extensible context expression in 'packet-in'

Former *packet-in* message contained little information about the packet parsed in the Datapath. The only match field present was the switch input port. In order to get the other packet fields, a controller needs to parse the packet header, included in the end of *packet-in*. This causes an unnecessary parsing repetition in the control plane. With the OXM introduction, OpenFlow 1.3 solves this problem sending the extracted packet fields in the form of OXMs, making it easier for the control plane to retrieve the packet fields.

While an standard switch implementation requires only context information, which are input port, metadata and tunnel_id, our implementation follows the option to add all parsed fields in a *packet-in* message.

Set Field action

Support for rewriting packet fields exists since the first OpenFlow version. However, it was limited to a small set of fields. In OpenFlow 1.3, with the OXM introduction, a *flow mod* message can carry a *set field* action with any of the OXMs defined by the specification. It is up for the switch designers to decide which fields are allowed for overwrite.

Implementation of *set field* is slightly intricate, as the consistence is achieved through the match fields. For instance, a flow with a *set field* action to rewrite the IP source address needs to present in the match fields the same ethertype - 0x800 in hexadecimal - of the IP protocol. The way the pack and unpack of match fields and actions is performed by different functions needs to be checked in the Datapath. When handling a new *flow mod* message, the Flow Table calls the function *dp_actions_check_set_field_req*. This function uses an Oflib function to check if the prerequisites are ok and validates the action.

Another frequent task caused by rewriting fields is protocol CheckSum recalculation. Fields like the IP source and destination, in the case of change, require recalculation of IP and TCP CheckSum values. Fortunately these protocols' CheckSum calculation is very simple (BRADEN *et al.*, 1988). This is not the case for the SCTP protocol (STONE *et al.*, 2002). SCTP CheckSum is calculated using a Cyclic Redundancy Check (CRC). In order to recalculate the SCTP CheckSum value we used a Python program named pycrc [3]. The program takes as input the CRC polynomial and generates all the functions necessary for the calculations. Listings 4.5 shows the code to rewrite the SCTP destination port. In the packet field rewriting we attribute a pointer to the protocol struct representation and to the packet position obtained by the Packet Parser. Doing so, we can easily change the current value of the action value.

```
case OXM_OF_SCTP_DST:{
                crc_t crc;
                struct sctp_header *sctp = pkt->handle_std->proto->sctp;
                size_t len = ((uint8_t*) ofpbuf_tail(pkt->handle_std->pkt->
                    buffer)) - (uint8_t *) sctp;
                uint16_t v = htons(*(uint16_t*) act->field->value);
                sctp->sctp_csum = 0;
                memcpy(&sctp->sctp_dst, &v, OXM_LENGTH(act->field->header));
                crc = crc_init();
                crc = crc_update(crc, (unsigned char*)sctp, len);
                crc = crc_finalize(crc);
                sctp->sctp_csum = crc;
                break;
            }
```

Listing 4.5 – Ethernet parsing in the nbee_link module

Per-flow Metering

The Meter Table implementation follows the architectural details and responsibilities of the element described on section **??**. Firstly we defined a structure for the Meter Table. The

[3] pycrc v0.8.2, Available at http://www.tty1.net/pycrc/

main components are the table features, such as the max number of entries and supported band types, and a Hash Map of meter entries. Other members include a reference pointer to the Datapath, allowing a Meter Table to call the function to send OpenFlow messages; and two counters: one for the number of meter entries and another one for the quantity of bands. Secondly, we implemented a set of functions: initialization and destruction of the Meter Table; *meter mod* and *meter features* messages handlers; find and apply a meter entry.

Structures of meter entries are composed of a configuration - which contains information about the meter id and meter bands - and a struct for recording statistics. In addition, the meter entry has pointers to the Datapath and the Meter Table, similar to what is done in the Meter Table struct. Finally, it has a list of flow references. If the meter entry is deleted, all flows sending packets to the meter entry are deleted.

Meter entry bands are chosen accordingly to a configured rate - in Kilo packets (Kpps) per second or Kilobits per second (Kbps). Thus, it is necessary to measure the flow matched packets in function of one of the specified unities. The first idea to implement rate measuring scheme considered the use of matched flow counters, divided by the number of matched bytes by some time interval. Although easy to implement, this approach proved itself inaccurate after some attempts to limit the bandwidth between two hosts connected to the switch.

After a better analysis of the task and some literature research, we found and implemented a simple and efficient algorithm used for rate policy: the Token Bucket (TANENBAUM, 2002). Figure 11 illustrates how the Token Bucket works within a meter band. Simply put, each meter band has a bucket attached to it. At every second the bucket is refilled with a number of tokens equal to the meter rate. When a packet is sent to the Meter Table, it goes through each band's bucket belonging to the meter entry. Inside the bucket, packets consume a number of tokens equal to their size. If there are enough tokens, the OpenFlow pipeline continues processing the packet, otherwise, the meter band is chosen and executed.

Connection Features

Network control protocols must be designed with scalability and high availability in mind. Node failures and high traffic loads may cause frustration for early adopters of new technologies, as these two important points are not usually considered by initial versions. Previous OpenFlow versions fall into this category of protocols, often criticized by the lack of mechanisms to handle control plane issues.

More recent OpenFlow versions try to address control plane scalability and high availability with the addition of new features for OpenFlow connections. Auxiliary connections

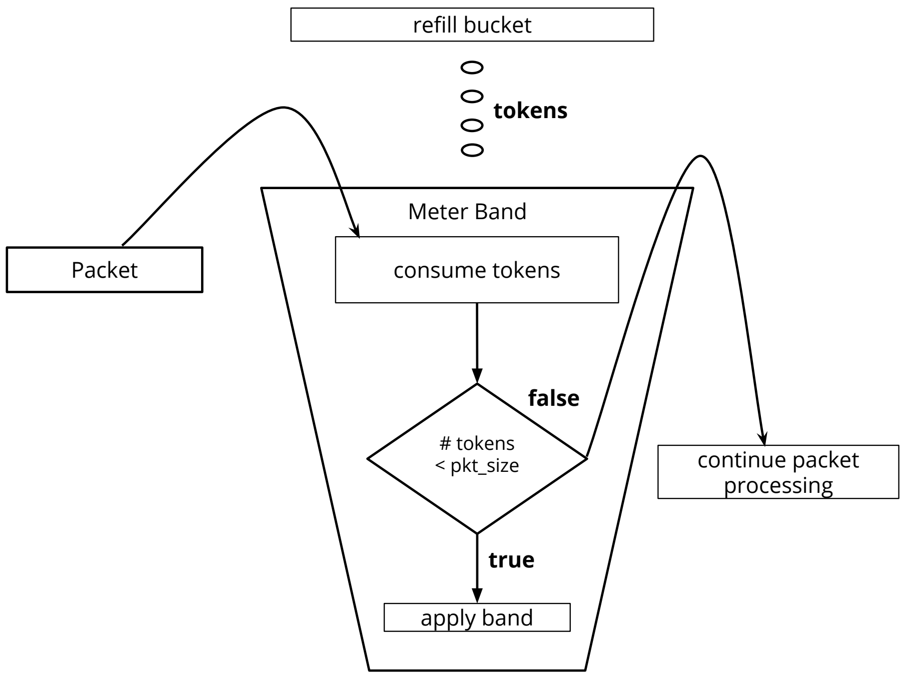


Figure 11 – Token Bucket Algorithm illustration inside the meter band

allow higher scalability for message exchanging, while controller roles try to enable fast failover for OpenFlow controllers. Event filtering, in turn, may be seen as a mechanism that sum up on these two topics. In the next subsections we will see a more detailed description of each feature and how they have been implemented in our software switch.

Auxiliary Connections

Auxiliary connections allow a controller to create more than one Communication Channel with a single switch. These connections add the possibility to exploit message parallelism and create a channel for specific types of messages. For instance, a controller can use one connection only for *packet-in* messages.

As a proof-of-concept we have implemented basic support for auxiliary connections. In our implementation, there is support for only one additional channel and it only carries *packet-in* messages. The following items show steps added in the switch code to handle auxiliary connection.

- The software switch sends OpenFlow messages for the Communication Channel, encapsulated into a struct called *ofbuf*. The struct is a buffer that holds information such as the pointer for the first allocated byte and the size of bytes in use. In order to identify which connection is being used to receive or send the message, we have added a new member called *conn_id*. The possible values for *conn_id* are MAIN_CONNECTION and PTIN_CONNECTION.

- A new connection listener was added to the Datapath. If an auxiliary connection is specified when running the datapath, the auxiliary listener is opened after the main connection listener.
- When the Datapath talks to remotes, it searches for the auxiliary connection. If the connection exists, it processes messages received by the connection.
- On sending OpenFlow messages, the switch by default maps to the MAIN_CONNECTION. If the message is a reply from a sent message of a sender connection, the connection id is set to the same id used by the sender. In the last case, if the message type is a *packet-in*, the switch uses PTIN_CONNECTION for the connection id.

The start of an auxiliary connection from one controller is disabled by default in our software switch standard program execution. To enable auxiliary connections a user should specify the *multiconn* option in the command line option.

Controller Role

Controller Role is a mechanism to permit connection of multiple controllers with different duties. One of the possible use cases of roles is for fast failover, in which when the main controller goes down, a backup controller assumes the switch command. There are three possible roles for controllers: Master, Slave and Equal. A master controller has permissions to send and receive any type of OpenFlow messages. Slaves have very strict default permissions, allowed only to receive a specific set of messages. The last role, Equal, is the default role when a controller connects to the switch and the other controllers connected do not have a defined role.

Role election is totally driven by the control plane, though some additional code is required for the switch. In order to implement controller role support in our software switch we first filtered asynchronous messages received by Slave controllers. Then we restricted slaves to send only read state messages, for example, *flow stats* and *table stats*. The last insertion is the algorithm defined by the specification to handle the *role request* generation_id. Messages with a generation id smaller than previous generation ids seen by the switch are discarded. Listing 4.6 presents the function that implements the algorithm.

```
static ofl_err
dp_check_generation_id(struct datapath *dp, uint64_t new_gen_id){

    if(dp->generation_id >= 0  && ((uint64_t)(new_gen_id - dp->generation_id)
        < 0) )
        return ofl_error(OFPET_ROLE_REQUEST_FAILED, OFPRRFC_STALE);
    else dp->generation_id = new_gen_id;
    return 0;
}
```

Listing 4.6 – Ethernet parsing in the nbee_link module

Event Filtering

Event Filtering enables controllers to filter undesired asynchronous messages, sent by the switch. Filtering of asynchronous messages is possible for three types: *port status*, *packet-in* and *flow removed*. In addition, a controller can also choose to not receive these message types for the generation reason. For example, a *packet-in* can be generated by an action to output the packet for the controller. This feature, along with auxiliary connections, gives power for controllers to create exclusive message channels.

Message filtering is handled by the Datapath. On a *set async request*, the Datapath sets the controller remote channel with bitmap values sent in the message defined by the OpenFlow 1.3 specification, shown on Listing 4.7. Each bit set in the bitmap represents a message type and a reason. For instance, a bit with value 4 in *flow_removed_mask[0]*, determines if the controller will receive *flow removed messages* with reason OFPRR_DELETE when the role is Master.

Filtering happens before the sending of an OpenFlow message. The Datapath function to send an OpenFlow buffer through the Communication Channel checks the remote configuration and the type of message to be sent. If it is one of the three asynchronous messages and the reason and the controller roles matches the remote filtering configuration, the message is dropped.

```

/* Asynchronous message configuration. */
struct ofp_async_config {
    struct ofp_header header;
    /* OFPT_GET_ASYNC_REPLY or OFPT_SET_ASYNC. */
    uint32_t packet_in_mask[2];
    /* Bitmasks of OFPR_* values. */
    uint32_t port_status_mask[2]; /* Bitmasks of OFPPR_* values. */
    uint32_t flow_removed_mask[2];/* Bitmasks of OFPRR_* values. */
};
OFP_ASSERT(sizeof(struct ofp_async_config) == 32);
```

Listing 4.7 – Ethernet parsing in the nbee_link module

5 Evaluation

In this chapter we evaluate our work in terms of the requisites presented on section 1.1. The first section shows in numbers how many features are covered by the software switch. Subsequently, in the next section, we present the results of performance benchmarks tests. The last section of the chapter is a qualitative evaluation about the code's ease to change. We demonstrate the code portability, highlighting the port of the software switch to another processor architecture in a different operating system.

The software switch evaluated version dates from the last commit pushed to GitHub. The box below shows the dates and last code changes description.

- **commit** cb740bd2565ac7e5d61ebe30ee75160a5452a033
- **Commit:** Eder Leão Fernandes <ederleaofernandes@gmail.com>
- **CommitDate:** Mon Feb 23 18:42:49 2015 -0300

  Add flags member to ofp_flow_stats.

  Fix missing flags field in the response of a flow stats request.

Feature Completeness

Evaluating the proper operation of the OpenFlow switch features is not a trivial task. This is caused by the multiple and rich configurations allowed by the specification. For example, testing all flow match fields combinations would require creation of a large number of flows and packets, making manual tests very time consuming. For this reason, automatic test frameworks, discussed on section **??**, are the best options to test the switch functionality in order to evaluate feature completeness.

OFTest and Ryu Certification are the two test frameworks used for the switch validation. As mentioned in chapter **??**, both are important tools for the software switch development. While Ryu certification has a strong focus on validation of the Datapath, OFTest offers a nice set of test cases for control and data plane message exchange. In the next sections we present a resume of the results obtained.

OFTest results

Testing in OFTest is simple as it provides scripts in Python to run the switch and the test cases. Each test case starts a controller which connects with a running switch, executes the test instructions and checks the switch answers.

Some messages from controller to switch, like a *flow stats* request, and symmetric messages demand an answer from the switch. Thus, the main purpose of the framework usage with the software switch is for message handling validation. Although OFTest has capabilities to evaluate the pipeline processing - for instance, checking if a packet was correctly forwarded by a flow - we found in Ryu a more comprehensive test set for this task.

Table 1 shows test results for basic OpenFlow messages. The major type of messages of the test set are messages to query information about the state of manifold switch elements, such as *GroupFeatureStats* and *MeterStats*. Also, there are some configuration messages, like the *PortConfigMod*. In all tests the switch returned the right answer for the control plane.

Table 1 – Basic OpenFlow messages

Message	Result
AggregateStats	ok
AsyncConfigGet	ok
DescStats	ok
Echo	ok
EchoWithData	ok
FeaturesRequest	ok
FlowStats	ok
FlowRemoveAll	ok
GroupDescStats	ok
GroupFeaturesStats	ok
GroupStats	ok
MeterConfigStats	ok
MeterFeaturesStats	ok
MeterStats	ok
QueueStats	ok
PortConfigMod	ok
PortDescStats	ok
TableStats	ok

Controller roles test results are shown in table 2. These tests check if the software switch changes correctly controller roles, and if the respective permission police is respected. As in the previous message tests, all role tests were successful.

Ryu Certification results

The Ryu Certification tests are divided into five categories: Action, Set-Field, Match, Group and Meter. The test sets of each category are very comprehensive, with tests for different packet types.

Table 2 – Role request message results

Role Request Tests	**Results**
RoleRequestEqualToSlave	ok
RoleRequestSlaveToMaster	ok
RolePermissions	ok
RoleRequestEqualToMaster	ok
RoleRequestNochange	ok
SlaveNoPacketIn	ok

Table 3 is a resume of test results - the complete list of test cases can be found on Annex **??** - for this work compared to the other three switches presented on chapter 2. White cells give the number of tests passed, while grey cells show the number of test cases that returned an error. Tests are divided by each category, with the last two columns giving the total sum of working and non working features. The first row presents the results for this work. [1]

Table 3 – Ryu Certification results comparison

Switch	**Action**		**Set-Field**		**Match**		**Group**		**Meter**		**Total**	
ofsoftswitch13	50	6	159	7	708	6	15	0	30	6	848	25
Open vSwitch	34	22	96	74	534	180	0	36	0	36	670	321
LINC	24	32	68	102	428	286	3	12	0	24	523	456
Trema	50	6	159	11	708	6	15	0	34	2	852	25

Results show that the software switch has a higher number of working features than Open vSwitch and LINC. With only 25 errors, it is tied with Trema in the number of supported features. There is a small difference between ofsoftswitch13 and Trema in the total number of tests passing. This happens because Ryu Certification does not execute four tests due to old switch restrictions.

The values presented in this section are from the official certification site. Some failed test results are presented on the site work in our internal test setup. For instance, matching on *PBB ISID* value works as expected when tested in our development machine. However, we chose to show the official results as we could not identify the reasons for different results. In addition, some test may never pass. For example, *IP proto* modification causes packet malformation, because the IP proto packet field will not conform with the next layer protocol, which leads to a test failure.

[1] ofsoftswitch13 is the software switch repository name

Performance Benchmarks

One of the software switch requirements listed on chapter 1 is to reach a maximum throughput of at least 100 Mb/s. For this reason we evaluated the switch performance in terms of network metrics. In this section we show how the switch performs for different packet sizes in comparison with the userspace switches LINC and Trema. We do not compare with OVS, since it is a software switch for production networks. Also, we investigated how performance is affected by the number of flows and by the number of tables traversed to match a packet.

The machine configuration used to perform measurement tests are listed in the box below.

- **Processor**: 8x Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz
- **Memory**: 6003MB
- **Operating**: System Ubuntu 14.04.2 LTS

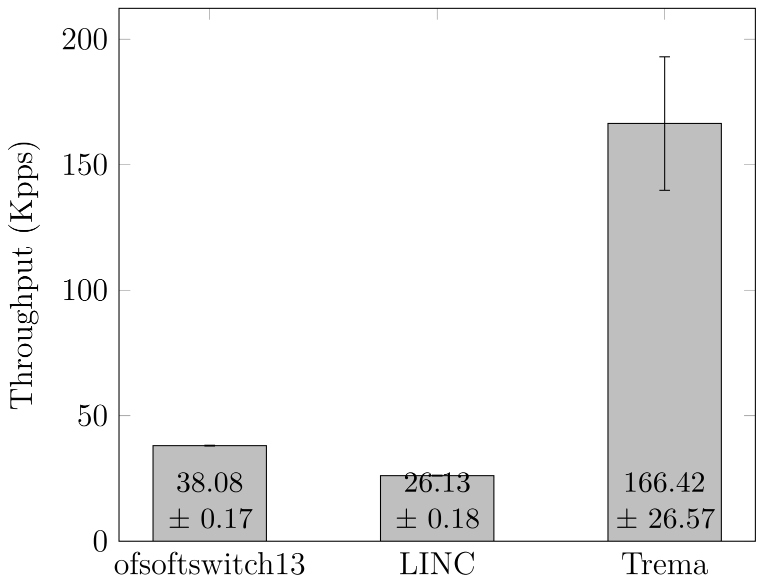
Maximum Throughput

This test evaluates the maximum forwarding rate the software switch can reach in comparison to other userspace implementations.

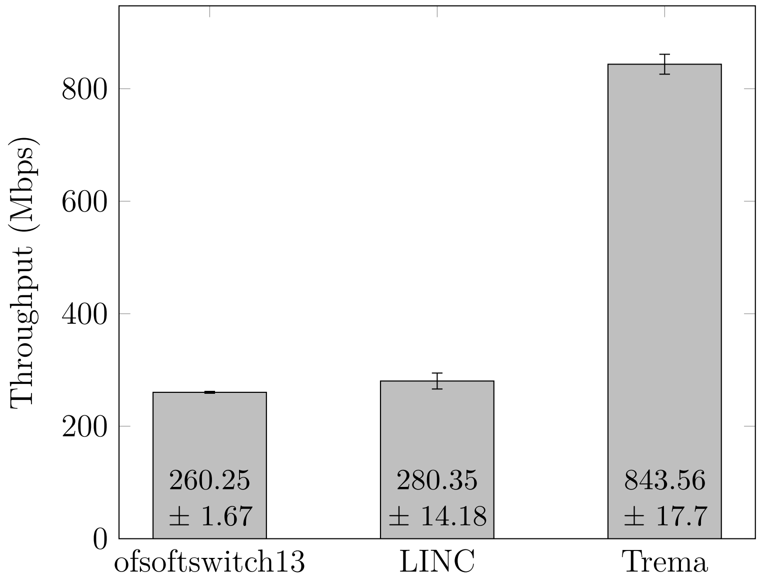
The setup for maximum throughput evaluation is the following:

- A running instance of the software switch with two virtual interfaces - Port 1 and Port 2 - attached.
- One flow installed in the Flow Table to match a packet sent to Port 1 with destiny ethernet 00:00:00:00:01. The action is an output packet to Port 2.
- A packet traffic generator. We used a simple program named packeth (ISKRATEL, 2003).
- A script running in the Linux terminal checking the current packet rate on Port 2.

The test starts by installing the flow in the switch Flow Table. Afterwards, we inject packets, using the traffic generator, directly into Port 1. The bandwidth results of Port 2, reported by the script, are used to calculate the average rate and the standard deviation. Two transmission measurements were made: for small packets of 64Kb and bigger packets of 1500Kb. Figure 12 shows results for both experiments.



(a) Comparison using packets of 64 bytes



(b) Comparison using packets of 1500 bytes

Figure 12 – User space software switches throughput comparison

Switch forwarding performance for small packets is evaluated in Kilo packets per second (Kpps). Figure 12a shows that ofsoftswitch13 can handle 38.08 Kpps. This result is very far from Trema and approximatelly 32% more efficient than LINC. Bigger packets are measured in Megabits per second. Results presented in Figure 12b show that ofsoftswitch13 and LINC, with rates of 260.25 Mbps 280.35 Mbps respectively, are slower than Trema with a bandwidth of 843.56 Mbps.

Although Trema overcomes our work in throughput performance, due to optimizations like multiple threads, the most important result of this experiment is the software switch maximum throughput found. This value is higher than the value established by the software switch requirements.

Throughput in function of flows and tables

This experiment measured two factors that may affect the software switch performance. One is the number of flows installed in one table. The second is the number of tables traversed until the match is found.

For this test we used Mininet with the software switch connecting two hosts. Bandwidth is measured through the Iperf session established between the two hosts. Flow Table setup for the two cases are the following:

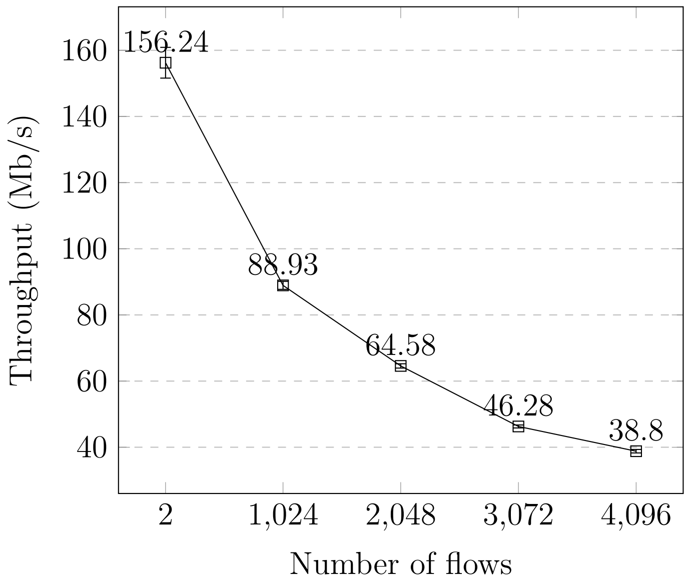
(A) **Number of flows**. A certain amount of flows, with the same priority, that will not match packets is installed. In the end, two flows, with the same or lesser priority than the previous, are installed to forward packets between the two hosts.

(B) **Number of tables**. Flows to send the packet to next table are installed until the penultimate table. Then, in the last table two flows are added to forward the traffic between the hosts.

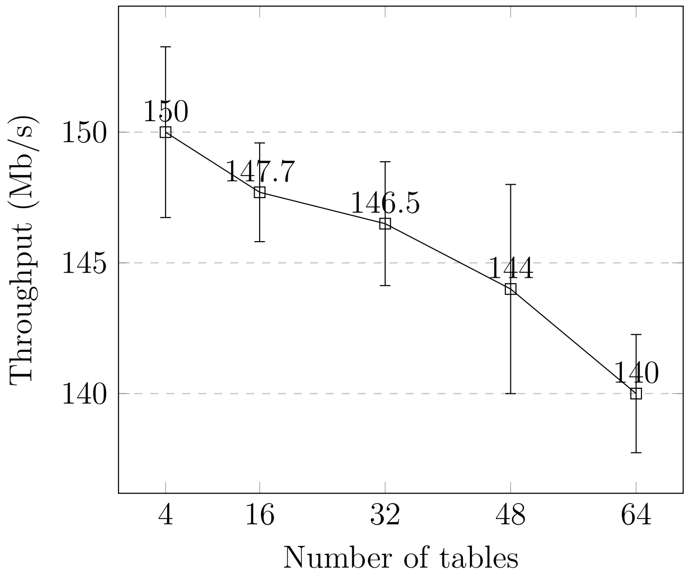
The graphs in the Figure 13 shows that both cases have a strong influence over the switch performance. The most sensitive case is for one table shown in Figure 13a, as the number of flows increases the throughput decreases linearly. The increase in the number of tables, shown by the graph in the Figure 13b, also causes a linear decrease in the packet rate, though it is smaller than in the first case. These results were expected, since the software switch implements linear matching. Thus, this experiments were important to verify one improvement area for the software switch.

Ping Round Trip Time

Round Trip Time (RTT) is the time between a data request and answer. Several factors might affect the total RTT and influence network's latency. Two examples are: the



(a) Throughput per number of flows in one table



(b) Throughput per number of tables

Figure 13 – Influence of the number of installed flows on the throughput.

number of nodes between two communicating hosts and the transmission medium. The time a packet takes to enter and leave a switch is also considered for the RTT. Thus, it is important to measure how much the software switch affects the RTT.

In order to measure the RTT between two hosts connected by our software switch - we also compare LINC and Trema -, the following steps are executed:

1. Creation of two Linux containers (LXC) - Host 1 and Host 2 - with a pair of virtual interfaces *veth0* and *veth1*. LXC is an operating system lightweight virtualization

technology, in which it is possible to run multiple isolated Linux instances as containers. With LXC, we run two containers to serve as the network hosts.

2. Execution of a software switch instance with the container virtual ports attached to switch interfaces.
3. Installation of two flows in the switch Flow Table to forward the traffic between the two hosts.
4. Configuration of Host 1 and Host 2 with IP addresses in the same network. In our test Host 1 is configured with the IP address 192.168.0.1 and host 2 as 192.168.0.2.
5. Execution of the *ping* program in Host 1 to ping the address 192.168.0.2. Ping is a program to send and measure the time between an Internet Control Message Protocol (ICMP) "Echo request" and the ICMP "Echo Reply". The number of Echo requests sent is 100 and the packet sizes are 64Kb.

Switch results comparison is shown in Table 4. These tests give a good approximation for the software switch impact over the network delay, because it is connected directly to the hosts. As expected, because of the previous results, Trema is the most efficient among the userspace software switches. The ofsoftswitch13 obtains a low minimum RTT, with 0.304 ms, compared to the average of approximately 1ms. LINC has a very high RTT, with more than a half second to complete. This is not a surprise, because the throughput tests, shown in section 5.2.1, revealed that LINC does not handle small packets efficiently.

An acceptable RTT value depends on the application running over the network. Latency sensitive programs, like multiplayer online games, benefit from a low RTT. Considering a small network, with not many hops, the RTT in our software switch is acceptable.

Table 4 – Ping Round Trip Time comparison between software switches

Software Switch	**Minimum**	**Average**	**Maximum**	**Standard Deviation**
ofsoftswitch13	0.30	1.07	1.82	0.31
LINC	303.90	554.77	821.48	253.03
Trema	0.12	0.40	0.48	0.04

6 Conclusion

... *In the next two sections, we present some obtained results and notorious use cases. Finally, we conclude this chapter discussing future areas for research and improvement in the software switch.

Results

*In this section we list positive results achieved on the dissemination of our work:

- **Publications.**

Use Cases

- **Base for new Gateway features implementation.**
- **Academic.** The software switch has found good adoption by the academic community.
- **Industry.** Industrial development is harder to track because it is usually closed. However, one successful case is in the development of an application for ISP or Telco enterprise like Ericsson.

Future Work

Bibliography

ALCATEL-LUCENT. Evolution of the Broadband Network Gateway. 2010. Cited on page 5.

ARORA, D. *Proactive Routing in Scalable Data Centers with PARIS*. 2013. Not cited on the text.

BALLANI, H.; COSTA, P.; KARAGIANNIS, T.; ROWSTRON, A. Towards predictable datacenter networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 41, n. 4, p. 242–253, ago. 2011. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2043164.2018465>. Cited on page 13.

BIANCHI, G.; BONOLA, M.; CAPONE, A.; CASCONE, C. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 44, n. 2, p. 44–51, abr. 2014. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2602204.2602211>. Not cited on the text.

BRADEN, R.; BORMAN, D.; PARTRIDGE, C. *Computing the Internet checksum*. IETF, 1988. RFC 1071. (Request for Comments, 1071). Updated by RFC 1141. Disponível em: <http://www.ietf.org/rfc/rfc1071.txt>. Cited on page 25.

DEERING, S.; HINDEN, R. *Internet Protocol, Version 6 (IPv6) Specification*. IETF, 1998. RFC 2460 (Draft Standard). (Request for Comments, 2460). Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112. Disponível em: <http://www.ietf.org/rfc/rfc2460.txt>. Cited on page 22.

GIT. *git –distributed-even-if-your-workflow-isnt*. 2005. <http://git-scm.com/>. [accessed: 23-Fev-2015]. Not cited on the text.

HAN, B.; GOPALAKRISHNAN, V.; JI, L.; LEE, S. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, v. 53, n. 2, p. 90–97, Feb 2015. ISSN 0163-6804. Cited on page 1.

ISKRATEL, M. J. *packeth - Ethernet packet generator*. 2003. [accessed: 01-Mar-2015]. Disponível em: <http://packeth.sourceforge.net/packeth/Home.html>. Cited on page 34.

NAGAPPAN, N.; MAXIMILIEN, E. M.; BHAT, T.; WILLIAMS, L. Realizing quality improvement through test driven development: Results and experiences of four industrial teams. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 13, n. 3, p. 289–302, jun. 2008. ISSN 1382-3256. Disponível em: <http://dx.doi.org/10.1007/s10664-008-9062-z>. Not cited on the text.

NBEE. *NetBee Library*. 2012. <http://www.nbee.org/>. [accessed: 11-Nov-2014]. Cited on page 20.

NEMETH, B.; SIMONART, X.; OLIVER, N.; LAMOTTE, W. The limits of architectural abstraction in network function virtualization. p. 633–639, May 2015. ISSN 1573-0077. Cited on page 2.

ONOS. *ONOS Segment Routing*. 2014. <https://wiki.onosproject.org/display/ONOS/Installation+Guide>. [accessed: 03-Mar-2015]. Not cited on the text.

OpenWRT. 2004. Disponível em: <http://www.openwrt.org>. Not cited on the text.

P, B.; D, D.; M, I.; MCKEOWN, N.; REXFORD, J.; SCHLESINGER, C.; TALAYCO, D.; VAHDAT, A.; VARGHESE, G.; W, D. Programming Protocol-Independent Packet Processors. 2013. ISSN 01464833. Disponível em: <http://arxiv.org/abs/1312.1719>. Cited 3 times on page(s) xvii, 6, and 7.

PATRA, P.; ROTHENBERG, C.; PONGRACZ, G. MACSAD: High performance dataplane applications on the move. *IEEE International Conference on High Performance Switching and Routing, HPSR*, v. 2017-June, 2017. ISSN 23255609. Cited 2 times on page(s) xvii and 7.

POPA, L.; YALAGANDULA, P.; BANERJEE, S.; MOGUL, J.; TURNER, Y.; SANTOS, J. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. v. 43, p. 351–362, 08 2013. Cited on page 13.

RADHAKRISHNAN, S.; GENG, Y.; JEYAKUMAR, V.; KABBANI, A.; PORTER, G.; VAHDAT, A. Senic: Scalable nic for end-host rate limiting. USENIX Association, Berkeley, CA, USA, p. 475–488, 2014. Disponível em: <http://dl.acm.org/citation.cfm?id=2616448.2616492>. Cited on page 14.

REITBLATT, M.; CANINI, M.; GUHA, A.; FOSTER, N. Fattire: Declarative fault tolerance for software-defined networks. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. New York, NY, USA: ACM, 2013. (HotSDN '13), p. 109–114. ISBN 978-1-4503-2178-5. Disponível em: <http://doi.acm.org/10.1145/2491185.2491187>. Not cited on the text.

RISSO, F.; BALDI, M. Netpdl: An extensible xml-based language for packet header description. *Comput. Netw.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 50, n. 5, p. 688–706, abr. 2006. ISSN 1389-1286. Disponível em: <http://dx.doi.org/10.1016/j.comnet.2005.05.029>. Cited on page 20.

ROBERTO, B.; THOMAS, D.; H, F.; A, M.; M, J.; N, S.; K, H.-J. Rethinking Access Networks with High Performance Virtual Software BRASes. *EWSDN*, 2013. Cited on page 6.

RODRIGUES, H.; SANTOS, J. R.; TURNER, Y.; SOARES, P.; GUEDES, D. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. USENIX Association, Berkeley, CA, USA, p. 6–6, 2011. Disponível em: <http://dl.acm.org/citation.cfm?id=2001555.2001561>. Cited on page 13.

RUPARELIA, N. B. Software development lifecycle models. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 35, n. 3, p. 8–13, maio 2010. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/1764810.1764814>. Not cited on the text.

SHOURMASTI, K. S. *Stochastic Switching Using OpenFlow*. [S.l.]: Institutt for telematikk, 2013. 70 p. Not cited on the text.

STONE, J.; STEWART, R.; OTIS, D. *Stream Control Transmission Protocol (SCTP) Checksum Change*. IETF, 2002. RFC 3309 (Proposed Standard). (Request for Comments, 3309). Obsoleted by RFC 4960. Disponível em: <http://www.ietf.org/rfc/rfc3309.txt>. Cited on page 25.

TANENBAUM, A. Computer networks. In: _____. 4th. ed. [S.l.]: Prentice Hall Professional Technical Reference, 2002. p. 401. ISBN 0130661023. Cited on page 26.

The Organization for Economic Co-operation and Development (OECD). Improving Networks and Services Through Convergence discussion paper. *2016 Ministerial Meetign*, 2016. Cited on page 1.

TOURRILHES, J. *OpenFlow prototyping*. 2013. <https://github.com/jean2/ofsoftswitch13>. [accessed: 11-Nov-2014]. Not cited on the text.

YIAKOUMIS, Y.; SCHULZ-ZANDER, J.; ZHU, J. *Pantou : OpenFlow 1.0 for OpenWRT*. 2011. Disponível em: <http://www.openflow.org/wk/index.php/OpenFlow_1.0_for_OpenWRT>. Not cited on the text.

Annex

ANNEX A – P4 code of BNG

ANNEX B – Publications

Three papers were published during this work and are listed below.

- Juan Sebastian Mejia, Christian Esteve Rothenberg. "Network Address Translation using a Programmable Dataplane Processor". In 7° Workshop em Desempenho de Sistemas Computacionais e de Comunicação (Wperformance) XXVIII Congresso da Sociedade Brasileira de Computação - CSBC 2018, Natal, 22 a 26 de Julho de 2018.