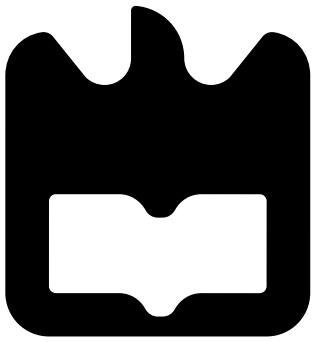
Alexandre Veiga
Almeida Santos

Redes Definidas por Software Flexíveis

Flexible Software-Defined Networks



Alexandre Veiga Almeida Santos

Redes Definidas por Software Flexíveis

Flexible Software-Defined Networks

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Professor Doutor Daniel Corujo, Professor do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor José Quevedo, investigador do Instituto de Telecomunicações

Esta dissertação foi realizada com o apoio do Instituto de Telecomunicações - Pólo de Aveiro, e com o apoio do projeto EC H2020 5GPPP 5Growthproject (Grant 856709).

To my parents. Who always ensured that nothing was missing

o júri / the jury

presidente / president

Professor Doutor Arnaldo Silva Rodrigues de Oliveira
Professor Auxiliar da Universidade de Aveiro

vogais / examiners committee

Doutor Flávio Silva Meneses
System Developer na Skyline Communications

Professor Doutor Daniel Nunes Corujo
Professor Auxiliar da Universidade de Aveiro

agradecimentos / acknowledgements

First I would like to express my honest thanks to my supervisors, Dr. Daniel Corujo and Dr. José Quevedo. The former, for giving me the opportunity to work in such a challenging problem and for the guidance throughout dissertation and the latter, for the guidance and work follow up almost every week, for sharing his knowledge and helping me to improve my own. I am also thankful to Instituto de Telecomunicações de Aveiro, especially ATNoG, for being such an important part in the shaping of my skills and therefore my future.

I also want to thank Angelo Corsaro and Luca Cominardi for all the help provided regarding Zenoh.

I want to thank to my family, the ones who were always there for me, but mostly my parents. The ones that gave me the opportunity to attend to University and especially the ones that gave me support, opened my eyes and did not let me give up when things got tougher.

I also want to thank my friends who made group works not look like work and for the trash talk conversations that really deepened our friendship.

Even though I know Espinafre will never read this, since he is my cute dwarf rabbit, I would like to express my gratitude for his presence, for being a good emotional support and such a good pet.

Finally and most important, I want to express my greatest thanks to Catarina, thank you for being such a lovely person and to be the one that always stood by my side, for the good and bad, and for the bad I mean the endless number of occasions were I was forced to go to McDonald's.

Thank you!

Palavras Chave

Redes Definidas por Software de Próxima Geração, P4, Plano de Dados Programável, Offloading, Zenoh

Resumo

As redes móveis de quinta geração (5G) conseguem oferecer melhores serviços que as suas anteriores gerações maioritariamente através do uso de tecnologias como redes definidas por software (SDN) e virtualização das funções da rede (NFV).

No entanto, após vários anos de implementações de soluções usando OpenFlow, chegou-se à conclusão que este tem limitações relativamente a protocolos desconhecidos, mesmo após vários anos da primeira versão. Então, a comunidade juntou-se e criou o que hoje é o ecossistema P4/P4Runtime. Sendo o P4 uma linguagem destinada à programação do comportamento do plano de dados e o P4Runtime o equivalente ao OpenFlow para equipamentos que suportam P4, no entanto permite uma gestão do comportamento do plano de dados independente do dispositivo e do protocolo, uma vez que não assume que os equipamentos de rede têm um comportamento fixo bem definido, normalmente descrito pelo chip ASIC.

Neste trabalho, faz-se uso do ecosistema do P4 para implementação de offloading de funções para os próprios equipamentos de rede e avalia-se se esta solução traz benefícios para a performance da rede. Devido à pouca exploração em P4 de sistemas publish-subscribe, que dependem tradicionalmente de brokers, foi decidido fazer offloading de funções de um desses sistemas através do uso de P4, permitindo ainda que a solução como um todo possa ser comparável com as oferecidas por um broker distríbuido. No entanto, o P4 tem limitações ao nível de gestão de sessões TCP. O Zenoh, um protocol publish-subscribe ainda em evolução e direcionado para IoT, permite também transporte sobre UDP, e é por isso um grande candidato a ser implementado em P4 para demonstrar as vantagens de fazer offloading de processamento para o plano de dados.

O sistema conceptualizado e desenvolvido foi então comparado com outros dois sistemas mais tradicionais que não fazem uso de offloading. Os resultados são animadores mostrando que existe benefício em fazer offloading de certas funções para o plano de dados, visto que certas operações podem ser feitas mais perto do utilizador final. No entanto, comparando os resultados com os oferecidos pelo Zenoh puro, os resultados são piores, sendo isto explicado pelo facto de os equipamentos de rede utilizados serem switches em software que não estão preparados para ambientes de produção e são muito penalizados por diversos fatores do comportamento do plano de dados. É por isso necessário fazer testes em equipamentos de hardware para uma avaliação mais profunda e consequente conclusão.

Keywords Next-Generation Software Defined Networks, P4, Programmable Data Plane, Offloading, Zenoh

Abstract

The fifth generation of mobile networks (5G) are able to offer better services than its predecessors mainly through the usage of software defined networks (SDN) and network functions virtualization (NFV)

However, after multiple solutions developed using OpenFlow, the conclusion was that the even after several years of the first version released, OpenFlow fails to offer full flexibility and cannot handle unknown protocols.With that in mind, the community got together and created what is known today as P4. P4 is a language designed to program the data plane behaviour, that, with the help of P4Runtime, the alternative of OpenFlow to P4 enabled devices, it allows the management of the data plane behaviour regarding the target or the protocol. All of that because, unlike OpenFlow, P4Runtime does not assume that network devices have a fixed and well defined behaviour, usually described by the ASIC chip.

In this work, P4 ecosystem is used to implement offloading of functions to the network devices and evaluate whether that is impactful for the network performance. Given the low amount of work developed with P4 regarding publish-subscribe systems, that traditionally rely on brokers, it was decided to offload several functions of such systems to the dataplane with P4, leading that the overall solution can be comparable to distributed broker ones. However P4 is limited regarding the management of state related data, just like of TCP sessions, which many publish-subscribe system rely on. Zenoh, a new publish-subscribe protocol that is still in early phases and directed to IoT, is also able to run over UDP and therefore is a great candidate to be implemented in P4 to overcome such issues. It is then used to show the advantages of doing offloading of processing to the dataplane.

The conceptualized system was then compared to two more traditional ones, that do not make use of offloading. The overall results achieved are promising. Results show that there are benefits in the offloading of certain tasks to the dataplane and therefore be closer to the end user and with that improve latency. However, regarding the pure Zenoh, the results achieved are poorer. That can be explained by the usage of software switches that are not production grade ready and whose performance is highly impacted by several data plane factors. That makes it necessary to do more tests on expensive hardware equipment for a more concrete conclusion.

Contents

List of Figures

List of Tables

Acronyms

3GPP Third Generation Partnership Project.

4G Fourth Generation of Mobile Networks.

5G Fifth Generation of Mobile Networks.

6G Sixth Generation of Mobile Networks.

API Application Programming Interface.

ARIB The Association of Radio Industries and Businesses.

ARP Address Resolution Protocol.

ASIC Application Specific Integrated Circuits.

ATIS The Alliance for Telecommunications Industry Solutions.

BGP Border Gateway Protocol.

BMv2 Behaviour Model version 2.

CAPEX Capital Expenditure.

CCSA China Communications Standards Association.

CD Continuous Delivery/Deployment.

CI Continuous Integration.

CLI Command Line Interface.

CPU Central Processing Unit.

DevOps Development and Operations.

DHCP Dynamic Host Configuration Protocol.

eMBB Enhanced Mobile Broadband.

ETSI European Telecommunications Standards Institute.

FPGA Field-Programmable Gate Array.

gNMI gRPC Network Management Interface.

gNOI gRPC Network Operations Interface.

GPU Graphics Processing Unit.

gRPC Google Remote Procedure Call (RPC).

GUI Graphical User Interface.

HD High Definition.

HTTP Hypertext Transfer Protocol.

ICMP Internet Control Message Protocol.

IoT Internet of Things.

IP Internet Protocol.

IPv4 Internet Protocol version 4.

IPv6 Internet Protocol version 6.

LLDP Link Layer Discovery Protocol.

M2M Machine to Machine.

mMTC massive Machine Type Communications.

MPLS Multi Protocol Label Switching.

NDN Named Data Networks.

NFV Network Functions Virtualization.

NFVI Network Functions Virtualization Infrastructure.

NG-SDN Next Generation - Software Defined Networks.

NIC Network Interface Controller.

ONF Open Networking Foundation.

ONL Open Network Linux.

ONOS Open Network Operating System.

OPEX Operational Expenditures.

OS Operating System.

OVS Open vSwitch.

OVSDB Open vSwitch Database.

PSA Portable Switch Architecture.

PTF Packet Test Framework.

QoE Quality of Experience.

QUIC Quick UDP Internet Connections.

RAM Random Access Memory.

RAT Radio Access Technology.

REST Representational State Transfer.

SAI Switch Abstraction Interface.

SD Standard Definition.

SDN Software Defined Network.

SNMP Simple Network Management Protocol.

SONiC Software for Open Networking in the Cloud.

SRAM Static Random Access Memory.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TSDSI Telecommunications Standards Development Society.

TTA Telecommunications Technology Association.

TTC Telecommunication Technology Committee.

UDP User Datagram Protocol.

UHD Ultra High Definition.

UI User Interface.

uRLLC Ultra-reliable and Low latency Communications.

VLAN Virtual Local Area Network.

VM Virtual Machine.

VNF Virtual Network Function.

VR Virtual Reality.

Chapter 1

Introduction

In the recent years, mainly due to the effort made by the 5G research community, there was a shift from the traditional network to more scalable and programmable ones. Solutions such as Software-Defined Networks (SDN) and Network Function Virtualisation (NFV) emerged. However, those solutions still lack fundamentals such as hardware and software independence, which essentially means that one is still stuck to specific hardware manufacturers and proprietary software tools, making it not truly independent as it was designed and conceptualized to be. This work aims to present a next generation solution that aims to fill the gaps presented in the previous solutions. This chapter starts to present the motivation behind the work done, followed by the statement of the problem and the objective of the work here presented. It ends with the overall structure of the document and the contributions of this work to the research community.

1.1 Motivation

In the early 60's, some brilliant minds in the United States decided that they could create a system to trade information on a real time basis, rather than using the widely used but limited circuit switching systems. That system would be based on packet switching which allowed to send more information. The experiments, led to the conclusion that computers sharing the same clock, could work together and even fetch information from remote machines, something that the telephone lines did not allow to do [1]. That system, whose main objective was to help engineers, was later named what we call Internet [2].

Today's Internet is much more than the original idea, however, the objective is the same, to keep people connected. Studies show that by the years, more and more equipments are being connected to the internet, in fact, in 2018 the number of devices connected were more than the planet's overall population [3]. By 2023 there are expected to be around 30 billion devices connected, with half of the connections being Machine-to-Machine (M2M) mainly due to the emerging number of internet of things (IoT) devices. The growth in the number of devices and connections will also bring traffic to the internet. Services such as streaming, gaming, etc, everything that need to transmit video across the network, are the most bandwidth usage services. The more

quality the videos have, the more demanding it will be to the network. The Standard Definition (SD) has a bit rate of 2 Mpbs, the High-Definition (HD) as a bit rate between 5 and 7 Mbps whereas Ultra-High-Definition (UHD) or 4K resolution has a bit rate between 15 and 18 Mbps, which is 9 times bigger than SD and 3 times the HD. In addition to that, 4K TV sets have a compound annual growth rate of 27% which means that by 2023, 66% of the TV sets will be 4K [3]. In a world where not only TV's but also security cameras, computer monitors and all other equipments supporting video, are starting to have 4K resolution, there is a need to improve the performance of the current network solutions.

This document addresses a possible solution that can be used to improve networks performance, which can be achieved through function offloading. It means that part or the totality of the functionalities of systems, are to be transferred and therefore executed on another equipment rather than the one where they are normally executed [4, 5]. This process is normally used to free limited resource devices, such as IoT devices, from hard processing tasks, but it can also be used to transfer that task to dedicated devices, as in this case. If tasks can be offloaded to network equipment, and therefore closer to the end user, then network links can be freed from a good amount of traffic. The current solutions started to free the network devices from heavy specific processing workload tasks, having them performed on a Virtual Machine (VM) or by containers in a cloud, which brings a lot of flexibility to scale the network components. However, due to high demand it could be beneficial to bring that processing back to the network devices, which can improve latency. Not to the same old and limited, but new, fast and fully programmable devices.

Traditional SDN heavily rely on OpenFlow, which is limited and makes users constrained to a set of very strict rules. That essentially means that at the dataplane level, programmable networks are forced to follow a strict set rules, not being able to use the full potential and therefore not being 100% programmable [6, 7]. That is, developers are only able to deploy what OpenFlow is able to process, which is not a problem for well know and established protocols, but a limitations when one wants to easily test new and custom protocols. This faulty behaviour can be overcame with P4, a protocol and target-independent packet processing language, that emerged as an alternative to OpenFlow to bring increased programmability and flexibility. The overall P4 ecosystem is designed to free the network managers from the top to bottom solutions offered by vendors, mainly relying on open source technologies. Such characteristic allows the usage of the cloud computing features to offer fast scalability of the network when it is demanded. However, such work is still in early stages and needs validation. Nevertheless, P4 it's continuously being improved to offer several other

features that were not possible up to the time of its creation. For example, at the time of writing, P4 allows the offloading of features to the dataplane. Such feature, with the right equipment and solution design, can enhance the network performance, for multiple parameters, such as network traffic releasing, lower latency, higher throughput, etc. Having the possibility to implement P4 based solutions can ease the network managers jobs while reducing the time to market of the developed solutions, which can lead to a massive adoption of P4 [8].
At the time of writing, it is very hard to deal with state related information with P4 and therefore hard to define publish-subscribe protocols that usually rely on TCP for reliability purposes. No one can deny the benefits of TCP for publish-subscribe protocols. While it eases the protocol definition, given the reliability purposes are the responsibility of the transport protocol, it guarantees the deliver of all packets. However, that is not a problem if publish-subscribe protocols are able to run over stateless transport protocols. Although still unspecified, Zenoh[9] is the protocol that ticks all the needed boxes. It is an emerging un-specified publish-subscribe protocol mainly targeted to IoT. It implements its own reliability measures and therefore it is designed to run over stateful and stateless transport protocols. That is, it can run over UDP, making it the perfect candidate to implement in P4, given its limitation regardless stateful information.
The outcome of the work here presented is the study, validation and evaluation of offloading features in publish-subscribe systems to the dataplane through the usage of the P4 ecosystem. Such system can be used, among others options, for alert messaging, that is, if a system failed or it is failing, triggering actions, or just collecting metrics. One thing that those systems have in common is the need of faster message delivering. The faster the message is delivered the less problems are likely to exist.
The key takeaway of this work is the suitability of P4 for such systems and how it compares to traditional solutions.

1.2 Problem Statement

Solutions such as SDN and NFV brought lots of benefits. Reduced CAPEX and OPEX, centralized control and fast scalability of the network are some of the examples. However, there are still limitations. For example, the tools used in SDN work very well with known network technologies and protocols, yet, fail to work with the new and emerging ones. That means network managers have their hands tied when it comes to deploy and experiment new solutions. Therefore, the deployment of new solutions

and/or new experiments will only be able to be done when the software and/or device manufacturer starts to provide support for the used protocol. The outcome of this behavior is reduced innovation, something that goes against the principles of SDN and NFV.

1.3 Objectives

The overall purpose of this work is to experiment and evaluate how offloading publish-subscribe features to the dataplane will influence the network performance. More specifically, study the value and benefits of publish-subscribe function offloading, from software to hardware, using the P4 system. The network devices are fully programmable and must be able to reduce the network traffic while speeding up the processing time, due to their dedicated hardware.
With an IoT publish-subscribe scenario in mind the overall goal can be divided into several minor objectives. The first one is to study the emerging Next-Generation Software Defined Networks (NG-SDN), its underlying technologies and how it differs from current solutions. The second objective is the development of a proof-of-concept ecosystem designed to validate the followed approach, that is, to experiment how such technologies can interact with each other and how can that be used to solve a specific problem. With that validated, a specific use case must be chosen, implemented using the with the same P4 ecosystem developed before and conclusions must be drawn on how such solution is compared to more traditional ones.

1.4 Document Structure

Chapter 2 starts to describe what is 5G, how it works and how it differs from the previous generations of mobile networks. Given that explanation, it presents some of the traditional tools used in order to fulfil the needs of 5G networks, such as NFV and SDN. Such tools, even though heavily used in today's networks suffer from several faults that need to be addressed. The chapter then offers a new solution that aims to fix the flaws and errors of the presently used technologies. It gives an overview on what the software stack is and how it compares to others technologies available. At the end of the chapter an overview of the developed work with this new technology stack

is presented.
Chapter 3 defines the experimental setup that was created as a proof-of-concept. It starts to describe the technologies used and the reason of its choice. It explains what Docker and Mininet is and the benefits of testing state-of-the-art networks with those two technologies. Then it essentially describes how it was designed and the blocks that compose the system, as well as the reason of choice for the block technology and even the version used. It explains how software principles, such as CI/CD, can be integrated into P4 network based solutions and, in the end it presents a top to bottom approach for a complete solution with a firewall use case, where the system is controlled by and external app. Such app offers a GUI and is intended to ease the work of network managers when deploying the routing behaviour to the network devices. It then details how Zenoh can be implemented using the conceptualized system developed previously for validation. It describes both the ingress and egress pipeline behaviours for the designed solution, as well as the behaviour and implementation of the traditional solutions that are used to evaluate the system against. At the end of the chapter the solution is evaluated against the traditional solutions for several network metrics, such as latency, jitter and lost packets. Lastly, it presents other evaluations that can help to explain the obtained results and take conclusions.
Chapter 4 presents the conclusion inferred from the results obtained and offers a personal view on what must be the continuation of this work. That is, improve the designed solution through the increment of features that can be offloaded and what those features can be.

1.5 Contributions

The work here developed makes use of all the available technology stack to implement the offered solutions and it is also the first to implement Zenoh with state of the art technologies. Such work, given it is an unspecified protocol, can be a good starting point for future and more production grade solutions. Also, this work is one of the few that implement publish-subscribe systems using the P4 ecosystem.
It resulted in an accepted and presented poster to EuroP4'21

Alexandre Santos, José Quevedo and Daniel Corujo, "Realizing Zenoh with programmable dataplanes"

Chapter 2

State of the art and Background

The new generation of wireless broadband networks is a big upgrade from the previous network generations. Even though it is build on top of the older generation it aims to be faster while offering more bandwidth and lower latency. That way, services like autonomous driving, which heavily rely on big data and low latency communications, can really start to be the focus of car and AI companies, meaning an improvement to both industrial and personal lives. It is then more software focused than all its predecessors. However, there are already some known faults that can be corrected, especially in software defined networks. This chapter starts by presenting the 5G objectives and how SDN and NFV technologies were leveraged to further improve the operation of 5G services. It is followed by the theoretical background for a new generation of software defined networks with fully independence and even better service delivery in mind.

2.1 Fifth Generation of Mobile Networks (5G)

The Fifth Generation Networks is the new wireless broadband networks and they are much more than just a new faster version of the previous generation. One thing, not so common, is that it is built on top of the previous generation, meaning that several resources are shared. One may ask, if it is built on top of the previous generation, i.e. there is a shared infrastructure, how can it be different and be such a big improvement. Well, unlike the previous generations, 5G must provide support for all of the previous generations RATs whatever the generation they are used in, something that 4G networks did not have. Still, using the same infrastructure does not mean that no new hardware will be added, in reality, 5G will have more cells deployed than any of the previous generations [10]. In 4G, towers were big and high energy dependent, yet, in 5G, towers can be smaller due to their energy efficiency, making them perfectly fit for higher frequencies. However, the higher the frequency, the greater the attenuation during signal propagation, meaning less range and more towers deployed. In fact, 5G is targeting not only outdoor but also indoor cells, and although the latter is targeted for really specific areas, the cells are so small that are often called femtocells and have a range of about tens of meters [11].

4G was heavily characterized by the change from both packet and circuit to all IP

packet switching and minor changes in the network from hardware to software. This new generation however, aims to be more energy and cost efficient while providing less latency and decoupling between network components, have higher data rates, faster scalability and be based on a cloud native environment where services are deployed using containers [12]. All of those features are sustained by technologies as SDN and NFV, see Section 2.3.

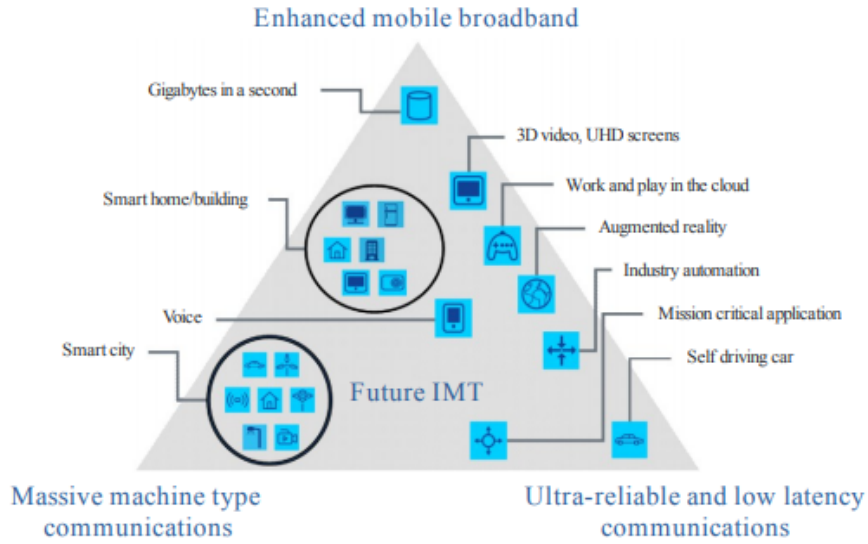


Figure 2.1: 5G Mobile Network Services [13]

In Figure 2.1 are presented the three main services categories classified by the International Telecommunication Unit (ITU).

- **Enhanced Mobile Broadband (eMBB)** aims to provide high data rates at low latency while providing uniform connectivity throughout the whole coverage area. That will translate in a better Quality of Experience (QoE) for the user, where some of the use cases are virtual reality (VR) and augmented reality (AR). It will also allow high user mobility, such as in cars or public transport.

- **Massive Machine Type Communications (mMTC)** aims to focus on services that demand high connectivity, such as smart cities and smart agriculture.

- **Ultra-reliable and Low-latency Communications (uRLLC)** aims to be the key enabler for Industry 4.0, since it is focused in latency-sensitive services such as assisted and autonomous driving and remote management. One of the innovative features of this service is the usage on mission critical services and ultra reliable communication where it can be said that every second counts.

To summarise eMBB is focused on high bit rate services, mMTC on high connectivity equipment and uRLLC on low latency services. Nevertheless, one should keep in mind

that having three different categories does not mean that use cases cannot belong to several services at the same time. For example, mobile voice services are one case that can be inserted in all of the main categories, just like it is shown in Figure 2.1.

2.1.1 5G Architecture

One of the key features for the implementation success of such a complex solution as 5G, is standardization. The Third Generation Partnership Project (3GPP), who is actually a union of seven telecommunications standard development organizations, such as ARIB, ATIS, CCSA, ETSI, TSDSI, TTA and TTC, is the industry group responsible for such task, having set the first 5G specifications with its Release 15, whose focus where 5G new radio and core network. Up to the date of writing, both Releases 15 and 16 are under the frozen state, meaning that specifications are closed and no more functions can be added to such release. Due to COVID-19, Release 17 was delayed and it is expected to be frozen by mid 2022.
In Figure 2.2 it is depicted the overall 5G system architecture for non-roaming scenarios, i.e. for 3GPP accesses.

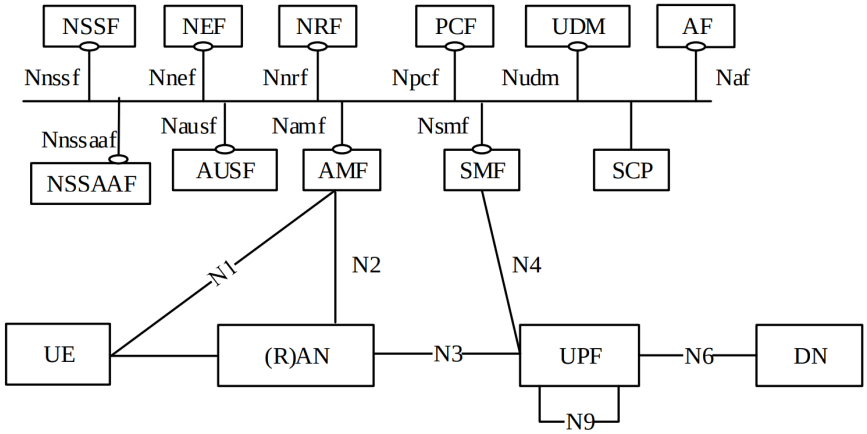


Figure 2.2: 5G System Architecture [14]

- **User Equipment (UE)** - a device connected to the network.
- **Radio Access Network (RAN)** - network infrastructure that acts as a connection point for user equipments to access a network via radio signal, usually a core network.
- **Data Network (DN)** - network which users want to access.

- **Access and Mobility Management Function (AMF)** - it receives all user related connection and session information, however, it is not responsible for session information processing. It's only responsibility is to handle events such as registration, connection, reachability and mobility.

- **Session Management Function (SMF)** - responsible for UE's session management and configuration of traffic steering at UPF.

- **User Plane Function (UPF)** - it is the gateway for traffic data from the Access Network, traffic which is delivered through a GPRS Tunneling Protocol (GTP) tunnel. Tasks performed by UPF can be packet routing and forwarding, policy enforcement, interconnection to DN, etc.

- **Policy Control Function (PCF)** - it is used not only for QoS and charging for sessions, i.e. it controls and monitors the usage of resources, but also for controlling service access and authorization, meaning that PCF can verify the right to access a specific service stored in a Unified Data Repository (UDR).

- **Unified Data Repository (UDR)** - it is the entity responsible for the storage and retrieval of subscription data by the UDM, storage and retrieval of policy data by the PCF and storage and retrieval of structured data for exposure.

- **Authentication Server Function (AUSF)** - as the name suggests, it is responsible for the authentication of UEs and the supply of key material to the requester NFs.

- **Unified Data Management (UDM)** - since it is responsible for the generation of authentication credentials and handling of the user identification based on subscription data and management, UDM offers services to the AMF, SMF, SMSF, NEF, GMLC and AUSF. It can be either statefull or stateless, although in the latter case it uses UDR to store information rather than using local memory.

- **Application Function (AF)** - interacts with core network in order to provide services to support, for example, traffic routing influence, NEF access, PCF interaction for policy control and IMS interaction with core network. Based on the implementation, AFs considered to be trusted by the operator can be allowed to interact directly with relevant Network Functions.

- **Network Slice Selection Function (NSSF)** - selects the set of Network Slice instances serving the UE and determines the AMF Set to be used to serve the UE, or, based on configuration, a list of candidate AMF(s).

- **Service Communication Proxy (SCP)** - some of the functionalities offered by SCP are indirect Communication, delegated Discovery, message forwarding and routing to destination NF/NF service and message forwarding and routing to a next hop SCP.
- **Network Repository Function (NRF)** - it is responsible for service discovery function and for maintaining the health status of NFs and SCP.
- **Network Exposure Function (NEF)** - supports secure exposure of capabilities and events to 3rd party, AFs, Edge Computing, etc and provides a means for the Application Functions to securely provide information to 3GPP network and the translation of internal-external information. It also exposes analytics for external parties.

2.2 Network Functions Virtualization (NFV)

Network Functions Virtualization (NFV) is a technology that is designed to boost the deployment of services in the network. It aims to implement Virtual Network Functions (VNFs), which generic servers or cloud infrastructures can instantiate without the need of specialized and dedicated hardware, decoupling the hardware from the software. Some of the NFV benefits are reduced CAPEX and OPEX, increased time to market while encouraging the development of open source software which improves innovation [15].
Depicted in Figure 2.3 is the reference architecture for NFV, standardized by the European Telecommunications Standards Institute (ETSI). The architecture is only described at function level, making the implementation details open for everyone who wants to deploy such model.

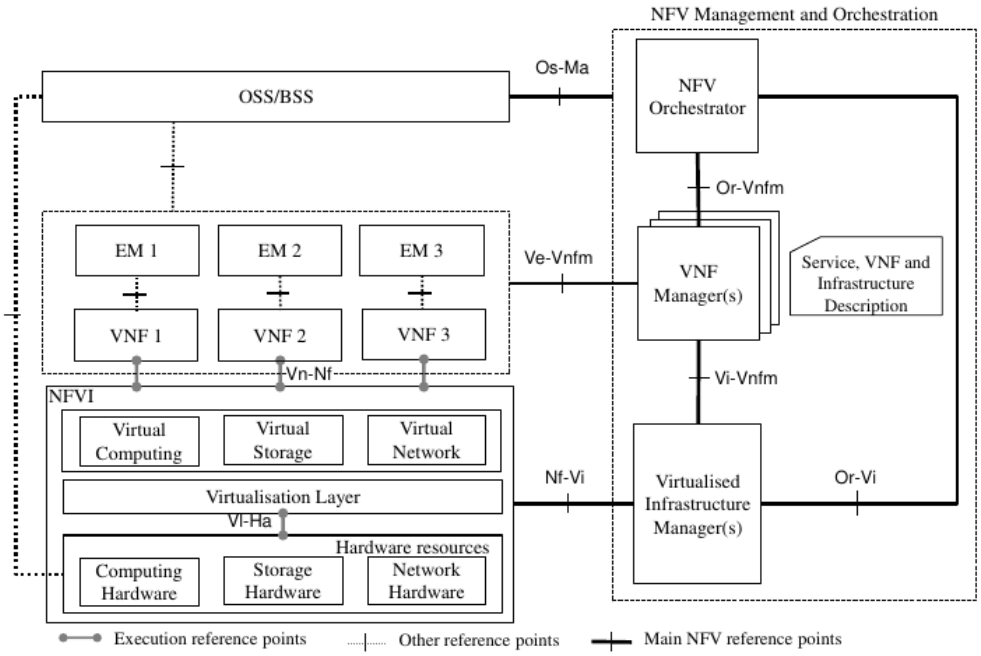


Figure 2.3: NFV reference architectural framework [16]

Analysing the figure above it becomes clear that exist three main working domains easily identified. The first, contains the virtualized network functions. The second, at the bottom of the image, is the Network Functions Virtualisation Infrastructure (NFVI), which is composed by all the hardware and software components needed for the deployment, management and execution of VNFs. The last, presented at the right part of the image, is the NFV Management and Orchestration that comprises four internal components:

- *NFV Orchestrator* - is in charge of the orchestration and management of NFV infrastructure and software resources.

- *VNF Manager* - it is responsible for VNF lifecycle management, i.e. for the instantiation, update, query, scaling and termination.

- *Virtualised Infrastructure Manager* - it comprises the needed functionalities for the control and management of the interaction of a VNF with computing, storage and network resources. It performs resource management, being in charge of the inventory of software and network resources, the allocation of virtualisation enablers, etc, but is also responsible for operations like root cause analysis of performance issues and the collection of fault information. Usually, multiple instances of this component are deployed.

- *Service, VNF and Infrastructure Description* - it is a dataset with information regarding the VNF deployment template, VNF Forwarding Graph, service-related information, and NFV infrastructure information models.

The architecture comprises another component, which is the Operations Support Systems and Business Support Systems (OSS/BSS), already present at the most of the telecommunications service providers stack. However, there is a challenge if such systems are still legacy, making the update of such systems a priority in order to be able to fulfill the 5G use cases [17].

2.3 Software Defined Networks (SDN)

Motivated by the advances in cloud computing and the need of CAPEX and OPEX reduction, a new concept of networks have emerged. SDN is a new type of networks that aims to make them more flexible and scalable through an architecture where software is decoupled from hardware, unlike previous production networks where each network equipment implemented the full stack. It does that by separating the control, or "Network OS" [18], and data planes. The data plane is responsible for the forwarding of packets, the reason why it is also called forwarding plane, while the control plane is responsible for the forwarding decisions. One of the key aspects of the control plane its the centralization it brings. Having a centralized control reduces memory usage since network topology does not need to be stored in every network equipment but rather on a single device, or multiple for redundancy and fault tolerance. It also improves network agility since multiple network equipment can be updated concurrently and therefore the network adapts faster to topology changes. Even though SDN was not created to fulfil the needs of 5G, since it emerged way before, it is widely used by network operators and providers due to the characteristics mentioned above.

2.3.1 Architecture

The overall architecture is depicted in Figure 2.4 and it can be separated in three different layers.

- **Infrastructure Layer** - contains the network equipment managed by the controller(s), typically hardware devices.
- **Control Layer** - contains one or more SDN controllers and acts as a middleware between the Application and Infrastructure layers, abstracting the all of the

hardware details from the software. The most used protocol for the interaction with the Infrastructure layer is OpenFlow.

- **Application Layer** - contains software that interacts with the SDN controller and whose function is to define the network behaviour. Applications can be as simple as metric collectors to highly complex routing algorithm enforcers.

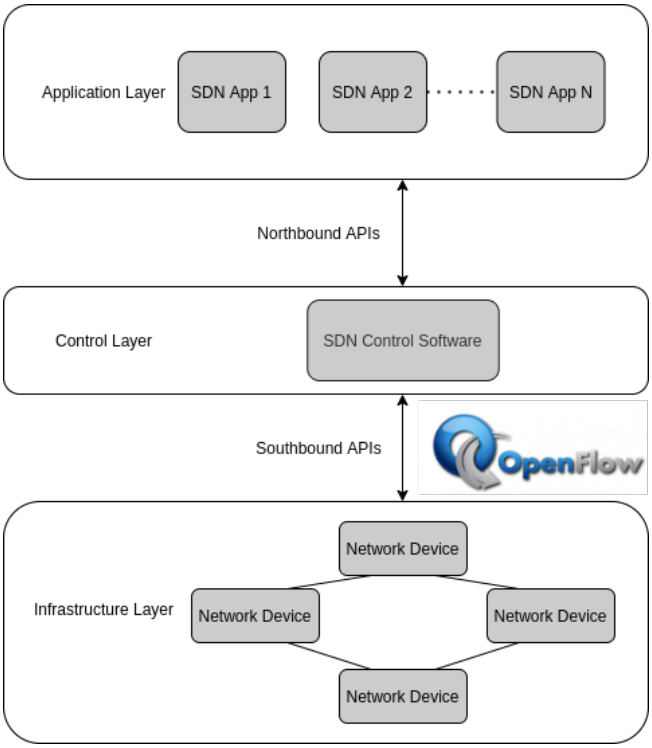


Figure 2.4: SDN Architecture

One key aspects of SDN is that, for the same network, different equipments can be controlled by different controllers, as depicted in Figure 2.5. Different parts of the network can even use different southbound technologies and network equipment software without network malfunction, i.e. traffic still go through the network with the desired behaviour.

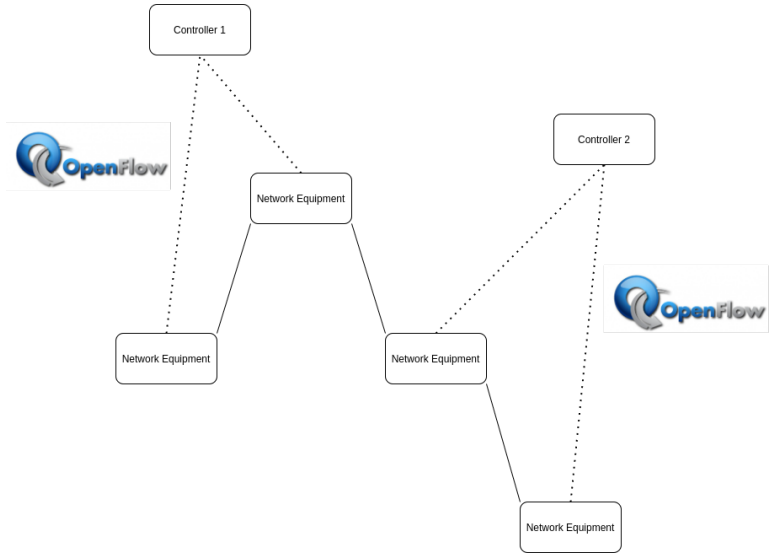


Figure 2.5: Multiple Controller Schema

While NFV and SDN are not required to one another, they are highly complementary, one can even say that SDN can be described as one of the NFV enablers. In Figure 2.6 it is presented the possible solution, proposed by ETSI, where SDN resources can be used in NFV architecture. In blue are presented the possible SDN resource locations, where a) is a physical switch or router, b) a virtual switch or router, c) a software based SDN enabled switch in a server NIC and d) a switch or router as a VNF. In purple is presented the possible location of the SDN controller. In 1) the SDN controller is merged with the virtualized Infrastructure Manager functionality and therefore the two functions are not distinguishable, in 2) the SDN controller is virtualized as a VNF, in 3) the SDN controller is part of the NFVI and is not a VNF and finally in 4) the SDN controller is part of the OSS/BSS. There is another possible location where the controller is a physical network function (PNF), but it was not studied and therefore does not appear in Figure 2.6.

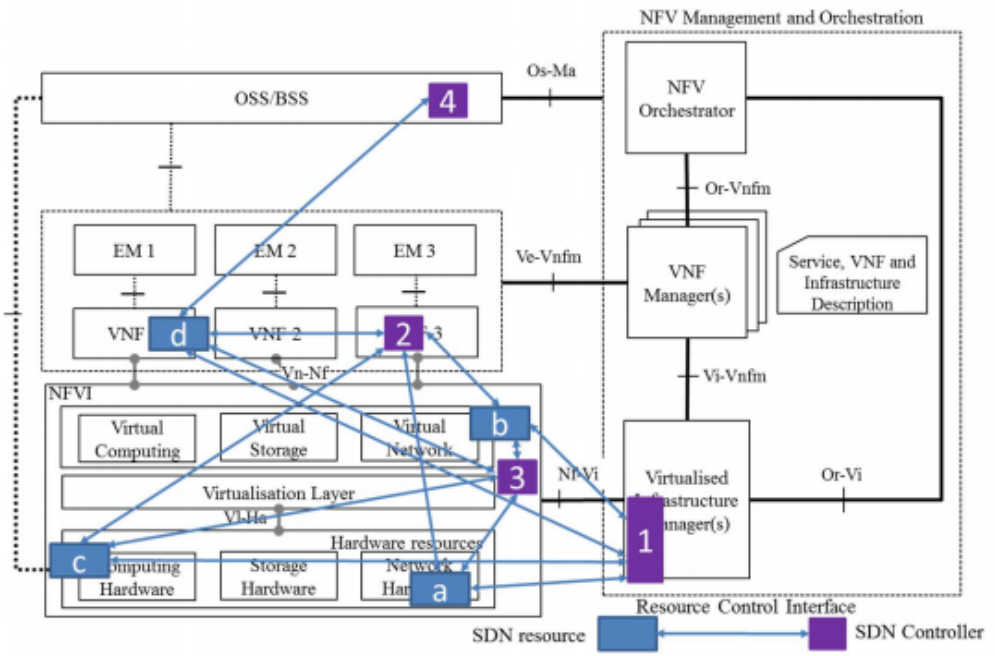


Figure 2.6: Possible SDN Resource Locations in the NFV Architectural Framework [19]

2.3.2 Protocols

OpenFlow is the standard for the interaction between the control and infrastructure layers. First, one is going to present OpenFlow and how it gained advantage over all its opponents and then present some of the other possible solution and how they differ from OpenFlow.

- **OpenFlow** - this protocol was first proposed in 2008 as a communication protocol to control the forwarding plane of a network or a device, however it was only in December 2009 that the first version was released. In March 2011, there was a critical event that would change OpenFlow forever. Open Networking Foundation (ONF) was created with the specific purpose of accelerating the delivery and commercialization of SDN, and it is today the entity responsible for the release and management of new OpenFlow specifications. ONF leverage on OpenFlow and its standardization was an important factor for it to become the most used southbound protocol. Up to the time of writing, several other solution emerged as OpenFlow alternatives but ONF still continue to base its solutions on this same protocol [6]. OpenFlow is used to carry control messages from the centralized controller to network equipment and started with a limitation of a single flow

table with three components in a flow entry, that were Header Fields, Counters and Actions. Throughout the years, it evolved to several fields, multiple tables, meters and groups. Fields which are pre-defined and fixed [20, 21].

- **Open vSwitch Database (OVSDB)** - OVSDB appears as an alternative for OpenFlow in multiple online searches, however, that is not the case. It is assumed that only both are mutually exclusive, but that is not true. While OpenFlow is designed to program flow rules, OVSDB main function is specifically to configure the OVS[1] itself, where OVS is a virtual switch that uses the OpenFlow protocol. Therefore it is common to use OVSDB alongside OpenFlow [22].

- **NETCONF** - It provides the mechanism for the installation, manipulation, and deletion of network devices configuration [23].

One could think that southbound protocols compete with each other for space in SDN environments. However, in fact, several solutions use multiple southbound protocols in parallel. OpenFlow is specialized in adding forwarding behaviour into network devices whereas OVSDB and NETCONF are specialized into device configuration and therefore often used alongside with OpenFlow. Many more examples exist but are not the focus of this work.

2.3.3 Network Controllers

The network controller on a SDN is the critical point of its success [24], since it is responsible for the control logic of the network.
There are multiple open-source controllers available which can be divided into two categories, physically centralized or distributed [24]. A centralized SDN control implies that only a single controller manages the entire network, which can lead to scalability issues. Examples of open-source centralized controllers are Ryu(Python) [2], POX(Python)[3], NOX(C++)[4], FloodLight(Java)[5] [25]. Distributed SDN control means that several controllers can manage the network, and is known for mitigating the issues brought about by centralized SDN architecture (poor scalability, Single Point of Failure(SPOF), performance bottlenecks, etc). That can be achieved either through a flat or hierarchical architecture. In the flat architecture, each area of the network is assigned to a

[1] Open vSwitch website:`https://www.openvswitch.org`
[2] `https://ryu-sdn.org/`
[3] `https://github.com/noxrepo/pox`
[4] `https://github.com/noxrepo/nox`
[5] `https://groups.io/g/floodlight`

specific controller. The hierarchical, on the other hand, the control plane is vertically partitioned, that is, it uses multiple levels (layers) depending on the required services [25]. Examples of open-source physically distributed controllers are ONOS(Java) [6] and OpenDayLight(Java)[7].

2.4 Next-Generation Software Defined Networks (NG-SDN)

In the previous sections, the definition and tools of 5G networks were presented with special focus on SDN and NFV. This section, however, aims to point out the flaws in the current solutions and present possible alternatives that could be the next step towards the new generation of networks, either through the update of 5G specifications or the definition of Sixth Generation of Mobile Networks (6G).

2.4.1 Overview

The objective behind SDN and NFV was to give freedom, from proprietary hardware and software, to network operators, while deploying more agile networks. Even though network management got easier, with the network control led by applications, networks are still not able to adapt as fast as desired. Operators are still stuck to proprietary vertical stacks, i.e., equipment manufacturers are still the ones with power, deciding when to release, for example, SDN updates, such as support for new OpenFlow versions. In reality, OpenFlow is also one of the factors that boosted the arising of other solutions. Given the close way of how OpenFlow is implemented, having pre-defined fields, it does not scale well on protocol update or even new network protocols. New solutions aim to provide freedom from equipment vendors while making the network more scalable and agile even for unknown protocols.
The overall design and tools of NG-SDN are to change the control of the networks to software developers. More and more networks are to be controlled solely by software, where hardware is just a means to run that same software. One key factor that brings network deployment and management towards software development, is the possible usage of well known practices, such as continuous integration and continuous delivery, to deploy and update networks.

[6] `https://opennetworking.org/onos/`
[7] `https://www.opendaylight.org/`

2.4.2 Bare Metal Switching

Unlike traditional solutions, where network providers are stuck to a stack of a specific vendor, i.e. the equipment has already pre-defined hardware and software, bare metal solutions have a degree of freedom in the customization of the equipment hardware while saving capital and operational costs using open source software. Such solutions also grant faster innovation, greater control of the services and the change of the device's hardware and software, something not possible when dealing with off the shelf products and proprietary software. However, if one is not interested in building the equipment component to component, bare metal switch vendors such as EdgeCore, Delta and others, already offer off the shelf hardware solutions, with the software still up to the operator. With full vendor abstraction in mind, bare metal solutions are the standard to have deployed in P4 architectures.

2.4.2.1 P4

P4[8] is a high-level language designed to data plane programming of network devices. It enables developers to promptly and easily define, at a very fine level, the desired forwarding pipeline, using a C like sintax and semantics. That pipeline defines the way equipments process and forward packets, providing an easy way to deploy new and custom protocols and services. One of the best examples to show how P4 is easily used to deploy and test new protocols is the work of Kozlowski *et al.* [26]. The authors used P4 to test a quantum network protocol, the Midpoint Heralding Protocol, on a quantum simulated network. Through the use of P4 programs, the authors were able to fully specify how the dataplane should process packets and the expected format of packets and therefore able to address quantum principles such as entanglement. With entanglement being a property of quantum systems which states that, when particles are twisted together in such way, they form a single system and the observable value of one is equal to the observable value of another, and therefore every operation applied to one particle correlates to the others as well [27, 28].
Such work would not be possible without the protocol independence offered by P4. The main objective of P4 is to be able to fully program high performance forwarding ASICs, software switches, FPGAs, NICs or CPUs, without limitations, even after programs are deployed into production. That way, equipments can dynamically adapt to network configuration and topology changes. Currently, P4 as already passed through multiple

[8]P4 specification page: `https://p4.org/p4-spec/docs/P4-16-v1.2.1.html`

version revisions being $P4_{16}$ the final version implemented. Depicted in Figure 2.7 is the overall architecture of P4.

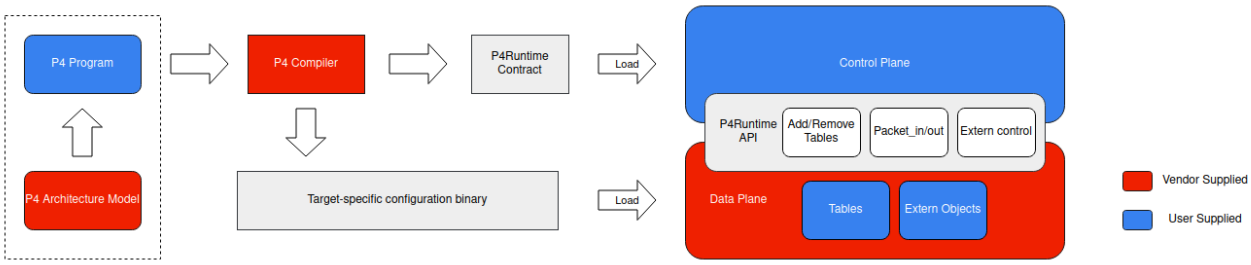


Figure 2.7: P4 Architecture

2.4.2.2 P4 Compiler

The P4 compiler is a specific compiler for the P4 programming language whose function is to transform the high-level pipeline definition into low level code readable by the network equipment. The compiler essentially produces two files upon the compilation of a P4 program. One, named target-specific configuration binary in Figure 2.7, is different depending on the target. Target here represents a physical or software device. That file is used by the target to process packets in a way that is consistent with the pipeline defined in the P4 program. The other generated file, named P4Runtime contract in Figure 2.7, is equal for every target and represents a schema of the P4 program in a Protobuf Text format.
For switches other than software, for example, switches with the barefoot tofino chip, the compiler needs to be provided by the chip vendor. In the software switches case (BMv2), the community offers an open-source baseline solution of a P4 compiler[9] that can be changed to offer more features accordingly to the user needs. The work of Zanna *et al.* [29] is one of the examples. The authors developed an extension to the P4 reference compiler, offered by the P4 community, to support the monitoring and control of wireless network frames.

2.4.2.3 Forwarding Pipelines

It was already mentioned that P4 programs define the desired pipeline of a network equipment. However, one has not defined what this pipeline or what its purpose is.

[9]P4 reference compiler GitHub page: `https://github.com/p4lang/p4c`

The very first step of a network equipment on a packet arrival is the extraction of the packet headers, whether they are Ethernet, IPv4 or IPv6, TCP, etc. The information extracted on those headers is later used in order to know what to do next or where to send a packet. High-speed switches like P4 ones, use a multi-stage pipeline to do that. This way, even though there is a end-to-end latency in the order of nanoseconds, multiple packets can be processed in a parallel way, i.e. at the same time. For instance, while one packet is at stage 2 of header extraction, if there is at least one packet in the waiting queue, then it can be processed on the stage 1 of that pipeline. Pipelines have different implementations based on the freedom of the switching chips. Switching chips like the ones provided by Barefoot Networks [10], where pipeline stages are programmable, have a different pipeline implementation than the switching chips provided by Broadcom [11], where the stages are fixed function, i.e. the chips are limited to do the operations already defined. While the former type of chips has the most freedom and future scalability, the latter type of chips is the most common in the industry. Even though fixed-function chips dominate the market at this point, the objective of this work is to provide an overview of the programmability benefits . So, for the purpose of the job one will only present the architecture of programmable pipelines and the purpose of its different stages.

Every programmable pipeline follows the architecture depicted in Figure 2.8, which gives an overall view of the Protocol Independent Switch Architecture (PISA). It is visible that it is composed by three main components, the Parser, Match-Action Pipeline and finally, the Deparser.

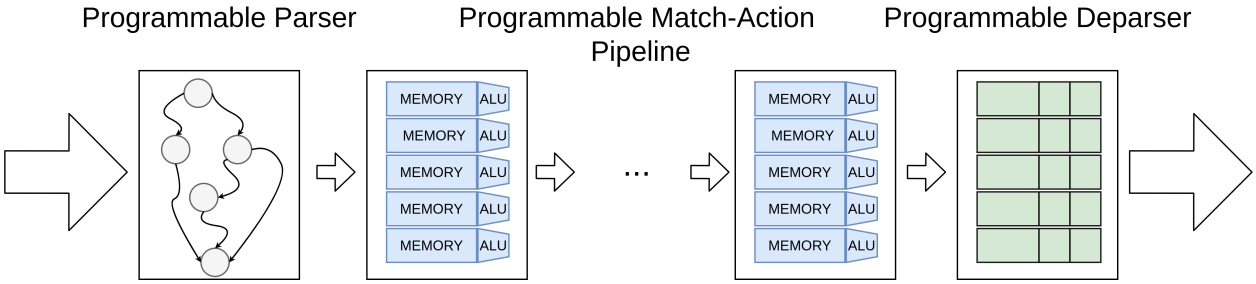


Figure 2.8: Protocol Independent Switch Architecture (PISA)

The parser is responsible for packet header extraction, that is made by mapping the bits into pre-defined structures. The match-action pipeline is responsible for performing actions on the packet, i.e., add or change headers, store metadata, etc. The deparser is

[10] Barefoot Networks Chips: `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html`

[11] Broadcom Chips: `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs`

responsible for gathering all of the packet valid information, build the packet and send it to the desired egress port.

2.4.2.4 Pipeline Abstraction

Different targets implement the physical pipeline differently. It is the job of the P4 compiler to map the logical to the physical pipeline. To make this possible, there are several logical architectures designed for different targets. Table 2.1 summarizes the existing architectures and targets[12].

Target	**Pipeline**
BMv2 Simple Switch	v1model.p4
BMv2	psa.p4
Tofino	tna.p4 [13]
NIC	pna.p4
eBPF	ebpf_model.p4
DPDK	psa.p4

Table 2.1: Targets and available architectures

While tna.p4 is targeted for Tofino chips, pna.p4 for NICs, etc, psa.p4 is the ideal logical pipeline intended to run on every single target.

Figure 2.9: Portable Switch Architecture (PSA)

Comparing Figures 2.8 and 2.9, the differences are very clear. First the PSA pipeline gained a new component, the traffic manager, responsible for queuing, replicating, and scheduling packets. Second, the pipeline is now clearly divided into Ingress processing (left blocks) and Egress processing (right blocks).
PSA represents the ideal architecture, however, it is still a work in progress and therefore, developers must use the architecture that fits them the better.

[12] `https://github.com/p4lang/p4c/tree/main/p4include`
[13] tna.p4 must be provided by the chip vendor along with the compiler

2.4.3 Switch Operating System

Just like a server or a personal computer, bare metal switches also need an operating system running. The common foundation for this Switch OS is the Open Network Linux (ONL), which is an open source project of the Open Compute Project (OCP). Up to the time of writing there are several switch OS options available to run on top of ONL. Examples of such options are Stratum[14], SONiC[15], which will be both described in the next sections, FBOSS [16], DENT [17] and DANOS[18].

2.4.3.1 Stratum

Stratum is an open source silicon independent switch operating system developed by ONF for SDN, where silicon independence means that Stratum wants to be free from proprietary silicon interfaces and software APIs. However, Stratum is more referred as a thin switch operating system, meaning that Stratum alone is not able to process and forward packets. In fact, Stratum is installed on top of ONL whose function is to perform hardware abstraction. Stratum is then responsible for providing a set of interfaces capable of configuring the settings and behaviour of the device. That interfaces are exposed on the Northbound side and the three most important are P4Runtime, gNMI and gNOI. All of the mentioned are gRPC services. The overall architecture is depicted at Figure 2.10.

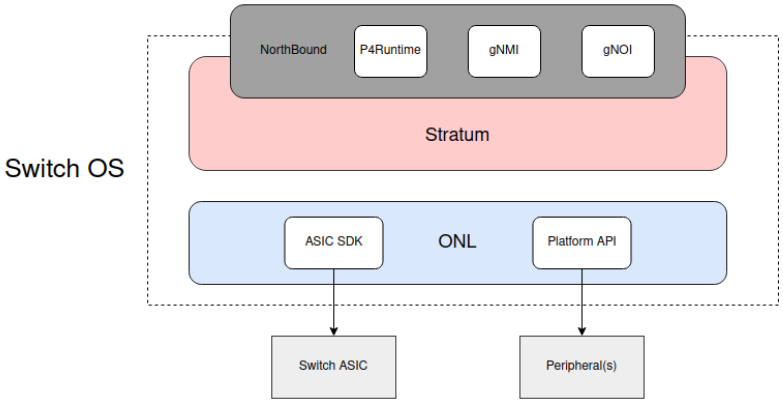


Figure 2.10: Switch OS with Stratum

[14] https://opennetworking.org/stratum/
[15] https://azure.github.io/SONiC/
[16] https://github.com/facebook/fboss
[17] https://dent.dev/
[18] https://www.danosproject.org/

P4Runtime

P4Runtime is an API defined by a program independent interface whose objective is to control the forwarding behaviour of the equipment at runtime, based on the pipeline described by a P4 program.

API	Target Independent[19]	Protocol Independent[20]
P4 compiler auto-generated	✓	–
BMv2 CLI	–	✓
OpenFlow	✓	–
SAI	✓	–
P4Runtime	✓	✓

Table 2.2: Comparison between runtime control APIs

In Table 2.2 it is possible to see how the P4Runtime compares to other used APIs. Analysing the table, it is clear that P4Runtime is the only providing target and protocol independence. The former, in the P4Runtime case, refers to the fact that for different equipment, either software(BMv2) or hardware(ASICs, etc), the P4Info file, or P4Runtime contract as in Figure 2.7, is the same. The latter means that P4Runtime messages do not depend on the data plane packet headers. In other words, messages are processed even for the new unspecified or proprietary protocols if they are described in the P4 program pipeline. Therefore, to summarize, even when the P4 pipeline or the hardware changes, the interface stays the same [30].

One of the key points for P4/P4Runtime is that it aims to be as fast as fixed protocols such as OpenFlow, something that has been already proved [31]. It is an important factor given that if an equipment cannot process packets in a short amount of time, it could lead to buffer overflows and consequently packet losses.

It is also worth mentioning that most of OpenFlow research is heavily control focused and P4 is purely data plane focused, therefore P4/P4Runtime is the perfect alternative to OpenFlow on the data plane. However, while OpenFlow is supported by many devices, P4 and P4Runtime, which are still in early phases, have a limited number of compatible devices.

[19]Target Independence: The contract is equal no matter the device, physical or software

[20]Protocol Independence: Control and data planes messages are not limited by packet headers

Management Interfaces

gRPC Network Management Interface (gNMI) is a service used to manage networks elements using gRPC, which make it way more efficient than others, such as NETCONF, RESTCONF or native REST, whereas gRPC Network Operations Interface (gNOI) is a set of gRPC-based microservices that allow operations such as certificate management, software upgrade, etc.

2.4.3.2 Software for Open Networking In the Cloud (SONiC)

Like Stratum, SONiC is an open-source switch operating system. While Stratum is developed by ONF, SONiC is developed by Microsoft. SONiC differs from Stratum by its ability to offer multiple services, such as BGP, DHCP Relay, SNMP, LLDP, etc, as docker containers, ready to be used out of the box. However, the major difference between these two switch OS is the interface used to interact with it. While Stratum uses P4Runtime, SONiC uses SAI.

Switch Abstraction Interface (SAI)

SAI[21] relies on the same objective of P4Runtime. It aims to keep the programming interface consistent while allowing network hardware vendors to develop innovative hardware architectures to achieve greater speeds. However, like Table 2.2 shows, SAI is not protocol independent, given that it only allows to process what is possible by the SAI pipeline, and therefore it does not offer the flexibility of P4Runtime.

P4 Integrated Network Stack (PINS)

With the benefits of P4Runtime and the tools that SONiC offer out of the box, the PINS[22] project was created. The project is not yet available as an open-source one, but it is expected to be by the end of the year. It is being developed by major companies such as Google, Alibaba Group, Intel, NVIDIA, etc, and incubated at ONF. Such project aims to leverage the tools offered by SONiC with options as P4/P4Runtime. That is achieved by the description of the SAI pipeline in P4 and an optional P4Runtime server added to SONiC. This way the existing tools can be leveraged but new tools can be also added and quickly tested due to the flexibility of P4/P4Runtime.

[21] `https://github.com/opencomputeproject/SAI`
[22] `https://opennetworking.org/pins/`

2.4.4 NG-SDN Network Controllers

In Section 2.3.3 were presented the most used OpenFlow SDN controllers. However, as mentioned in the previous sections, NG-SDN uses P4Runtime and not OpenFlow. At the time of writing there are very limited open-source network controllers that provide P4Runtime support, being ONOS one of them.

2.4.4.1 Open Network Operating System (ONOS)

ONOS was traditionally designed to work with the OpenFlow protocol but it now supports P4Runtime. However, ONOS has already a lot of apps developed for the OpenFlow that should not be changed. One way that the developers found to fix this issue, is to build a translator from OpenFlow to P4Runtime.
As depicted in Figure 2.11, which shows the overall process for flow operations, agnostic and aware apps behave differently. Apps installed to ONOS that are pipeline aware, have a clear definition of pipeline tables, actions, fields, etc, and can create direct flow rules. However, traditional ONOS apps, which are agnostic to a specific P4 pipeline, rely on the pipeliner to transform Flow Objectives into Flow Rules. Tat way, it allows network equipment to interact with the controller the same way as in traditional SDN, when equipment does not know how to process the packet, it is sent to the controller where it is processed and rules can be installed in network devices.

2.4.4.2 Custom P4Runtime based controller

There is a lack of open-source P4Runtime network controllers. If one does not want to use most of the ONOS features, there is no need to waist computational resources on a heavy controller, since it can build their own. An example on how to build a custom P4Runtime network controller can be found here [23]. For that it must use the specification available here [24].

[23]Library example:`https://github.com/p4lang/p4runtime-shell/tree/master/p4runtime_sh`
[24]P4Runtime Specification:`https://p4.org/p4runtime/spec/v1.3.0/P4Runtime-Spec.html`

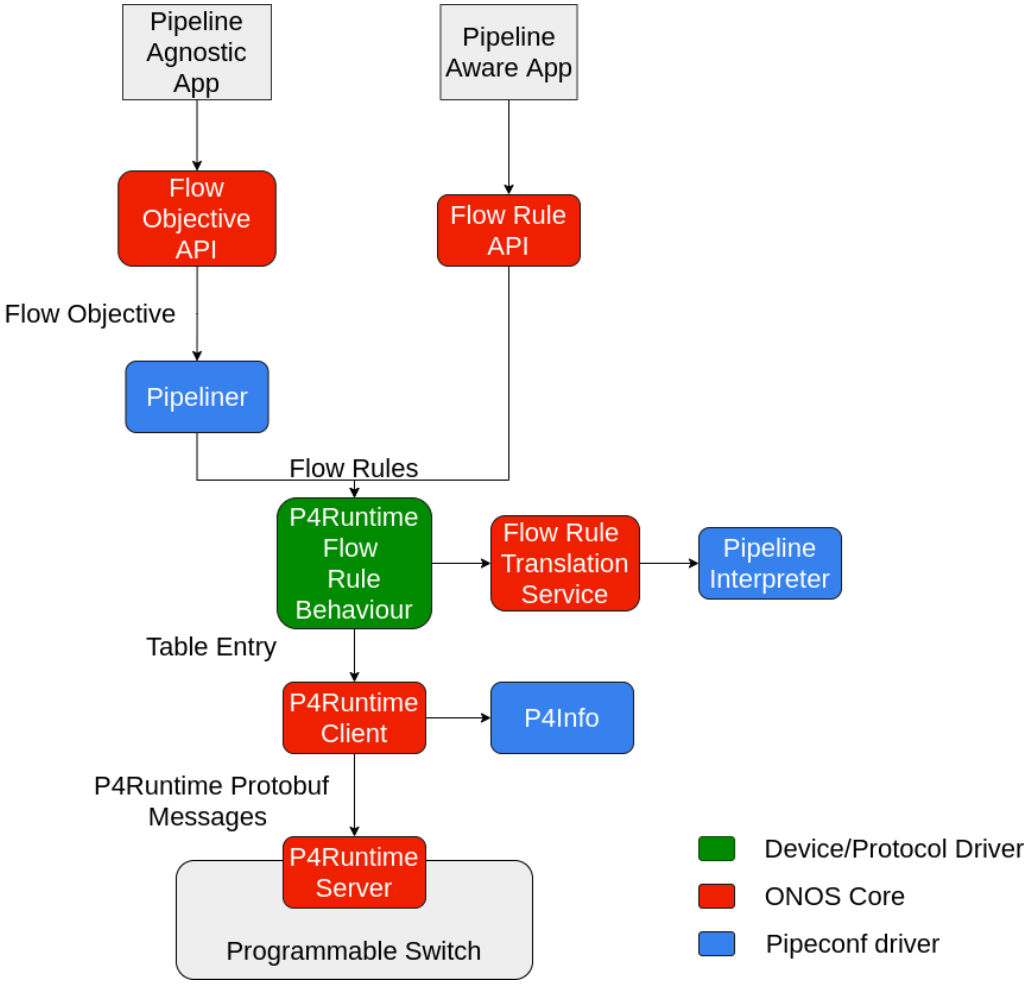


Figure 2.11: Flow Operations

2.5 ONF Solutions

One thing that P4 brought is the freedom of choice when it comes to build a complete solution. One is not meant to be tied to solutions of specific vendor. ONF however is a community driven entity and therefore its work is open source, with relevant products to this work such as ONOS and Stratum. However ONOS is much more than it seems to be. In fact, ONF ships with ONOS several ready to use apps, that can be turned on to do some specific task. The one with most interest for this project is Trellis, which is in fact a group of apps targeted to handle IPv4/IPv6 unicast/multicast, VLANs, MPLS Segment Routing, among others. One of the most important apps of the Trellis app stack is the Segment Routing app, which among the previous mentioned features also allows to use tasks such as ARP and ICMP handling without changing the source code. However that only works with a solution also developed by ONF that is a specific

pipeline, mainly targeted for datacenters, called fabric.p4. Such pipeline is designed to use segment routing features to forward traffic on a leaf-spine architecture, but it can also be used on any other topology.

2.6 Related Work

Along this chapter the P4 ecosystem was described. However, one has not mentioned the possible applications of such technology and what are the targeted areas of researchers. As already stated, P4 is designed to be as flexible as possible, which may be the reason of its growing popularity, meaning that problems from multiple areas can be tackled with P4. Depicted in Table 2.3 are the solutions that are currently being actively studied. As one can see, there are plenty of areas where P4 can be used, and with a tool that is yet to mature, one can only imagine what can be achieved in the future.

Table 2.3: P4 research fields

Monitoring	Detection of Heavy Hitters Flow Monitoring In-Band Network Telemetry DSL-based Monitoring Systems Path Tracking
Traffic Management	Data Center Switching Load Balancing Congestion Notification Traffic Scheduling Traffic Aggregation Traffic Offloading
Routing and Forwarding	Source Routing Multicast publish-subscribe Systems Named Data Networks Data Plane Resilience
Advanced Networking	Cellular Networks Internet of Things Industrial Networking Time Sensitive Networking Network Function Virtualization
Network Security	Firewalls Port Knocking DDoS Attack mitigation Intrusion Detection Systems Connection Security

This work aims to offer a different approach of the already implemented P4 approaches

to publish-subscribe systems. While the majority of the research on P4 publish-subscribe rely on custom protocols, which generally lack validation, this work aims to present a P4 solution to publish-subscribe systems that rely in Zenoh, a rising publish-subscribe protocol targeted to IoT. Through the usage of Zenoh one is able to leverage P4 capabilities and offload some processing to the dataplane and therefore closer to the end user. Also, such solution allows the reduction of packets that travel through the network and so free links for other services.

2.7 Summary

This Chapter started by presenting 5G, what its architecture is and how it will change the world. It then presented NFV and SDN and how are those technologies used by 5G to improve networks performance.
After the introductory concepts it started by presenting what is the new generation of SDN, the overall ecosystem and how it compared to traditional SDN. The main takeaway of the chapter is the presentation of P4, the P4 ecosystem and the available technology stack that can be used for the implementation of specific solutions. The chapter ended with an overview of the work that is currently being developed by researchers with P4 related tools.
The next chapter will present a system that can be used as an entry point to a possible solution of an overall top to bottom product, designed using the P4 ecosystem.

Chapter 3

Scenarios Analysis and Evaluation

P4 is an innovative technology that heavily relies on open-source software. Unlike traditional solutions where vendors offer a top to bottom solution, with such ecosystem developers need to find all the needed pieces and make them work together. This chapter starts by providing an overview of which technologies can be used to build a top to bottom solution and how can they be leveraged to offer a solution not yet available for modern networks. Then, with all the validation done it presents a specific use case which can be leveraged with P4, such as publish-subscribe systems. Traditional publish-subscribe protocols use TCP as transport layer for reliability and security reasons. However, with P4, it is difficult to express TCP due to its limitations regarding managing large state related information, leaving such processing to the network controller and the CPU. That is why most of the publish-subscribe protocols are not to be described by P4, at least for now. Zenoh, on the other hand, implements its own reliability and security mechanisms, making it possible to run over TCP, UDP and others. Therefore, it is the perfect candidate to implement in P4.
Publish-subscribe protocols traditionally rely on a centralized entity for packet forwarding. With P4 such task can be offloaded to the dataplane, reducing end-to-end latency and the number of packets in the network. This Chapter presents three different scenarios to be compared and at the end it presents a comparison of metrics from the scenarios with and without offloading, and also a pure Zenoh implementation with BMv2 as a L2 solution.

3.1 Overall Architecture Design

At the time of writing there is no such thing as a base model to implement P4 solutions. However, given its open-source background, there are lots of building blocks available for one to use. As described in the previous chapter, there are plenty of options to choose from, either to the switch OS, network OS and even target equipment. Due to monetary constraints it was decided to use as many open-source solutions as possible, mainly due to high price of the hardware equipment. In this case not only one equipment is needed but several of them and therefore the cost would be substantial. One way to reduce costs is to use pure free software approaches through the use of the

available Docker images.
Docker is the most popular open-source approach for application oriented containers, and its containerized software will always run the same, regardless of the underlying infrastructure [32]. It emerged to fix the "It works on my computer" statement. One key feature of Docker is that a lot of open-source software is made available through Docker container images, such as Mininet.
One drawback of having all of the system components as independent modules, it's the way they interect with external systems and it's dependencies, so a study of which versions work among components needed to be conducted. After some experiments, it was decided to choose Stratum and ONOS, given both are built by the same entity and therefore the effort needed for things to work tends to be smaller. ONOS however, has multiple docker container versions available, some stable, some pre-released and others still in development phase, where each version can be highly different from the previous. Up to the version 2.2.7, ONOS offered as part of Trellis Apps, the Segment Routing App. For the newer versions of the controller it was not possible to discover through which channels that behaviour is offered and therefore the version used was ONOS 2.2.7[1], which is the last version to provide support for the Segment Routing app.
Depicted in 3.1 is the overall architecture conceptualized, in which due to monetary constraints it was not possible to build the system with state of the art programmable hardware switches. It was decided to developed the system using the available software switch at the present date which is BMv2, even though the performance of such equipment, that relies on the CPU, is no where near the offered by programmable ASICs that the programmable hardware switches have.

[1]ONOS 2.2.7 GitHub: `https://github.com/opennetworkinglab/onos/tree/2.2.7`

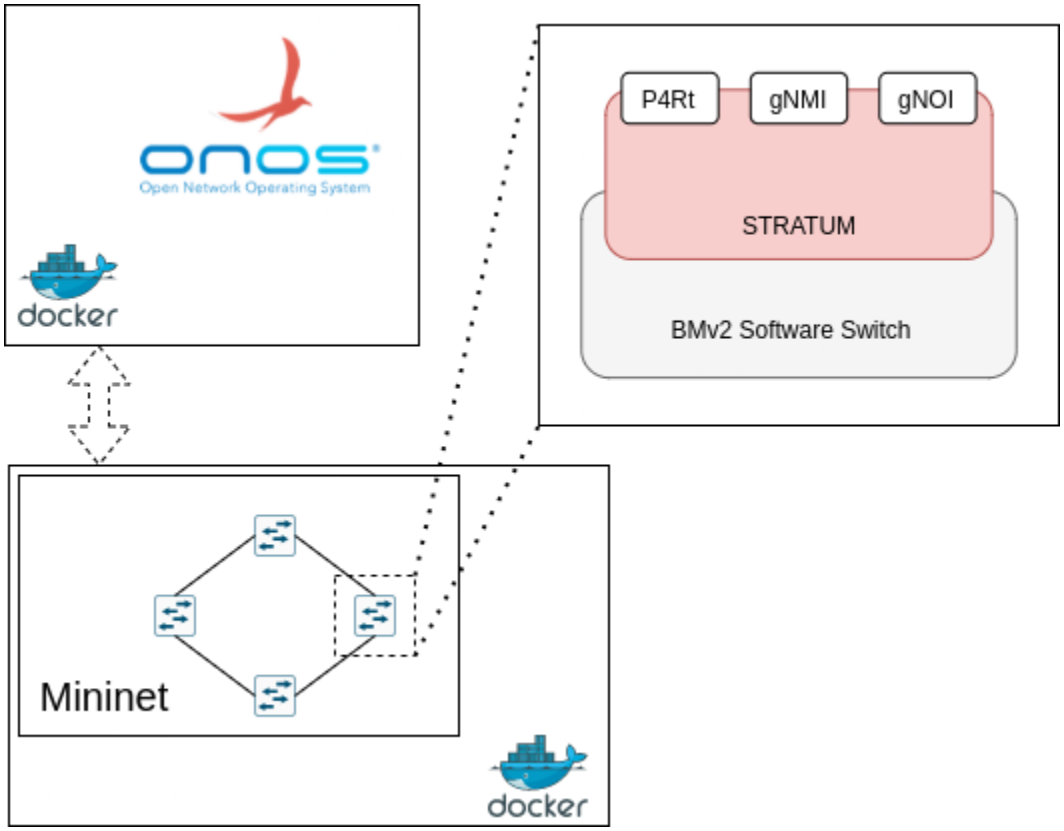


Figure 3.1: Overall system architecture

3.2 Setup Scenario

The experimental setup described in this section is meant to be used as a proof of concept so that one makes sure the overall tech stack works well together and could be used for several use cases rather than just a particular case. One way to initially test the system is to build a simple use case around it. In this particular case a firewall. That is, the developed system must replicate the behaviour of a firewall. The overall idea is to use the developed pipeline to allow or drop packets that arrive on the network equipment based on the packet fields, up to the transport layer. Such behaviour can also be achieved with OpenFlow but the focus at this point is to use the technology available in the P4 ecosystem and develop a top to bottom solution. At the Appendix A there is a collection of files used that can be used as a entry point to replicate the conducted experience.

All of the work developed settled on the premise that the wheel should not be reinvented. So, the overall pipeline derived from the fabric.p4 pipeline, mentioned in the previous chapter, in order to use the behaviour of the Segmented Routing App offered

by ONOS. That way, handling ICMP and ARP do not need any action from the developer. However, to make it work, one must guarantee that the added tables and actions do not break the main pipeline.

3.2.1 Mininet

Mininet is an open-source network emulator [33], widely used by researchers and students for SDN purposes [34]. It features a CLI, for network debugging or running network-wide tests, and a Python API for network creation and experimentation. While it was initially built for the OpenFlow protocol and OVS [35], it is possible to use mininet with custom and more advanced features such as P4Runtime and Stratum [2].
As it might have been notice through its description, mininet eases SDN experiments within a controlled environment. Unlike traditional networks, P4 based networks, are purely software controlled and prone to the errors that such environment brings. Mininet's ability to shut down the entire network and create new experiments, as well as easily change the network topology and also to run multiple devices at the same time was one of the key enablers of the development of such experimental setup and the reason why it was chosen in the first place.

3.2.2 Continuous Integration and Deployment

Traditional networks did not have a tool that allowed to test the network functioning before rules were applied into production. One thing that tools such as P4 brought, is the possibility to integrate the development of networks in Continuous Integration and Delivery cycles. While integration cycles must compile and validate pipelines, compile and test controller apps, deployment cycles are designed to automate the app installation, update processes and the network configuration and topology update. Up to the time of writing, neither version of ONOS support hot pipeline swap, i.e. the target device will stop working for a period of time and there must be at least one other device that can handle requests while a device is updating its pipeline.

3.2.2.1 Gitlab CI/CD

GitLab was used as a code repository along the development process, mainly due to

[2]Mininet Stratum Example:`https://github.com/stratum/stratum/tree/main/tools/mininet`

the possibility of using the built in feature of CI/CD pipelines.
With the possibility to test the developed pipeline each time a change was made, a CI/CD pipeline, depicted in Figure 3.2, was developed. With the integration of the developed pipeline with the GitLab repository, each time a commit is pushed to the repository the CI/CD pipeline is triggered and everything is tested to check for broken features.

Figure 3.2: Continuous Integration pipeline

The CI/CD pipeline works as follows. The first stage is designed to compile the P4 pipeline and in case of failure no other stage is meant to run. The second is targeted to validate the P4 pipeline behaviour through the use a framework described in the next section. The third stage fetches the result from the first stage and compiles the app with the new pipeline. The final stage is targeted to pack the app and have it ready to deploy into production. The pipeline description is available at Appendix A.5.
For the deployment part, even though not used, ONOS offers a REST API designed for such purpose, where a single, multiple or all switches are to be updated.
CI/CD cycles aim to discover errors early in the development phase and with that,reduce shipment times. In the P4 ecosystem that can be achieved with the integration of the Packet Test Framework (PTF) into the pipeline test phase of the integration cycle.

3.2.2.2 Packet Test Framework (PTF)

The behaviour of the network is now described by software, therefore it is important to test such behaviour before systems are deployed into production. The P4 community offers a framework able to test P4 pipelines[3]. It is designed to simulate the switch behaviour on packet arrivals, where packets are to be created by the user for the test purpose.
In this specific case, the tests targeted the drop or not of packets based on the packet fields and they may be source and/or destination IPs, source and/or destination ports, transport protocol, etc. In Appendix A.7, it is presented an example of how a simple drop or allow test is done.

[3]Example: `https://github.com/opennetworkinglab/ngsdn-tutorial/tree/advanced/ptf`

3.2.3 External Control App

In order to control the desired behaviour of the data plane, one needs to be able to control which rules are installed at each device at any moment, to install and remove rules and collect metrics of the dataplane. However it is not feasible to have that information pushed through, for example, the command line interface offered by BMv2 or to check rules installed via a log file. So, using the available features of ONOS, a REST API on the northbound side of the controller was exposed, such API would then be used to control the data plane behaviour.
To interact with the API, an application was developed using React[4].
Depicted in Figure 3.3 is the overall high-level architecture. The system behaviour is the one that follows, on incoming HTTP requests to the exposed controller API, the firewall module methods are called and rules are installed or deleted on the target devices. One benefit of having the system divided as in Figure 3.3, is that software components such as the controller are easily scalable. So, rules can be stored temporarily or permanently in a database for external apps to access it, reducing the number of calls to network devices and only interacting with network devices to install or delete rules.

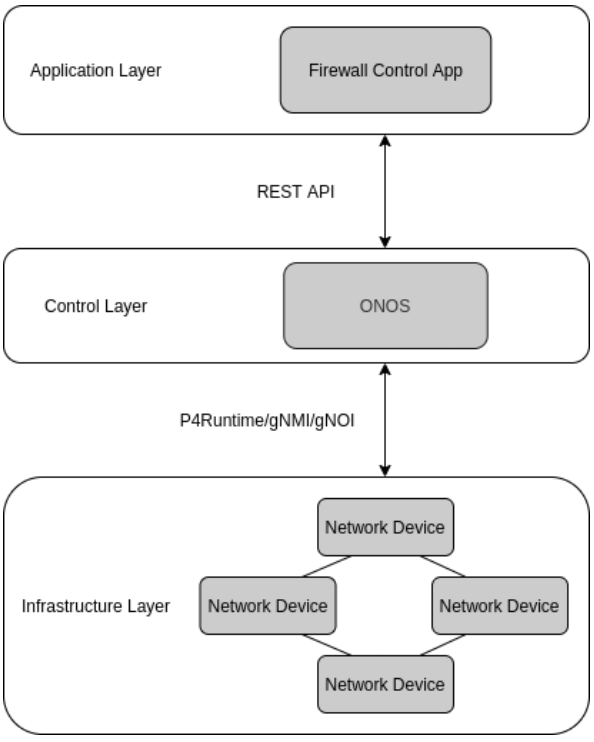


Figure 3.3: P4 Firewall architecture

[4] `https://reactjs.org/`

In Figures 3.4 and 3.5 are depicted the UI used to control the devices and its pipeline. Such fields must match the ones that are in table firewall_filtering_table in Appendix A.8. With even simple user interfaces as the ones in Figures 3.4 and 3.5, network

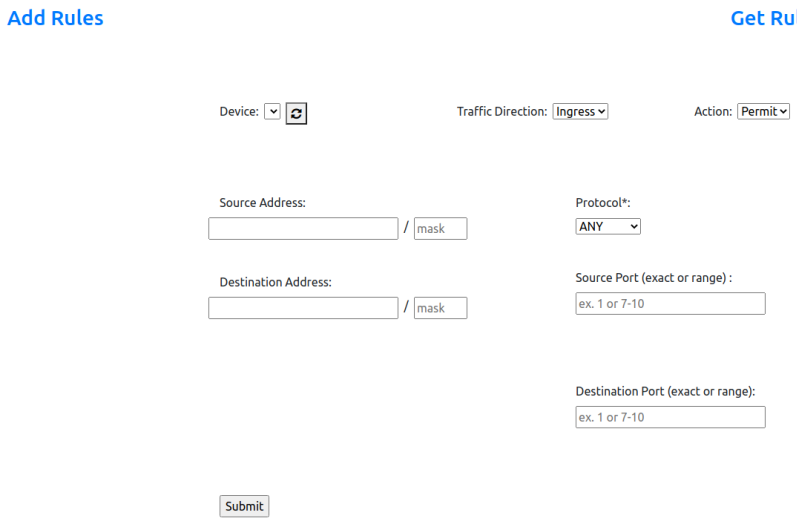


Figure 3.4: Firewall Contro App - Add rules UI

managers can easily get access to all the of the dataplane information, on a human readable way. For example, in this particular case, if one wants to delete a specific rule it just needs to get to the line where that rule is presented and click the delete button, which is simple and can reduce the number of errors when using a CLI. Also, one key point is that different network devices controlled by the same entity can have different pipelines. Having different external control apps connected to the controller can ease the management of such devices through access groups.

To conclude, whenever the process of pushing rules to devices is not dynamically calculated, as in the typical case of a firewall, it is worth to have a UI for users to control the network behaviour. The next sections present a specific use case, not yet explored, using the concepts validated with the experimental setup described up to this point.

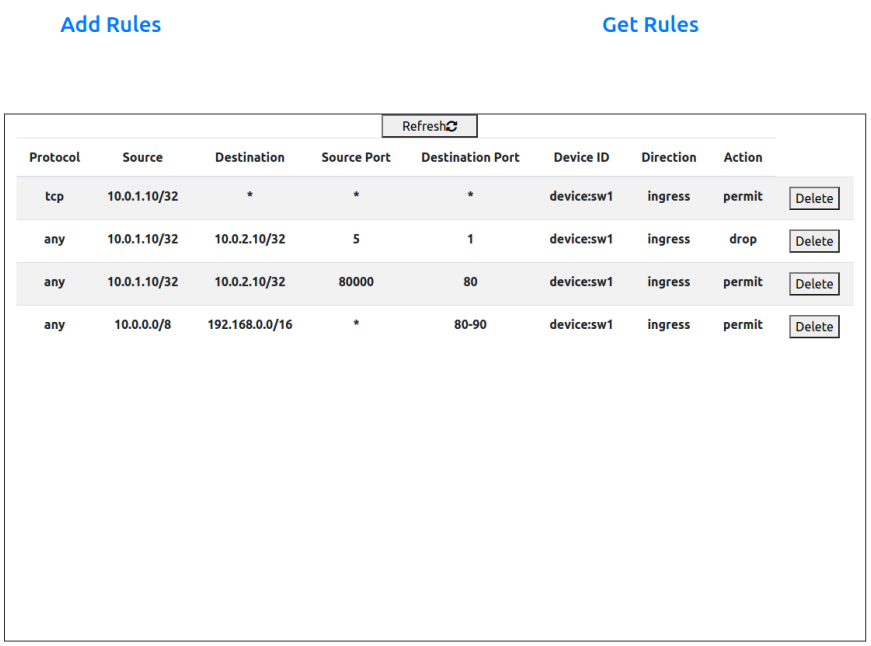


Figure 3.5: Firewall Contro App - Get/Delete Rules UI

3.3 Publish-Subscribe

Publish-Subscribe messaging is a type of asynchronous service-to-service communication. Asynchronous communications, as opposed to synchronous communications, do not block the requester until the response of the required arrives. Such model can be used to enable event-driven architectures, or to decouple applications in order to increase performance, reliability and scalability. One key feature of such architecture is that message exchange is decoupled and anonymous. That is, publishers neither know subscribers' identities nor whether any subscribers with matching interests exist at all. They all rely on a centralized broker/queue manager to forward the messages as depicted in Figure 3.6. Such systems can be divided into 3 major categories [36]. Topic-based, depicted in Figure 3.6, where messages are associated with topics by the publishers and are selectively routed by the message broker to destinations with matching topic interests expressed by the subscribers. Content-based, where subscribers define a set of filters that specify its interests making reference to publication message content. The messages are only delivered if the message content matches the constraints defined by the subscriber. Finally, Type-based, where publications are instances of application-

defined types and subscriptions express interest in receiving publications of a specified type or sub-type. It provides guarantees such as type safety and encapsulation.

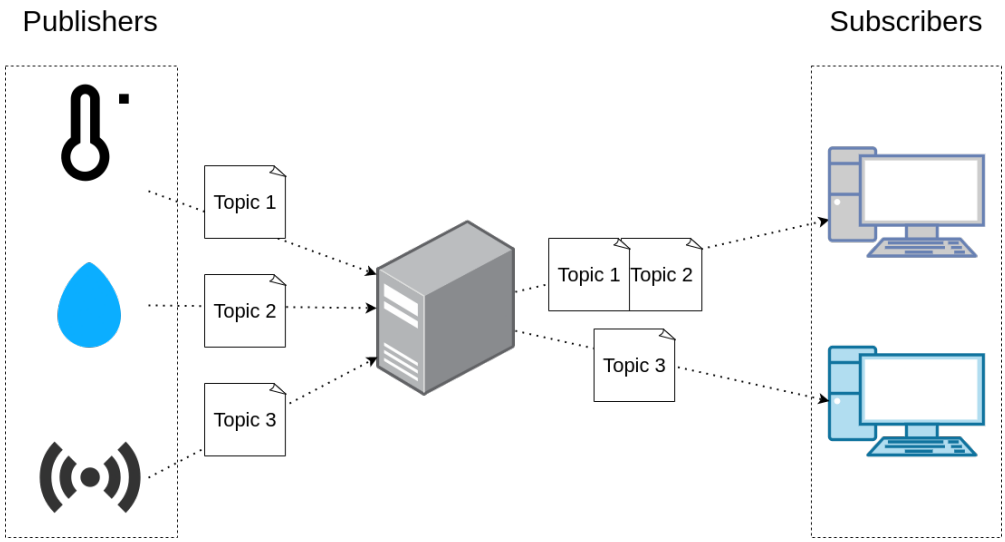


Figure 3.6: Publish-Subscribe Overall Architecture

Regarding P4 implementations of publish-subscribe, [37] presented a work that targeted content-based publish-subscribe systems. The aim of the work is to reduce the header overhead through its encoding. The work however, fails to compare the developed system to traditional systems and so conclusions cannot be taken.

3.4 Offloading

In Table 2.3 were mentioned all of the areas targeted by researchers to build with P4. In this Section, one is going to dig deep into offloading related work, given its importance to the overall objective of this work.
Offloading, in computer systems, means that the responsible devices for heavy computational tasks are changed to more powerful ones. They may be hardware accelerators (e.g., GPUs, FPGAs), servers, supercomputing resources or cloud computing infrastructures [38]. That leads to a more efficient power management, fewer storage requirements and higher application performance [39]. SDN is an example on function offloading. While in traditional networks, network equipment were partially responsible for all the task in the network, with SDN, several jobs were migrated to software

running on high performance servers.
P4 ecosystem, however, allows to bring back such functions to network devices without losing software-only related features, such as DevOps. Several work as already been done regarding offloading tasks to the dataplane.
Regarding Load Balancing, SilkRoad [40] implements stateful load balancing on P4 hardware switches. It leverages the increasing SRAM sizes and stores per-connection states at ASICs. It inherits all the benefits of highspeed low-cost ASICs such as high throughput, low latency and jitter, and better performance isolation, while ensuring per-connection consistency during Dynamic IP(DIP) pool changes. The work however did not provide any metric comparison to known software or OpenFlow load balancers. [41] developed several load balancer mechanisms using P4, such as Connection Hash, Round Robin, Weight Round Robin and Random Connection. The setup was evaluated for each one of the different mechanisms and the authors plan in the future to provide a comparison to both software and commercial hardware load balancers such as Array and F5.
Publish-Subscribe systems, as mentioned before, typically rely on a centralized broker for message forwarding. However P4 eliminates the need of such entity. [42] presented a custom network protocol for publish-subscribe systems implemented in P4 and the goal is to eliminate the need of a central unit, since the traffic is forwarded based on distribution trees, which eliminates a potential bottleneck. The protocol is built directly on top of the Ethernet header and aims to make all forwarding decisions on the dataplane without the need of an application broker and without being bound to specific network protocols of the TCP/IP stack. The authors use distribution trees embedded in the packet for forwarding and were able to achieve an higher performance than traditional forwarding mechanism such as unicast, application-layer multicast, broadcast, among others. Their solution was able to reduce bytes and packets in the network while decreasing the end-to-end delay. Even though the work claims that such solution is better than existing ones, custom protocols lack of validation.
Most of the P4 implementations of firewalls rely on the same concept presented above, that is, block or allow traffic based on packet fields. [43] however, implements an efficient stateful firewall in P4. Stateful firewalls are known to fix the limitations of stateless firewalls. While the former tracks all the network state, the latter uses packet filtering based on header fields, although, the former is more likely to be error prone and achieve poorer performance [44]. The developed system, however, achieved the same performance for the both firewalls with the stateful firewall implementing state recording, detection, and integration. Therefore enhancing the capabilities of P4.

3.5 Zenoh

3.5.1 Overview

As described before, there are already a lot of work done with offloading tasks to the dataplane, namely traffic offloading and firewalls. There are also work developed that moves away from the TCP/IP stack at the dataplane, with a lot of work being developed in the NDN area, in which Zenoh is based. This work, on the other hand, aims to merge these two features and offload tasks to the dataplane through an unspecified protocol. It is accomplished through the description of the Zenoh protocol at the P4 pipeline. The main objective is to compare it to traditional solutions and conclude if it is worth to implement publish-subscribe systems at the dataplane.

Zenoh[5] [9] is a novel unspecified topic based publish-subscribe protocol mostly targeted for IoT. It aims to blend traditional publish-subscribe with geo distributed storage, queries and computations while offering low latency and high throughput, something that could be leveraged even more using P4.

While traditional publish-subscribe protocols are designed to run over TCP, mainly for reliability purposes, Zenoh is able to run over different transportation protocols of the network stack. It gives more flexibility to the users since it can work over UDP, TCP, QUIC or TLS. Due to the different transport link protocols, Zenoh is designed to implement reliability on its own. It uses embedded sequence number in packets in a way that message recipients can verify if the packet is expected or not, and inform the sender of the last sequence number received if it uses reliable channels. Unlike traditional publish-subscribe protocols where a centralized broker must exist, Zenoh works more like a decentralized broker, where every Zenoh Router is aware of the network topology, computed through the Bellman Ford Algorithm, and forward packets based on that.

Another major difference between Zenoh and other known protocols is the way it implements numbers in the packets. While other protocols use length bytes for systems to know how many bytes need to be parsed, Zenoh uses a special format to represent numbers. Numbers are defined by a set of up to 10 bytes, stored in the little-endian notation, with the last byte not bigger than 0x7F. For example a number represented as 0x81 0x93 0xB8 0x40 in the packet, must be processed as 0x40 0xB8 0x93 0x81. That way, the parsing of numbers of the Zenoh protocol is defined by a loop that iterates over several bytes, up to a maximum of 10, until it finds a byte with the most significant bit equal to 0.

[5] `https://zenoh.io/`

At the time of writing Zenoh is still a work in progress and the specification followed is available at `https://github.com/eclipse-zenoh/zenoh/blob/branch_0.5.0-beta.8/zenoh/src/net/protocol/proto/msg.rs`. However, not all of the Zenoh features were implemented but only the simple publish-subscribe packet exchange.
Zenoh has a very straightforward description. Packet headers always start with a format like the one depicted in Figure 3.7 and there can be packet types inside another packet types, for example, Data and Declare packets are embedded into Frame packets. Not all flags are used at times.

Flag 1 (1 bit)	Flag 2 (1 bit)	Flag 3 (1 bit)	MessageCode (5 bit)

Figure 3.7: Zenoh Header Format

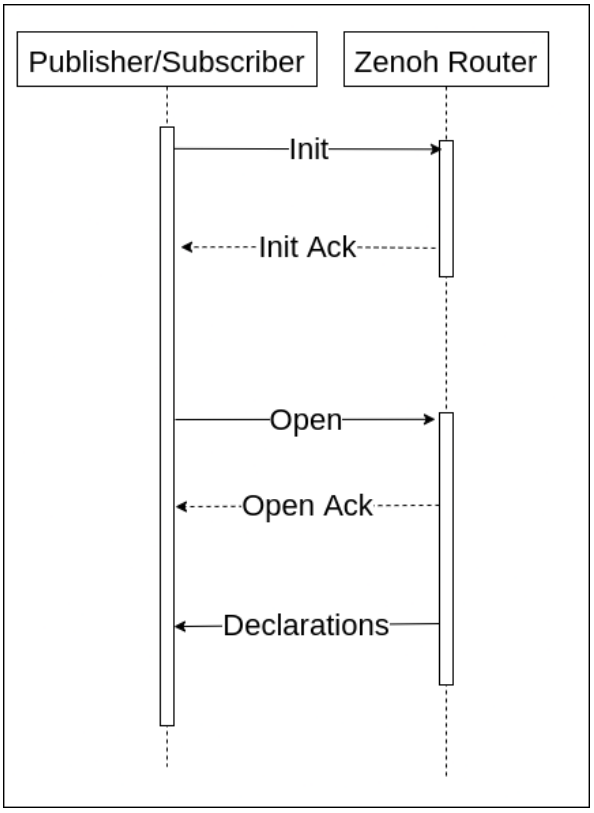


Figure 3.8: Zenoh Session Message Exchange

Zenoh message exchanges can be divided into two categories, session and others. Session message exchange depicted in Figure 3.8 assume that parties are aware of one

another, that is, there is no need for scouting messages. Session establishment process is equal for publishers and subscribers.
The message exchange starts with an Init message sent to the Zenoh Router. That message has information about the type of the sender, Client in this case, and the peer Id. The Zenoh Router should respond to that message with a Init Ack, which has the same format as the Init message but with the acknowledgment flag set to 1. Up to this point there is no stored information on the Zenoh Router, which is way to limit the effect of DoS and DDoS attacks. The next step is to send an Open message to the Zenoh Router, that message must contain a cookie that it is sent by the Zenoh Router in the Init Ack message. Along with that, the sender must also indicate the first sequence number to be used to send messages to the Zenoh Router. In response to that the Zenoh Router sends an Open Ack message that is also the Open message with the acknowledge flag set to 1. The router must also send the first sequence number with which will start to send messages. A session can only be considered established after the Open Ack response by the Zenoh Router is received. After the Open Ack, the router sends all the know resources, publishers, subscribers, etc, in the form of Declaration messages.

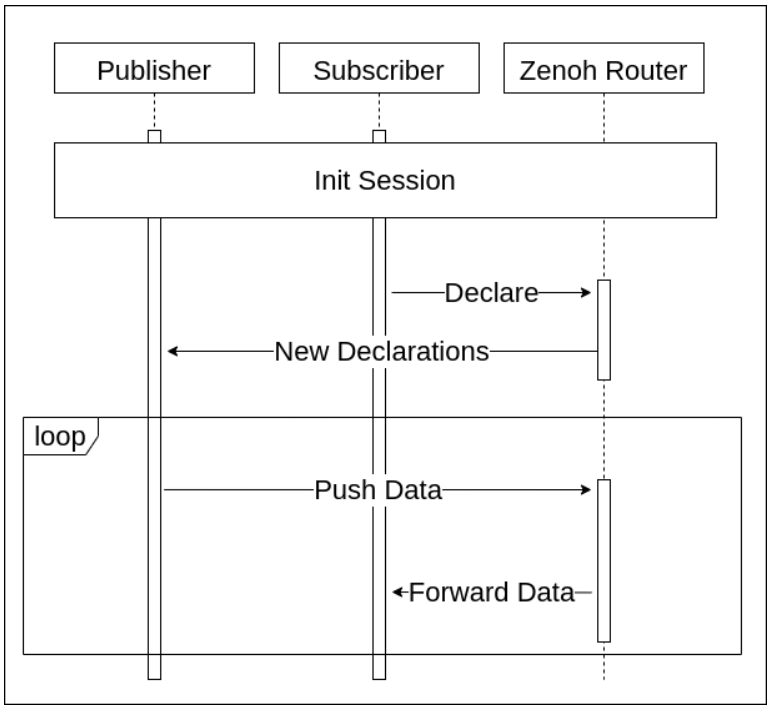


Figure 3.9: Zenoh Data Packet Exchange

Subscribers advertise they want to subscribe a given topic. If that topic is already defined at the message received by the router in the session establishment, the subscriber

declares that it wants to subscribe that topic, represented by an numeric ID. Otherwise, the subscriber must first declare a resource, which is represented by the topic name and should have the format /x/y/z, and then it must declare that it wants to subscribe such resource. On new declarations, the router sends to all known publishers and subscribers the new resources, subscribers, etc. Zenoh publishers never advertise they want to send data to a specific topic. Instead, they wait for the Zenoh Routers, the set of equipment that is the system broker, to advertise that there are subscribers expecting data for a given topic. If they are designed to send data to that topic, then they start to send such data. That process is depicted in Figure 3.9.

3.6 Scenarios

3.6.1 Solution Overview

The main objective of this work is to evaluate the benefits of offloading tasks to the

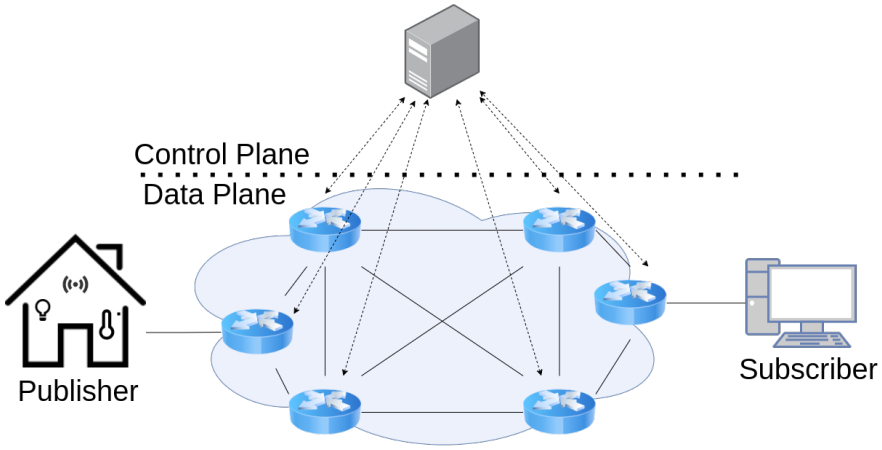


Figure 3.10: Network Topology

dataplane. The first step consisted of choosing what task should be or are feasible to offload. As mentioned, Zenoh is a publish-subscribe protocol, therefore, the forwarding of packets to subscribers is a perfect candidate task to offload. First it was decided to test the system with one publisher and one subscriber on a given topology. Such topology is depicted in Figure 3.10. Analysing the links connecting the switches, it is clear that the shortest path between the publisher and the subscriber is composed of

4 different equipment and 5 links.
Three scenarios then emerged. The first, with no offloading at all, where all packets that reach the switches and have UDP source or destination port equal to 7447, must be sent to the network controller. There the software modules in control of the network behaviour handle the packets. On the other hand, it also exist a scenario with offloading, only the first data packet is sent to the controller, for packet control and flow rule management. All of the other data packets are processed at the dataplane level. Finally, the last scenario, the entity that controls what happens in the network is a pure Zenoh Router. In that case, BMv2 only acts as a L2 switch and it is up to the Zenoh Router to process Zenoh packets.
The three scenarios have different implementation complexity. The scenario solely based on the pure Zenoh Router requires the least amount of development. It makes use of available Zenoh Python3 client libraries and the available Zenoh Router code for binary compilation. The P4 pipeline is rather simple with a low number of parsing stages, with parsing up to the IPv4 header, and a reduced number of actions. The scenario without offloading makes use of the same Zenoh clients, but it requires a pipeline with a parser that goes up to the UDP header. Also, the modules added to the network controller are much more complex with tasks such as network link monitoring and topology computation and a packet processor module responsible for handling Zenoh packets that arrive at the CPU. Lastly, the offloading case, the most complex of all three, has the deepest and more complex pipeline, given the fact that needs to parse Zenoh Packets. The modules added to the network controller are similar to the ones on the scenario without offloading, but simpler, given the fact that data forwarding is offloaded to the dataplane.
Regarding the offloading and without offloading scenarios, the session exchange is the same, as it is the network controller that process such messages. The process is depicted in Figure 3.11 and it is similar to the traditional Zenoh session establishment depicted in Figure 3.8.

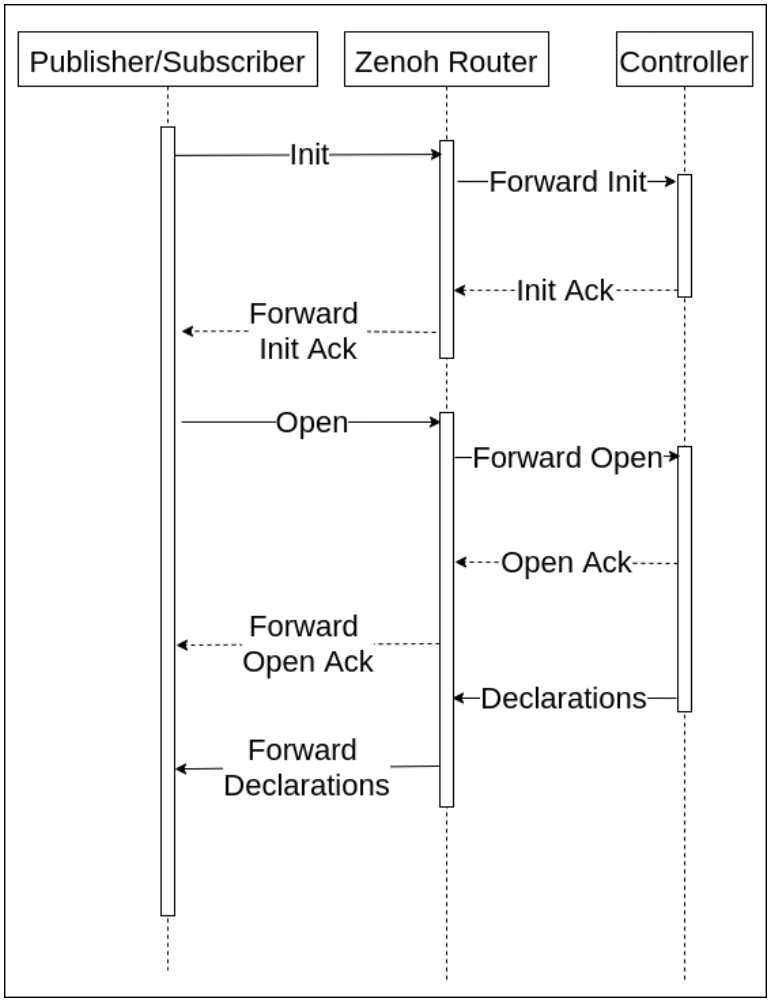


Figure 3.11: Session establishment process

As expected, the overall message exchange other than session messages, will be similar to the one depicted at Figure 3.9 but, in this case, different on both scenarios. While one needs to send all the messages to the controller in order to be processed, the other does that job directly on the dataplane. In Figure 3.12 is displayed the overall publish-subscribe process for the offloading scenario and in Figure 3.13 is depicted the process for the scenario without offloading.

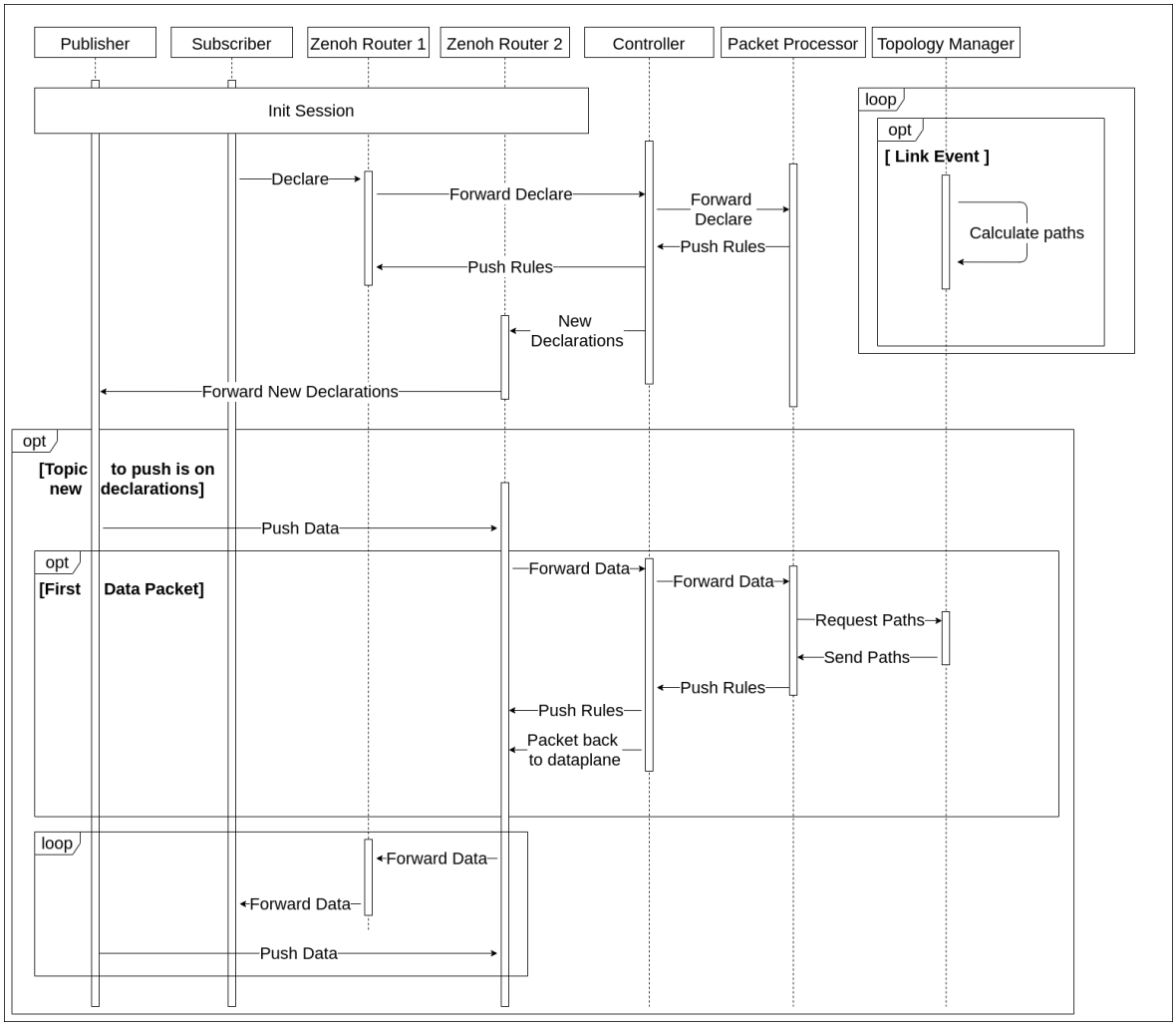


Figure 3.12: Pub sub process with offloading

The differences on those scenarios start with the data packets, which can be seen at the bottom part of the Figures 3.12 and 3.13. In the offloading case, only the first data packet, regarding a specific topic that arrives on the network equipment, is sent to the controller. That is used for processing and installation of rules in the target devices. All other data packets that arrive on the device with the same topic, even from different publishers, are processed directly on the data plane. However, for the case where no offloading exists, every single packet must be sent to the controller for processing, where a specific module is responsible for processing and creation of all the needed packets.

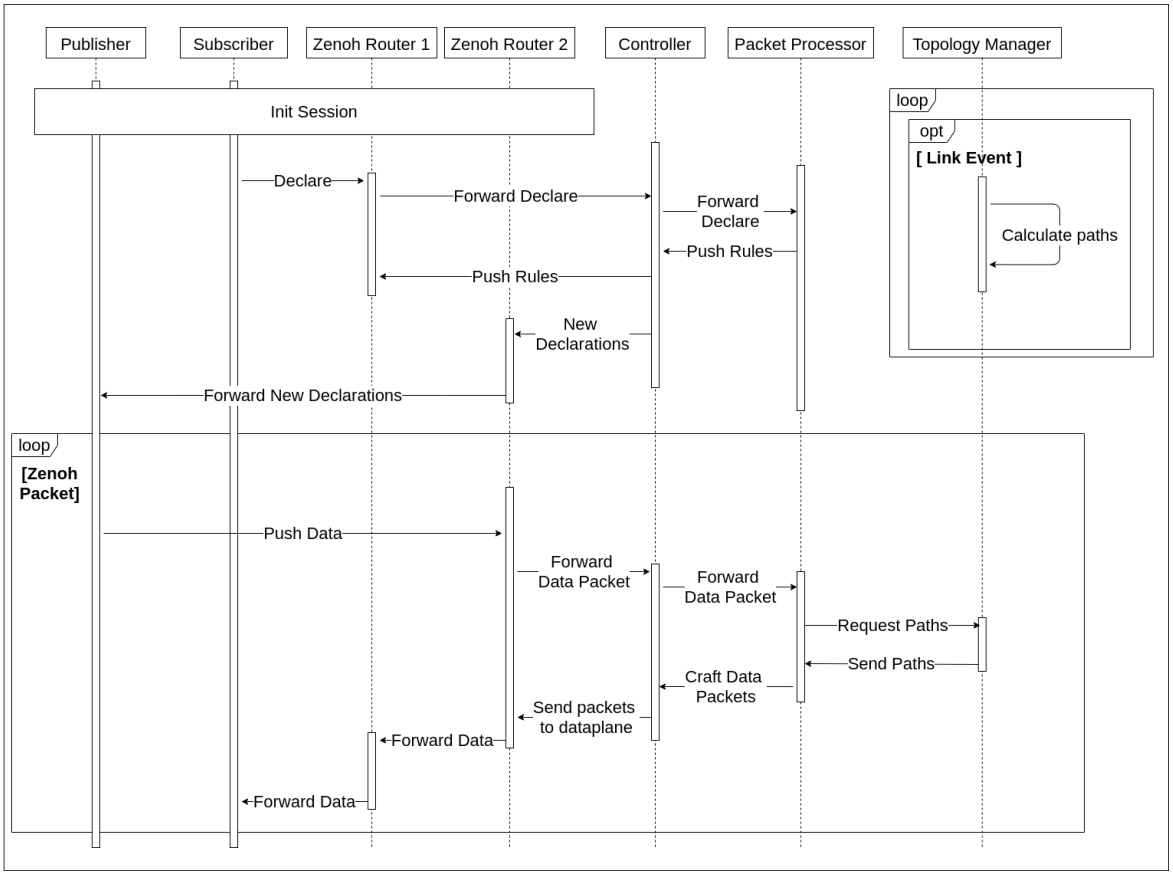


Figure 3.13: Pub sub process with no offloading

Zenoh hosts, either publishers or subscribers, send and must receive KeepAlive messages. While KeepAlive sent by the hosts is automatic by the Zenoh Client library, the KeepAlive sent to the hosts must be processed by the system. In order to do that, it was also developed a module, that at every 3 seconds, checks which host are on, that is, are sending KeepAlive messages within an interval of time. For every host that is on, the module builds a KeepAlive message and sends it to that host.

3.6.2 Shortest Path Algorithm

The routing in Zenoh is implemented using the Bellman Ford Algorithm. However SDN allows developers to implement their own algorithms to solve Single-Source Shortest Path Problem (SSSPP). Multiple algorithms can be applied, such as Dijkstra's Algorithm (DA), Bellman Ford Algorithm (BFA), Prim's Algorithm (PA), Depth-First

Search (DFS) or Breadth-First Search (BFS). DFS and BFS should only be used when no weights are assigned to the edges. Given the fact that network links most often have weights attached, these two algorithms were immediately discarded. The other three algorithms differ in various ways. PA and BFA can work with negative weighed edges while DA cannot. Traditionally network links have positive weight values, overruling the benefits brought by BFA. Also, DA and PA are greedy algorithms, with time complexity $\mathcal{O}(E \log V)$, while BFA, which is not greedy, have a time complexity of $\mathcal{O}(E.V)$. At this point, both PA and DA are tied. However, it was decided to go with DA due to the nature of PA. While DA aims to compute the shortest path from one node to all other nodes, PA aims to create the Minimum Spanning Tree between all nodes in the network, which is prone to failure when computing shortest paths.

Algorithm 1 Dijkstra's algorithm

```
1:  for Devices in AvailableDevices do
2:    ProcessedNodeSet ← new Set();
3:    QueueDevicesToProcess ← new Queue();
4:    while QueueDevicesToProcess != Ø do
5:      dev ← QueueDevicesToProcess.head();
6:      for Link in dev.EgressLinks() do
7:        if LinkDstDevice not in ProcessedNodeSet then
8:          if DevPathCost + LinkCost < DstDevPathCost then
9:            DstDevPathCost ← DevPathCost + LinkCost;
10:           UpdatePathStored();
11:         end if
12:       end if
13:     end for
14:     ProcessNodeSet.add(dev);
15:   end while
16: end for
```

It was already mentioned that internally, Zenoh uses the BFA to compute paths. However, with the use of SDN tools, one is not bounded to use that same algorithm, just as in this case, other algorithms can be used and the system works the same way, or even better. That also shows that developers have a degree of freedom when working in the implementation of features using software.

3.6.3 Offloading Pipeline Description

Having the desired behaviour defined it is time to start building the pipeline for the offloading scenario. Again, as in the experiment of the previous chapter, the overall pipeline is based on the fabric.p4 offered by ONF. However in this case not only the

ingress but also the egress pipeline needed to be extended.
Zenoh can work with reliable or best-effort channels. Reliable channels guarantee the deliver of data while best-effort channels does not. Therefore packet losses can occur depending on the traffic load. Consider to this case the best-effort channel.
Depicted in Figure 3.14 is the description of ingress pipeline.

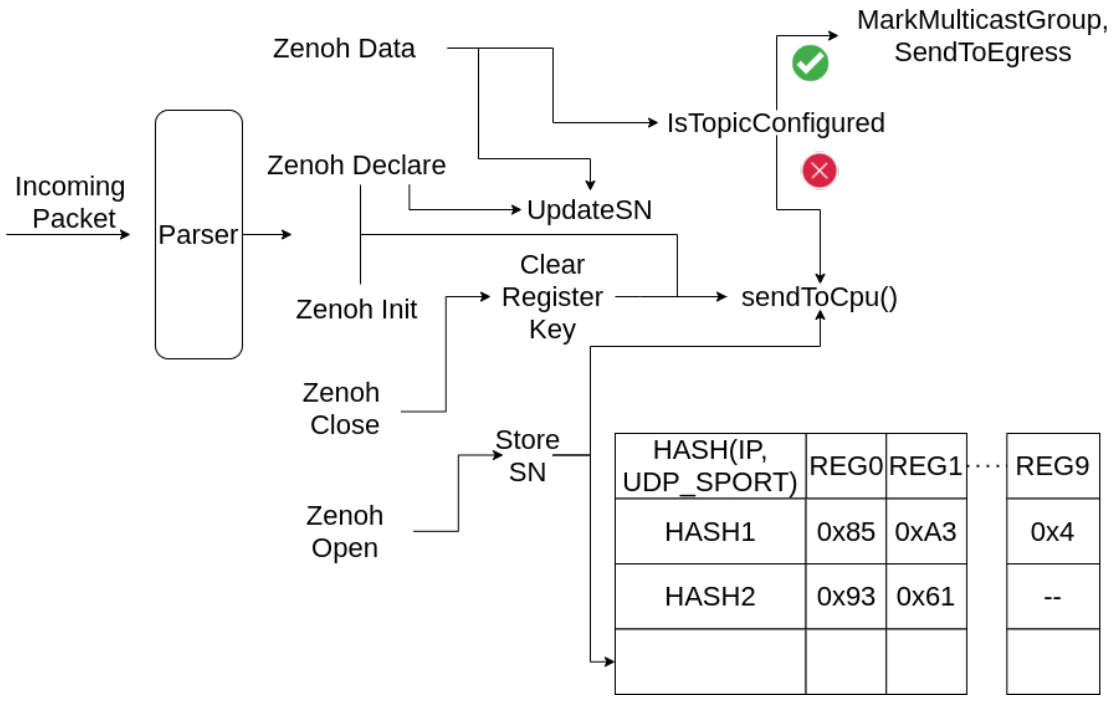


Figure 3.14: Ingress Pipeline

All packets, other than data packets, need to be sent to the controller, however, some may need tasks to be performed at the dataplane level before that occurs.
Init packets are sent straight away to the controller which is responsible for the creation of the respective Init Ack packet.
Open packets, before being sent to the controller, must be processed for the storage of the first sequence number. Even though a best-effort channel is used, all of the incoming packets with sequence numbers must be compared to the stored ones, in case it matches, processing proceeds, otherwise it is updated. If a reliable channel is to be used in case of a miss match the packet can be sent to the controller. The packet sequence number is stored using up to 10 registers. Such registers are bloomfilters with a maximum of $2^{16} - 1$ entries where each entry allocates 1 byte only. The key used to access the specific bloom filter position is computed through an hash of the source

IPv4 and UDP source port. Only after that, the packet is available to be sent to the controller.
Declare Packets are to be computed by the controller, for subscriber and resource management. However, before it is sent to the CPU it must update the sequence number of the sender, stored in the registers.
Data packets are the task that is to be offloaded to the dataplane, so, on a packet arrival two thing can occur. The topic is either configured or not. That information is stored on a bloom filter whose key is the topic number and value is 1 bit, 1 if configured, 0 otherwise. If the topic is not configured it must be sent to the controller. There, the controller will gather all the subscribers for that topic and fetch the available paths for each one of them. After that, it will install rules on the target devices, based on the fetched paths, so that subscribers can receive the desired packets. Finally the packet is sent back to the dataplane as it was received. That will go through the same process as before but this time the topic is considered as configured and therefore the packet will be marked as a multicast packet and be sent to the egress pipeline. This time however, the sequence number is not updated as it was already when it first arrived at the switch.
Close packets are used to clear the sequence number of the sender. Register values, pointed by the user key, are wiped and put to 0. After that, the packet is sent to the controller for user deletion at the control plane.
In the case of Open and Data packets, the egress pipeline is complementary to the ingress one. While every other packet sent by the controller is sent to the deparser straight away, these packet types must be processed. In Figure 3.15 is depicted the egress pipeline behaviour. After the Open packet is sent to the CPU, the controller must respond with an Open Ack packet. That Open Ack packet has the first sequence number to be used to connect to the end user, therefore it must be stored on the egress pipeline. The data structure used to store it, is the same as the used for the storage of sequence numbers in the ingress pipeline, however, in this case, the key used to accesss the desired position is the destination IPv4 and UDP destination port, which in fact will have the same value as the key used in the ingress pipeline. The packet is then sent to the desired egress port.

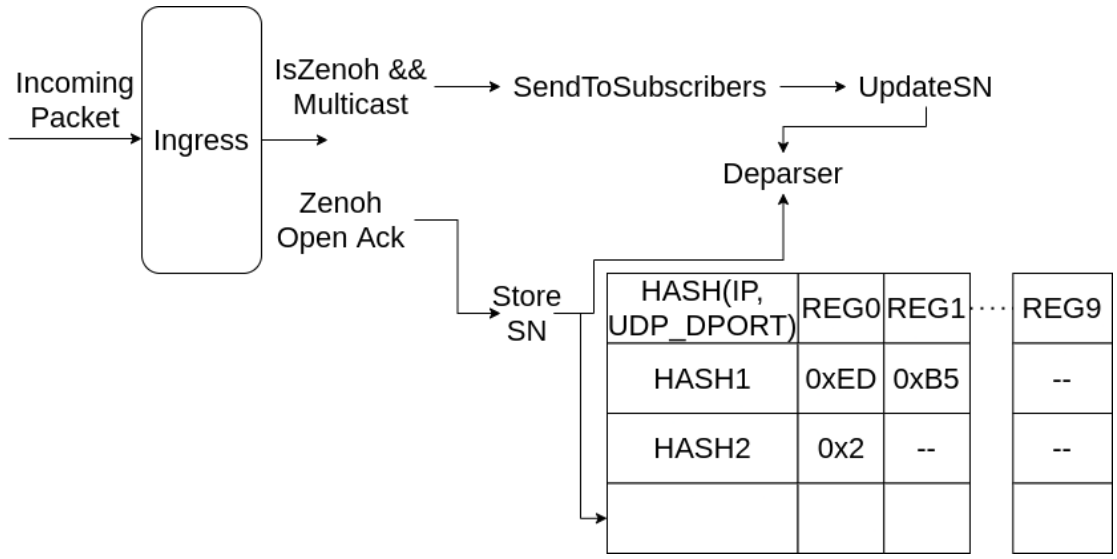


Figure 3.15: Egress Pipeline

Data packets are different from other packets due to the use of the replication engine, i.e., one packet on the ingress can lead to n packets at the egress pipeline. Such behaviour is managed by the control plane on declare subscriber packets arrival. The controller must create or update the replication group of the desired device. Such replication group will be used to replicate the input packet for n egress ports. Such egress ports, which are the access point for a number of end hosts, along with the multicast group, which is nothing but the topic number, will be used to filter through a P4 table. In case it matches, the packet Ethernet, IPv4 and UDP is changed to meet the target requirements, which can be either the next switch or a subscriber.
Close packets, not shown in the figure, are sent back to dataplane by the controller after processing. That is used to also clear the sequence numbers used for that end host. After the deletion of the sequence number, the packet is marked to drop.

3.7 Evaluation

The system was evaluated using the experimental setup described in the previous chapter, however the Mininet's base container[6] used, developed by ONF, which was chosen for its stratum support, had some limitations regarding Zenoh Python clients and Zenoh itself. Zenoh makes available several clients, both in Python and Rust, being the latter the base for the former. Since the chosen mininet container did not

[6] `https://hub.docker.com/r/opennetworking/mn-stratum`

have Rust support, such container had to be changed and some additions needed to be implemented, such as the installation of Rust and Python version 3.9. Dockerfile with such additions is available at Appendix B.1. After all the requirements installed in the container, Zenoh client libraries could be used and the system tested and evaluated.

All of the tests were made with an intel i5 quad-core 1.6GHz CPU and 8GB RAM using the topology presented in Figure 3.10. The tests consisted of measuring packet departure at the publisher, and packet arrival at the subscriber and collect measures such as latency, jitter and the number of packets lost. The results for each one of the scenarios are presented in Table 3.1. The results show that there is a benefit of

Table 3.1: Topology Results

Case	**Latency(s)**	**Jitter(s)**	**Packets Lost**
Offloading	0,055 ± 0,0005	0,013 ± 0,0003	0
No Offloading	0,081 ± 0,0028	0,032 ± 0,0020	10,8 ± 2,16
Zenoh Router	0,024 ± 0,0001	0,003 ± 0,0002	0

offloading data forwarding to the dataplane regarding with and without offloading scenarios. However, the pure Zenoh Router achieved a much better performance both in latency and jitter in comparison to the offloading case. Due to that, it was decided to conduct more tests. Such tests consisted in the variation of the switches number between publishers and subscribers and also the variation in the number of publishers and subscribers to evaluate how a different number of data streams impact the system. Given time constraints and limited computational resources the combinations of publishers and subscribers needed to be restricted to the ones presented in Table 3.2. Each one of those combinations was tested with a different number of switches between end hosts. The number of switches started at one and was increased one up to a maximum of five.

Combination	**nº Publishers**	**nº Subscribers**
C1	1	1
C2	1	5
C3	5	1
C4	5	5

Table 3.2: Publisher Subscriber Combinations

3.7.1 Latency

The overall latency results presented in Figure 3.16 follow the results obtained with the given topology. The pure Zenoh Router scenario performance, is always better than the other two scenarios. The offloading case still achieves better latency results than without offloading in all the tests.

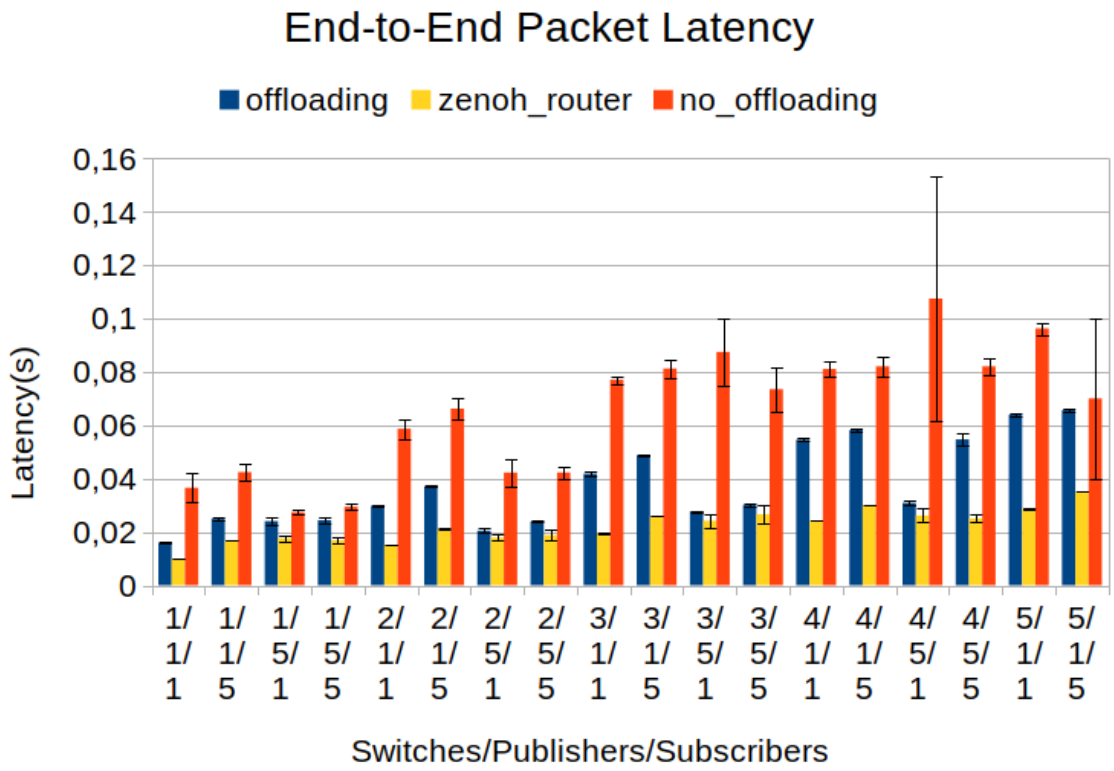


Figure 3.16: Latency Results

Analysing the previous figure, one conclusion comes to mind. All of the scenarios perform better on cases with an higher number of publishers, which means that there is no optimization when only 1 publisher exists.

3.7.2 Jitter

As expected the jitter results, presented at Figure 3.17 follow the results obtained for latency. Again, the confidence intervals achieved with the scenario without offloading are much less precise than of the other two. Although, in this case, the conclusions taken in the analysis of the latency results are not so straightforward. While the scenario without offloading still maintains the same pattern, in the other two scenarios

the jitter is typically higher the more packets are flying in the network. However, one thing is certain, the results achieved for 1 publisher and 5 subscribers are higher than the results for 5 publishers and 1 subscriber.

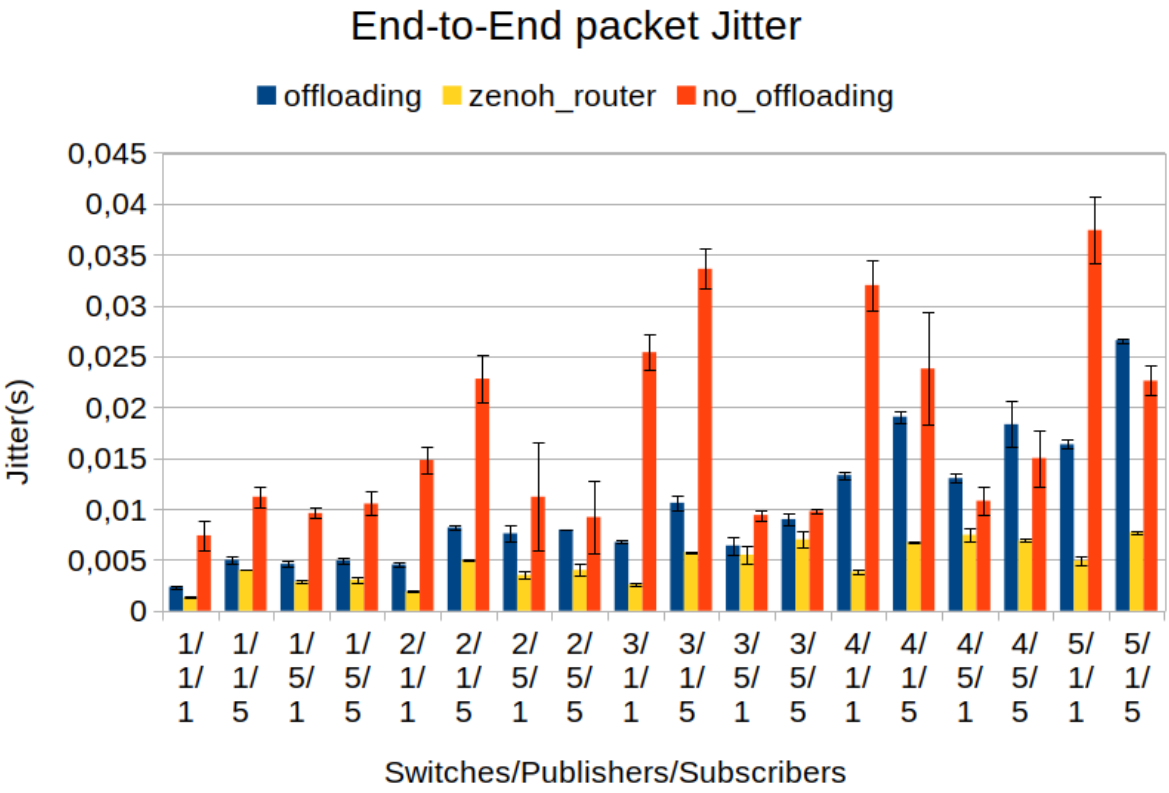


Figure 3.17: Jitter Results

3.7.3 Packets Lost

Given the usage of best-effort channels, packet losses can occur. The scenario without offloading is much more CPU intensive than the other two, which may be the reason why so many packets are lost in this scenario, even for a simple topology with only 1 publisher and 1 subscriber. The more equipment exist the more packets will arrive at the controller, even if the number of publisher and subscribers, stay the same. The overall results are depicted in Figure 3.18. It is clear that both the offloading and the pure Zenoh Router scenario did not have any packet losses. However, one cannot guarantee that in a real world scenario packet losses will not occur, given all the variables that influence it.

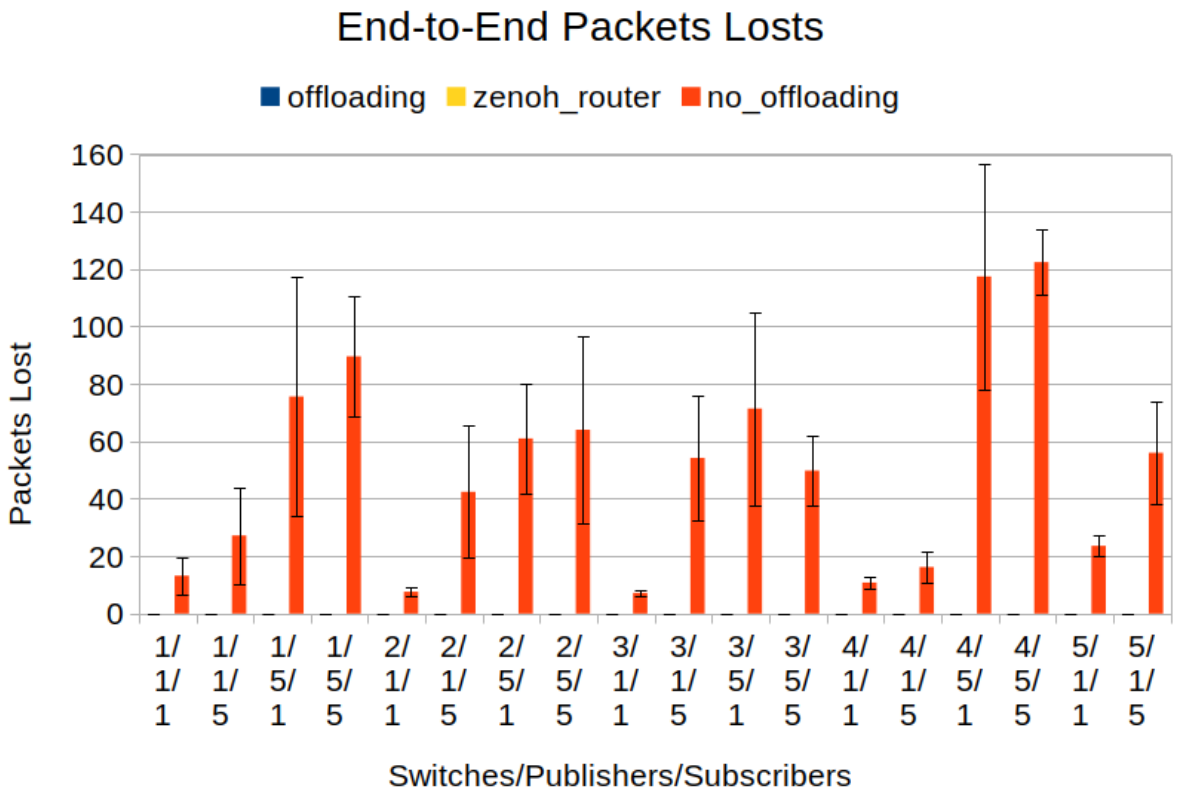


Figure 3.18: Lost Packets Results

3.7.4 Discussion

Given the results obtained it is clear that when there is only one publisher results are worser than with five publishers. That may be due to the usage of the multi-stage pipeline. In the case where there is only one publisher there is not a full usage of the benefits brought by the multi-stage, whereas with an higher number of publishers the parsing stages are executed in parallel and therefore reduce the overall latency and all of the metrics that derive from that.

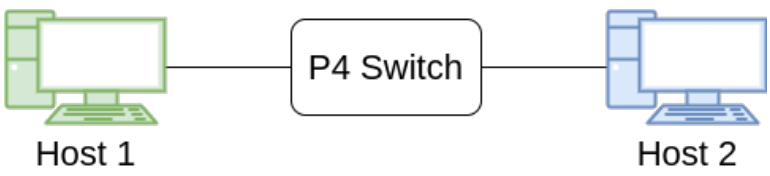


Figure 3.19: Performance test architecture

Due to the known limitations of BMv2, some preliminary tests were made to evaluate

how much the performance is affected by the number of parsing stages. It was carried an experiment, with the topology depicted in Figure 3.19 where the number of parsing stages varied from 3 to 7. Depicted in Figure 3.20 are the results obtained with the mininet iperf command. The results validate the loss of BMv2 performance the more parsing stages exist. In this case, from 3 to 7 parsing stages, the throughput dropped from 30 Mb/s to almost 15 Mb/s.

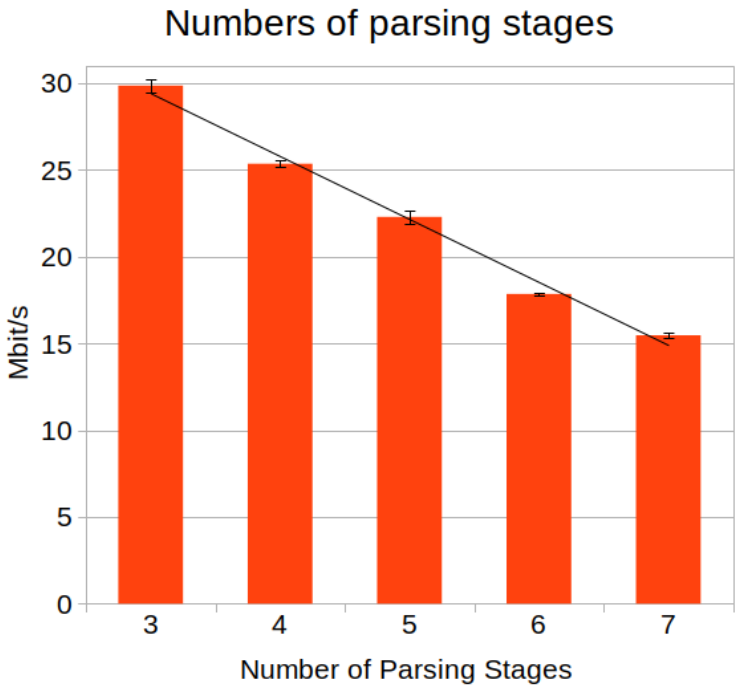


Figure 3.20: Effect of the number of parsing stages in the bmv2 performance

The results presented in Figure 3.20 can help to explain the overall results obtained and why the pure Zenoh scenario performed better than the offloading scenario. In Table 3.3 are summarized the number of parsing stages of each of the conducted scenarios. The carried experience with offloading had an average of 19 parsing stages, given by the average length of the Zenoh Clients sequence numbers of 4, in opposition to the pure Zenoh case where the BMv2 had an average of 4 parsing stages. That alone could be a reason why the scenario with offloading has worse performance than the pure Zenoh scenario. Another reason that may complement the previous one is the fact that, Zenoh is fully targeted for x86 architectures while BMv2 is only meant to be used on x86 architectures to reproduce the behavior of a P4-programmable ASIC or NIC, but it's targeted for testing and debugging and it's not a production grade switch.

Scenario	Parsing Stages
Offloading	≈ 19
No Offloading	5
Pure Zenoh Router	4

Table 3.3: Comparison between number of parsing stages in each scenario

3.8 Summary

This chapter presented the overall objective of a experimental setup and the technologies used to accomplish the development of such setup. It started by presenting what docker is, why is it used and its benefits. It was followed by what mininet is and what is mainly used for, specifically in this case and the benefits of using mininet to develop using the P4 ecosystem. It also presented the version of the ONOS chosen and why was that specific version elected, which was the support of the Segment Routing App, that offers ICMP and ARP handling and routing through MPLS, out of the box.
This chapter also showed how CI/CD cycles can be integrated with programmable networks for behaviour testing, with open-source tools such as PTF. It finished with an example of an app that was developed to control network device tables. Such solution shows that network managers can create their own tools, connect it to open source solutions and manage the network in an easy way through a GUI.
It was followed by how publish-subscribe systems work and what offloading is. Then, it presented Zenoh, a novel publish-subscribe protocol, which was used to fulfill the objective of this work, to show how offloading tasks to the dataplane can be beneficial for a network. After the Zenoh presentation, the overall solution was described, how was Zenoh used and the overall system behaviour. Lastly, the overall results achieved were presented, with an extra section of discussion that presented preliminary tests made to BMv2 that can be useful for the result analysis.
The next Chapter will present the overall conclusions and future directions.

Chapter 4

Conclusions

This work consisted of a P4 implementation of a novel protocol called Zenoh. Such work was used to evaluate the benefits of offloading task to the dataplane, compared to traditional solutions. It started by presenting the traditional SDN solutions and a theoretical approach to the state-of-the-art NG-SDN technologies. It was followed by a conceptual approach to the problem, with the presentation of the architecture and development tools chosen. It finished with the presentation of a specific use case and the overall results obtained for the test scenarios. To conclude, this work contributes to show the flexibility offered by P4 and how offloading tasks to the dataplane can affect the network performance.

4.1 Conclusion

This work aimed to show if there are benefits of offloading tasks to the dataplane. For that, it was decided to use state-of-the-art solutions such as P4. P4 allows developers to deeply describe the network behaviour through a pipeline. It stands from other solutions such as OpenFlow and SAI for being able to be target and protocol independent, that is, it is not limited to the traditional TCP/IP stack and even custom or proprietary protocols can be described by the language and processed by the P4Runtime protocol, the connection point between data and control planes.
Offloading tasks to the dataplane means that, the processing should happen in the dataplane and not in the control plane. Many works have offloaded firewalls and several other task to the dataplane, however, there is limited work with publish-subscribe systems and the ones that exist do not make comparisons to traditional systems. So, it was decided to explore such feature.
Traditional publish-subscribe protocols are only able to run over TCP. However, that is a difficult task to accomplish with P4. P4 can easily process TCP packet but it is hard to manage TCP sessions and everything that is related to it. Zenoh, on the other hand, is designed to run over multiple transport protocols and be implemented with best-effort channels. That makes the process simpler and easily achieved through P4.
The overall system consisted of topology with several switches connected and only 1 publisher and 1 subscriber. Zenoh traditionally uses the Bellman-Ford algorithm to

compute paths, however, SDN tools allow that to be changed and therefore developers have a bigger degree of freedom. In this case, it was decided to use the Dijsktra Algorithm, given there was no benefit of using the Bellman-Ford one. The developed system, with offloading, was tested against 2 other systems. One without offloading and other with a pure Zenoh Router. The pure Zenoh Router performed better in all of the tests, but the offloading scenario outperformed the scenario without offloading, in several cases it was 2x faster to deliver packets. Given the obtained results several other tests were made, with a variation in the number of publishers, subscribers and the switches between end-hosts. The achieved results confirmed the results obtained before. Even though the pure Zenoh Router achieved better performance, that does not mean that the offloading scenario is not better than the pure Zenoh Router but that the chosen target device, which is BMv2, is not designed for production but rather test environments. BMv2 is highly affected by factors such as, for example, the number of parsing stages. In fact, during the tests, the number of parsing stages in the offloading scenario was more than 4x the number of parsing stages of the pure Zenoh Router scenario, which for an equipment so heavily affected by the number of parsing stages, is a big deal. That influence is expected to be irrelevant or none if hardware equipment or a production grade switch is used. However, up to the time of writing hardware equipment is highly expensive and no P4 compatible production grade software switch exist.
To conclude, the overall work showed the flexibility of P4 and how it can be used to offload task to the dataplane. It analysed 3 different scenarios but tests with production-grade targets need to be done in order to take the final conclusions.

4.2 Future Work

As mentioned, Zenoh can work with reliable or best-effort channels. This work was based on best-effort channels, however, if one considers to use reliable channels, there are more tasks that can be offloaded to the dataplane. Zenoh reliability is achieved by sending Ack_Nack packets when sequence numbers received are not the expected ones. That packets must have the last packet sequence number received. If one thinks of this task to be computed at the edge of the network, that is, closest to the end-user as possible, that process can make the information about losses reach the end user much faster, which also reduces the number of useless packets that fly through the network. Also, depending on how Zenoh evolves, Init Ack and Close Ack packets can be created directly in the dataplane, without the controller interference, at least directly. The

controller will always have to process Init and Close packets, however, that does not mean that it needs to be the controller to create the response. The dataplane can just clone the packet to the control plane and at the same time create the response. The control plane will then create or delete the required information in an asynchronous way from the dataplane.

Other thing that can be implemented is the Firewall behaviour for a publish-subscribe system. That is, block of packets from a specific host to a specific topic, or block a host from receiving packets related to a topic. That can be made with an external app connected to the controller northbound API and a few more tables in the pipeline.

This work was based on the results of the software switch BMv2. The obtained results are not good in comparison to other software switches or hardware switches. Therefore, one key step is to test the pipeline in some of the state of the art hardware to really see how P4 can improve network metrics.

References

[1] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. A brief history of the internet. *SIGCOMM Comput. Commun. Rev.*, 39(5):22–31, October 2009.

[2] C.David Mercier and Selina Hembree. What the internet can do for you. *Industry Applications Magazine, IEEE*, 4:8 – 15, 12 1998.

[3] Cisco. Cisco annual internet report (2018–2023). Available: `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf`, March 2020. Accessed on 17-03-2021.

[4] Ali Shakarami, Mostafa Ghobaei-Arani, Mohammad Masdari, and Mehdi Hosseinzadeh. A survey on the computation offloading approaches in mobile edge/cloud computing environment: A stochastic-based perspective. *Journal of Grid Computing*, 18:1–33, 12 2020.

[5] Tao Zheng, Jian Wan, Jilin Zhang, Congfeng Jiang, and Gangyong Jia. A survey of computation offloading in edge computing. pages 1–6, 10 2020.

[6] Paul Göransson, Chuck Black, and Timothy Culver. Chapter 5 - the openflow specification. In Paul Göransson, Chuck Black, and Timothy Culver, editors, *Software Defined Networks (Second Edition)*, pages 89–136. Morgan Kaufmann, Boston, second edition edition, 2017.

[7] Justus Rischke and Hani Salah. Chapter 6 - software-defined networks. In Frank H.P. Fitzek, Fabrizio Granelli, and Patrick Seeling, editors, *Computing in Communication Networks*, pages 107–118. Academic Press, 2020.

[8] Tomasz Osiński, Mateusz Kossakowski, Halina Tarasiuk, and Roland Picard. Offloading data plane functions to the multi-tenant cloud infrastructure using p4. pages 1–6, 09 2019.

[9] Gabriele Baldoni, Julien Loudet, Luca Cominardi, Angelo Corsaro, and Yong He. Facilitating distributed data-flow programming with eclipse zenoh: the erdos case. pages 13–18, 06 2021.

[10] Mamta Agiwal, Hyeyeon Kwon, Seungkeun Park, and Hu Jin. A survey on 4g-5g dual connectivity: Road to 5g implementation. *IEEE Access*, 9:16193–16210, 01 2021.

[11] Jonathan Rodriguez. *Fundamentals of 5G Mobile Networks*. Wiley Telecom, 2014.

[12] Estifanos Mihret and Getamesay Haile. 4g, 5g, 6g, 7g and future mobile technologies. *American Journal of Computer Science and Technology*, 9:75, 02 2021.

[13] Imt-2020 background. Available: `https://www.itu.int/en/ITU-R/study-groups/rsg5/rwp5d/imt-2020/Documents/060R1e.pdf`.

[14] G. N. ETSI. Ts 123 501 - v15.3.0 - 5g; system architecture for the 5g system. Available: `https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/15.03.00_60/ts_123501v150300p.pdf`, September 2018.

[15] G. N. ETSI. Nfv white paper. Available: `https://portal.etsi.org/NFV/NFV_White_Paper.pdf`, October 2012.

[16] G. N. ETSI. Etsi gs nfv 002 v1.2.1 (2014-12). Available: `https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf`, December 2014.

[17] Prithwish Kangsabanik. Oss/bss impact with 5g applications and services. Available: `https://futurenetworks.ieee.org/images/files/pdf/applications/OSS-BSS-impact030518.pdf`, March 2018. [Online].

[18] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, 2010. Association for Computing Machinery.

[19] G. N. ETSI. Etsi gs nfv-eve 005 v1.1.1 (2015-12). Available: `https://www.etsi.org/deliver/etsi_gs/NFV-EVE/001_099/005/01.01.01_60/gs_nfv-eve005v010101p.pdf`, December 2015.

[20] Paul Zanna, Pj Radcliffe, and Karina Gomez Chavez. A method for comparing openflow and p4. In *2019 29th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–3, 2019.

[21] Aliyu Lawal Aliyu, Peter Bull, and Ali Abdallah. Performance implication and analysis of the openflow sdn protocol. In *2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 391–396, 2017.

[22] Markus Brandt, Rahamatullah Khondoker, Ronald Marx, and Kpatcha Bayarou. Security analysis of software defined networking protocols - openflow, of-config and ovsdb. 07 2014.

[23] Davorin Valencic and Vladimir Mateljan. Implementation of netconf protocol. pages 421–430, 05 2019.

[24] Yimeng Zhao, Luigi Iannone, and Michel Riguidel. On the performance of sdn controllers: A reality check. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 79–85, 2015.

[25] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. Distributed sdn control: Survey, taxonomy and challenges. *IEEE Communications Surveys Tutorials*, PP:1–1, 12 2017.

[26] Wojciech Kozlowski, Fernando Kuipers, and Stephanie Wehner. A p4 data plane for the quantum internet. pages 49–51, 12 2020.

[27] M. Melucci. *Introduction to Information Retrieval and Quantum Mechanics*. The Information Retrieval Series. Springer Berlin Heidelberg, 2015.

[28] Microsoft's quantum definition. `https://docs.microsoft.com/en-us/azure/quantum/overview-understanding-quantum-computing`. Accessed: 2021-05-04.

[29] Paul Zanna, Pj Radcliffe, and Dinesh Kumar. Wp4: A p4 programmable ieee 802.11 data plane. pages 1–6, 11 2020.

[30] Brian O'Connor, Yi Tseng, Maximilian Pudelko, Carmelo Cascone, Abhilash Endurthi, You Wang, Alireza Ghaffarkhah, Devjit Gopalpur, Tom Everman, Tomek Madejski, Jim Wanderer, and Amin Vahdat. Using p4 on fixed-pipeline and programmable stratum switches. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–2, 2019.

[31] Paul Zanna, Pj Radcliffe, and Karina Chavez. A method for comparing openflow and p4. pages 1–3, 11 2019.

[32] Docker overview. `https://www.docker.com/resources/what-container`. Accessed: 2021-09-20.

[33] Mininet overview. `http://mininet.org/overview/`. Accessed: 2021-09-20.

[34] Giuseppe Lena, Andrea Tomassilli, Damien Saucez, Frederic Giroire, Thierry Turletti, and Chidung Lac. Mininet on steroids: exploiting the cloud for mininet performance. pages 1–3, 11 2019.

[35] Olivier Flauzac, Erick Robledo, and Florent Nolot. Is mininet the right solution for an sdn testbed? pages 1–6, 12 2019.

[36] Mário Antunes, Diogo Gomes, and Rui Aguiar. Semantic-based publish/subscribe for m2m. In *2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 256–263, 2014.

[37] Ralf Kundel, Christoph Gärtner, Manisha Luthra, Sukanya Bhowmik, and Boris Koldehofe. Flexible content-based publish/subscribe over programmable data planes. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, page 1–5. IEEE Press, 2020.

[38] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. Chapter 5 - source code transformations and optimizations. In João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz, editors, *Embedded Computing for High Performance*, pages 137–183. Morgan Kaufmann, Boston, 2017.

[39] Farzad Khodadadi, A. V. Dastjerdi, and R. Buyya. Internet of things: An overview. *ArXiv*, abs/1703.06409, 2017.

[40] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 15–28, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Shih-Jung Hsu, Chih-Heng Ke, Yeong-Sheng Chen, Cheng-Feng Hung, and Yu-Wen Lo. Design and performance evaluation of a p4 based load balancer. pages 149–152, 10 2019.

[42] Christian Wernecke, Helge Parzyjegla, Gero Mühl, Peter Danielis, and Dirk Timmermann. Realizing content-based publish/subscribe with p4. In *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–7, 2018.

[43] Jian Li, Hao Jiang, Wei Jiang, Jing Wu, and Wen Du. Sdn-based stateful firewall for cloud. In *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 157–161, 2020.

[44] Joaquin Garcia-Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia, Salvador Martinez, and Jordi Cabot. Management of stateful firewall misconfiguration. *Computers Security*, 39:64–85, 2013. 27th IFIP International Information Security Conference.

Appendix A

P4 Firewall ecosystem

A.1 docker-compose.yml

```
version: "3.5"

services:
  mininet:
    image: opennetworking/mn-stratum
    hostname: mininet
    container_name: mininet
    privileged: true
    tty: true
    stdin_open: true
    restart: always
    volumes:
      - ./tmp:/tmp
      - ./mininet:/mininet
    ports:
      - "50001:50001"
      - "50002:50002"
      - "50003:50003"
      - "50004:50004"
    entrypoint: "/mininet/basic.py"

  onos:
    image: onosproject/onos:2.2.7
    hostname: onos
    container_name: onos
    ports:
      - "8181:8181" # HTTP
      - "8101:8101" # SSH (CLI)
    volumes:
      - ./tmp/onos:/root/onos/apache-karaf-4.2.8/data/tmp
```

```
    environment:
      - ONOS_APPS=gui2,drivers.bmv2,hostprovider, lldpprovider, pipelines.fabric, segmentrou
    links:
      - mininet
```

A.2 basic.py

```
#!/usr/bin/python

#  Copyright 2019-present Open Networking Foundation
#
#  Licensed under the Apache License, Version 2.0 (the "License");
#  you may not use this file except in compliance with the License.
#  You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
#  Unless required by applicable law or agreed to in writing, software
#  distributed under the License is distributed on an "AS IS" BASIS,
#  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#  See the License for the specific language governing permissions and
#  limitations under the License.

import argparse


from mininet.log import setLogLevel, info, debug
from mininet.net import Mininet
from mininet.node import Host
from mininet.topo import Topo
from mininet.cli import CLI
from stratum import StratumBmv2Switch
from time import sleep
import json
import os
import subprocess
import sys
from mininet.net import Mininet
from mininet.node import Controller, Host
```

```
CPU_PORT = 255
class IPv4Host(Host):
    """Host that can be configured with an IPv4 gateway (default route).
    """

    def config(self, mac=None, ip=None, defaultRoute=None, lo='up', gw=None,
               **_params):
        super(IPv4Host, self).config(mac, ip, defaultRoute, lo, **_params)
        self.cmd('ip -4 addr flush dev %s' % self.defaultIntf())
        self.cmd('ip -6 addr flush dev %s' % self.defaultIntf())
        self.cmd('ip -4 link set up %s' % self.defaultIntf())
        self.cmd('ip -4 addr add %s dev %s' % (ip, self.defaultIntf()))
        if gw:
            self.cmd('ip -4 route add default via %s' % gw)
        # Disable offload
        for attr in ["rx", "tx", "sg"]:
            cmd = "/sbin/ethtool --offload %s %s off" % (
                self.defaultIntf(), attr)
            self.cmd(cmd)

        def updateIP():
            return ip.split('/')[0]

        self.defaultIntf().updateIP = updateIP

class IPv6Host(Host):
    """Host that can be configured with an IPv6 gateway (default route).
    """

    def config(self, ipv6, ipv6_gw=None, **params):
        super(IPv6Host, self).config(**params)
        self.cmd('ip -4 addr flush dev %s' % self.defaultIntf())
        self.cmd('ip -6 addr flush dev %s' % self.defaultIntf())
        self.cmd('ip -6 addr add %s dev %s' % (ipv6, self.defaultIntf()))
        if ipv6_gw:
            self.cmd('ip -6 route add default via %s' % ipv6_gw)
        # Disable offload
        for attr in ["rx", "tx", "sg"]:
```

```
            cmd = "/sbin/ethtool --offload %s %s off" % (self.defaultIntf(), attr)
            self.cmd(cmd)

        def updateIP():
            return ipv6.split('/')[0]

        self.defaultIntf().updateIP = updateIP

    def terminate(self):
        super(IPv6Host, self).terminate()

class TaggedIPv4Host(Host):
    """VLAN-tagged host that can be configured with an IPv4 gateway
    (default route).
    """
    vlanIntf = None

    def config(self, mac=None, ip=None, defaultRoute=None, lo='up', gw=None,
               vlan=None, **_params):
        super(TaggedIPv4Host, self).config(mac, ip, defaultRoute, lo, **_params)
        self.vlanIntf = "%s.%s" % (self.defaultIntf(), vlan)
        # Replace default interface with a tagged one
        self.cmd('ip -4 addr flush dev %s' % self.defaultIntf())
        self.cmd('ip -6 addr flush dev %s' % self.defaultIntf())
        self.cmd('ip -4 link add link %s name %s type vlan id %s' % (
            self.defaultIntf(), self.vlanIntf, vlan))
        self.cmd('ip -4 link set up %s' % self.vlanIntf)
        self.cmd('ip -4 addr add %s dev %s' % (ip, self.vlanIntf))
        if gw:
            self.cmd('ip -4 route add default via %s' % gw)

        self.defaultIntf().name = self.vlanIntf
        self.nameToIntf[self.vlanIntf] = self.defaultIntf()

        # Disable offload
        for attr in ["rx", "tx", "sg"]:
            cmd = "/sbin/ethtool --offload %s %s off" % (
                self.defaultIntf(), attr)
            self.cmd(cmd)
```

```
        def updateIP():
            return ip.split('/')[0]

        self.defaultIntf().updateIP = updateIP

    def terminate(self):
        self.cmd('ip -4 link remove link %s' % self.vlanIntf)
        super(TaggedIPv4Host, self).terminate()


class DemoTopo(Topo):
    """2x2 fabric topology with IPv6 hosts"""

    def __init__(self, *args, **kwargs):
        Topo.__init__(self, *args, **kwargs)

        loglevel = "debug"

        topo_file = json.load(open(os.path.join(sys.path[0], 'topology/topologyLinear1.js

        equipments = {}
        for sw_name in topo_file['switches']:
            sw_info = topo_file['switches'][sw_name]
            sw = self.addSwitch(sw_name, cls=StratumBmv2Switch, cpuport=CPU_PORT, logleve
            equipments[sw_name] = sw

        for host_name in topo_file['hosts']:
            host_info = topo_file['hosts'][host_name]
            host = self.addHost(host_name, cls=IPv4Host, mac=host_info['mac'], ip=host_in
            equipments[host_name] = host

        for link in topo_file['links']:
            node1, node2 = link.split('-')
            link_info = topo_file['links'][link]
            self.addLink(equipments[node1], equipments[node2], port1 = int(link_info['por
def main():
    net = Mininet(topo=DemoTopo(), controller=None)
    net.start()
    CLI(net)
    net.stop()
```

```
    print ('#' * 80)
    print ('ATTENTION: Mininet was stopped! Perhaps accidentally?')
    print ('No worries, it will restart automatically in a few seconds...')
    print ('To access again the Mininet CLI, use `make mn-cli`')
    print ('To detach from the CLI (without stopping), press Ctrl-D')
    print ('To permanently quit Mininet, use `make stop`')
    print ('#' * 80)


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description='Mininet topology script for 2x2 fabric with stratum_bmv2 and IPv6 hosts

    args = parser.parse_args()

    setLogLevel('info')

    main()
```

A.3 topology.json

```
{
    "hosts": {
        "h1": {"ip": "10.0.1.10/24", "mac": "00:00:00:00:00:01", "gateway": "10.0.1.100" },
        "h2": {"ip": "10.0.2.10/24", "mac": "00:00:00:00:00:02", "gateway": "10.0.2.100" },
        "h3": {"ip": "10.0.3.10/24", "mac": "00:00:00:00:00:03", "gateway": "10.0.3.100" }
    },
    "switches": {
        "sw1": { "grpc" : "50001" },
        "sw2": { "grpc" : "50002" },
        "sw3": { "grpc" : "50003" },
        "sw4": { "grpc" : "50004" }
    },
    "links": {
        "sw1-sw3" : { "port1" : "1" , "port2" : "1"},
        "sw1-sw4" : { "port1" : "2" , "port2" : "1"},
        "sw2-sw3" : { "port1" : "1" , "port2" : "2"},
        "sw2-sw4" : { "port1" : "2" , "port2" : "2"},
```

```
        "h1-sw1" : { "port1" : "0" , "port2" : "3"},
        "h3-sw1" : { "port1" : "0" , "port2" : "4"},
        "h2-sw2" : { "port1" : "0" , "port2" : "3"}
    }
}
```

A.4 netcfg_firewall.json

```
{
  "devices": {
    "device:sw3": {
        "basic": {
            "managementAddress": "grpc://mininet:50003?device_id=1",
            "driver": "stratum-bmv2",
            "pipeconf": "org.onosproject.pipelines.fabric",
            "locType": "grid",
            "gridX": 500,
            "gridY": 600
        },
        "segmentrouting": {
            "name": "sw3",
            "ipv4NodeSid": 301,
            "ipv4Loopback": "192.168.2.1",
            "routerMac": "00:BB:00:00:00:03",
            "isEdgeRouter": false,
            "adjacencySids": []
        }
    },
    "device:sw4": {
        "basic": {
            "managementAddress": "grpc://mininet:50004?device_id=1",
            "driver": "stratum-bmv2",
            "pipeconf": "org.onosproject.pipelines.fabric",
            "locType": "grid",
            "gridX": 600,
            "gridY": 600
        },
        "segmentrouting": {
            "name": "sw4",
```

```
        "ipv4NodeSid": 302,
        "ipv4Loopback": "192.168.2.3",
        "routerMac": "00:BB:00:00:00:04",
        "isEdgeRouter": false,
        "adjacencySids": []
    }
},
"device:sw1": {
    "basic": {
        "managementAddress": "grpc://mininet:50001?device_id=1",
        "driver": "stratum-bmv2",
        "pipeconf": "org.onosproject.new-pipelines.fabric",
        "locType": "grid",
        "gridX": 800,
        "gridY": 800
    },
    "segmentrouting": {
        "name": "sw1",
        "ipv4NodeSid": 101,
        "ipv4Loopback": "192.168.1.1",
        "routerMac": "00:aa:00:00:00:01",
        "isEdgeRouter": true,
        "adjacencySids": []
    }
},
"device:sw2": {
    "basic": {
        "managementAddress": "grpc://mininet:50002?device_id=1",
        "driver": "stratum-bmv2",
        "pipeconf": "org.onosproject.pipelines.fabric",
        "locType": "grid",
        "gridX": 200,
        "gridY": 800
    },
    "segmentrouting": {
        "name": "sw2",
        "ipv4NodeSid": 102,
        "ipv4Loopback": "192.168.1.2",
        "routerMac": "00:aa:00:00:00:02",
        "isEdgeRouter": true,
```

```
          "adjacencySids": []
      }
  }
},
"ports": {
  "device:sw1/3": {
    "interfaces": [
      {
        "name": "sw1-eth3",
        "ips": [
          "10.0.1.100/24"
        ],
        "vlan-untagged" : 10
      }
    ]
  },
  "device:sw1/4": {
    "interfaces": [
      {
        "name": "sw1-eth4",
        "ips": [
          "10.0.3.100/24"
        ],
        "vlan-untagged" : 30
      }
    ]
  },
  "device:sw2/3": {
      "interfaces": [
        {
          "name": "sw2-eth2",
          "ips": [
            "10.0.2.100/24"
          ],
          "vlan-untagged" : 20

        }
      ]
    }
```

```
  }
}
```

A.5 .gitlab-ci.yml

```
stages:
  - compile
  - test
  - build

p4Build:
  image: docker:latest
  artifacts:
    paths:
      - p4/
    expire_in: 10 minutes
  services:
    - docker:dind
  stage: compile
  services:
    - docker:dind
  script:
    - echo 'Starting p4 compilation...'
    - 'ls -ls'
    - 'cd p4 && /bin/sh bmv2-compile.sh "fabric" "-DWITH_SIMPLE_NEXT"'
  only:
    - development

p4Test:
  image: docker:latest
  artifacts:
    paths:
      - ptf/
    expire_in: 1 day
  services:
    - docker:dind
  stage: test
  script:
    - echo 'Starting test phase...'
```

```
    - 'cd ptf && /bin/sh run_tests'
  only:
    - development
  dependencies:
    - p4Build


firewallAppBuild:
  image: maven:3.8.1-openjdk-11-slim
  stage: build
  script:
    - 'cp -r p4/p4c-out fabric_ext/src/main/resources'
    - 'cd fabric_ext && mvn clean install'
    - 'mvn org.onosproject:onos-maven-plugin:2.2:app'
  only:
    - development

  dependencies:
    - p4Test
```

A.6 Makefile

```
mkfile_path := $(abspath $(lastword $(MAKEFILE_LIST)))
curr_dir := $(patsubst %/,%,$(dir $(mkfile_path)))


onos_url := http://localhost:8181/onos
onos_curl := curl --fail -sSL --user onos:rocks --noproxy localhost


p4-test:
        @cd ptf && ./run_tests



mvn_ci:
        cp -r p4/p4c-out fabric_ext/src/main/resources
        cd fabric_ext && mvn clean install
```

```
mvn_pack:
	cd fabric_ext && mvn org.onosproject:onos-maven-plugin:2.2:app




_start:
	$(info *** Starting BMv2, ONOS and Hosts)
	docker-compose up -d



_stop:
	docker-compose down
	@echo "Containers are down"


p4_compile:
	cd p4 && make fabric
	@echo


app-install:
	$(info *** Installing and activating app in ONOS...)
	${onos_curl} -X POST -HContent-Type:application/octet-stream \
		'${onos_url}/v1/applications?activate=true' \
		--data-binary @fabric_ext/target/fabric_firewall_extension-1.0.0-SNAPSHOT.oa
	@echo




app-uninstall:
	$(info *** Uninstalling app from ONOS (if present)...)
	-${onos_curl} -X DELETE ${onos_url}/v1/applications/org.onosproject.fabric_firewall_
	@echo

app-reload: app-uninstall app-install
```

```
push-netconf-firewall: CONF := netcfg_firewall.json
push-netconf-firewall: push-netconf


push-netconf:
        $(info *** Pushing ${CONF} to ONOS...)
        ${onos_curl} -X POST -H 'Content-Type:application/json' \
                ${onos_url}/v1/network/configuration -d@./config/${CONF}
        @echo


onos-cli:
        $(info *** Connecting to the ONOS CLI...)
        $(info *** Top exit press Ctrl-D)
        @ssh -o "UserKnownHostsFile=/dev/null" -o "StrictHostKeyChecking=no" -o LogLevel=

onos-log:
        docker logs onos

mn-cli:
        $(info *** Attaching to Mininet CLI...)
        $(info *** To detach press Ctrl-D (Mininet will keep running))
        -@docker attach --detach-keys "ctrl-d" $(shell docker-compose ps -q mininet) || e

pcap:
        docker exec -it mininet /mininet/host-cmd \
        $(host) tcpdump -i $(host)-eth$(iface) \
        -U -w /tmp/$(host)-eth$(iface).pcap
```

A.7 PtfAddressTest.py

```
@group("address")
class SrcAddressExactPermit(P4RuntimeTest):
    """Tests
    """

    def runTest(self):
```

```
    print_inline("ICMP Packet ... ")

    configure = ConfigureFabric()
    configure.configFabricPipe(self)


    icmp_pkt = create_icmp_packet()

    self.testPermitPacketSrcExact(icmp_pkt, configure.inPort, configure.outPort)



@autocleanup
def testPermitPacketSrcExact(self, pkt, pIn, pOut):

    print('\n\nAllow Packet Test')
    inPort  = pIn
    outPort = pOut

    # ---- START SOLUTION ----
    self.insert(self.helper.build_table_entry(
        table_name="FabricIngress.fwll_filtering.firewall_filtering_table",
        match_fields={
            # Ternary match.
            "src": ( HOST2_IP, 0xffffffff),
        },
        action_name="FabricIngress.fwll_filtering.permit",
        priority=DEFAULT_PRIORITY
    ))
    # ---- END SOLUTION ----

    exp_pkt = pkt.copy()
    next_hop_mac = HOST1_MAC
    new_packet = pkt_route(exp_pkt, next_hop_mac)
    new_packet = pkt_decrement_ttl(new_packet)
    # maskedP = mask.Mask(new_packet)


    print_inline("ICMP Packet Sent... ")
```

```
        # Send packet...
        testutils.send_packet(self, inPort, str(pkt))

        # verify on outport port.
        testutils.verify_packet(self, new_packet, outPort)
        print_inline("ICMP Packet Received on Output Port... ")


        testutils.verify_no_other_packets(self)


@group("address")
class SrcAddressExactDrop(P4RuntimeTest):
    """Tests
    """

    def runTest(self):
        print_inline("ICMP Packet ... ")

        configure = ConfigureFabric()
        configure.configFabricPipe(self)


        icmp_pkt = create_icmp_packet()
        self.testDropPacketSrcExact(icmp_pkt, configure.inPort, configure.outPort)



    @autocleanup
    def testDropPacketSrcExact(self, pkt, pIn, pOut):

        print('\n\nDrop Packet Test')
        inPort  = pIn
        outPort = pOut

        # ---- START SOLUTION ----
        self.insert(self.helper.build_table_entry(
            table_name="FabricIngress.fwll_filtering.firewall_filtering_table",
            match_fields={
                # Ternary match.
```

```
                "src": ( HOST2_IP, 0xffffffff)
            },
            action_name="FabricIngress.fwll_filtering.drop",
            priority=DEFAULT_PRIORITY
        ))
        # ---- END SOLUTION ----

        # Send packet...
        testutils.send_packet(self, inPort, str(pkt))
        print_inline("ICMP Packet Sent... ")



        testutils.verify_no_other_packets(self)

        print_inline("Packet not received as expected...")
```

A.8 firewall_filtering.p4

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

#include "../header.p4"

control FirewallFiltering (inout parsed_headers_t hdr,
                   inout fabric_metadata_t fabric_metadata,
                   inout standard_metadata_t standard_metadata) {

    direct_counter(CounterType.packets_and_bytes) firewall_counter;


    action drop() {
        mark_to_drop(standard_metadata);
        fabric_metadata.skip_next = _TRUE;
        firewall_counter.count();

    }

    action permit() {
```

```
        // Allow packet as is.
        firewall_counter.count();
    }


    table firewall_filtering_table{
        key = {
            // optional with lpm, ternary does the job
            hdr.ipv4.src_addr               : ternary @name("src");

            // optional with lpm, ternary does the job
            hdr.ipv4.dst_addr               : ternary @name("dst");

            // required but can match all, ternary does the job
            fabric_metadata.ip_proto        : ternary @name("protocol");

            // optional with exact value or range
            fabric_metadata.l4_dport        : range @name("dport");

            // optional with exact value or range
            fabric_metadata.l4_sport        : range @name("sport");

        }
        actions = {
            drop();
            permit();

        }
        counters = firewall_counter;
        const default_action = drop();

    }

    apply {

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

Appendix B

Zenoh P4 ecosystem

B.1 Dockerfile

```
FROM opennetworking/mn-stratum:20.12 as runtime

RUN install_packages curl ca-certificates \
python3-pip python3-setuptools

RUN apt update -y
RUN apt upgrade -y

RUN apt install wget build-essential libreadline-gplv2-dev \
libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev \
libc6-dev libbz2-dev libffi-dev zlib1g-dev -y

RUN wget https://www.python.org/ftp/python/3.9.4/Python-3.9.4.tgz
RUN tar xzf Python-3.9.4.tgz

WORKDIR Python-3.9.4

RUN ./configure --enable-optimizations
RUN make altinstall


WORKDIR /
RUN python3 --version
RUN python3.9 --version


RUN python3.9 -m pip install --upgrade pip
RUN python3.9 -m pip install --upgrade setuptools
RUN curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs -o \
rustup.sh && sh rustup.sh -y
RUN pip3.9 install --no-cache-dir eclipse-zenoh==0.5.0-b8
```

```
WORKDIR /root
COPY ./stratum.py ./stratum.py
```

B.2 stratum.py

```
# coding=utf-8
# Copyright 2018-present Open Networking Foundation
# SPDX-License-Identifier: Apache-2.0

'''
This module contains a switch class for Mininet: StratumBmv2Switch

Prerequisites
-------------
1. Docker- mininet+stratum_bmv2 image:
$ cd stratum
$ docker build -t <some tag> -f tools/mininet/Dockerfile .

Usage
-----
From withing the Docker container, you can run Mininet using the following:
$ mn --custom /root/stratum.py --switch stratum-bmv2 --controller none

Advanced Usage
--------------
You can use this class in a Mininet topology script by including:

from stratum import ONOSStratumBmv2Switch

You will probably need to update your Python path. From within the Docker image:

PYTHONPATH=$PYTHONPATH:/root ./<your script>.py

Notes
-----
This code has been adapted from the ONOSBmv2Switch class defined in the ONOS project
(tools/dev/mininet/bmv2.py).
```

```
'''

import json
import multiprocessing
import os
import socket
import threading
import time

from mininet.log import warn
from mininet.node import Switch, Host

DEFAULT_NODE_ID = 1
DEFAULT_CPU_PORT = 255
DEFAULT_PIPECONF = "org.onosproject.pipelines.basic"
STRATUM_BMV2 = 'stratum_bmv2'
STRATUM_INIT_PIPELINE = '/root/dummy.json'
MAX_CONTROLLERS_PER_NODE = 10
BMV2_LOG_LINES = 5


def writeToFile(path, value):
    with open(path, "w") as f:
        f.write(str(value))


def pickUnusedPort():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('localhost', 0))
    addr, port = s.getsockname()
    s.close()
    return port


def watchdog(sw):
    try:
        writeToFile(sw.keepaliveFile,
                    "Remove this file to terminate %s" % sw.name)
        while True:
            if StratumBmv2Switch.mininet_exception == 1 \
```

```
                    or not os.path.isfile(sw.keepaliveFile):
                sw.stop()
                return
            if sw.stopped:
                return
            if sw.bmv2popen.poll() is None:
                # All good, no return code, still running.
                time.sleep(1)
            else:
                warn("\n*** WARN: switch %s died   \n" % sw.name)
                sw.printLog()
                print("-" * 80) + "\n"
                # Close log file, set as stopped etc.
                sw.stop()
                return
    except Exception as e:
        warn("*** ERROR: " + e.message)
        sw.stop()


class StratumBmv2Switch(Switch):
    # Shared value used to notify to all instances of this class that a Mininet
    # exception occurred. Mininet exception handling doesn't call the stop()
    # method, so the mn process would hang after clean-up since Bmv2 would still
    # be running.
    mininet_exception = multiprocessing.Value('i', 0)

    nextGrpcPort = 50001

    def __init__(self, name, json=STRATUM_INIT_PIPELINE, loglevel="debug",
                 cpuport=DEFAULT_CPU_PORT, pipeconf=DEFAULT_PIPECONF,
                 onosdevid=None,
                 **kwargs):
        Switch.__init__(self, name, **kwargs)
        self.grpcPort = StratumBmv2Switch.nextGrpcPort
        StratumBmv2Switch.nextGrpcPort += 1
        self.cpuPort = cpuport
        self.json = json
        self.loglevel = loglevel
        self.tmpDir = '/tmp/%s' % self.name
```

```
        self.logfile = '%s/stratum_bmv2.log' % self.tmpDir
        self.netcfgFile = '%s/onos-netcfg.json' % self.tmpDir
        self.chassisConfigFile = '%s/chassis-config.txt' % self.tmpDir
        self.pipeconfId = pipeconf
        self.longitude = kwargs['longitude'] if 'longitude' in kwargs else None
        self.latitude = kwargs['latitude'] if 'latitude' in kwargs else None
        if onosdevid is not None and len(onosdevid) > 0:
            self.onosDeviceId = onosdevid
        else:
            # The "device:" prefix is required by ONOS.
            self.onosDeviceId = "device:%s" % self.name
        self.nodeId = DEFAULT_NODE_ID
        self.logfd = None
        self.bmv2popen = None
        self.stopped = True
        # In case of exceptions, mininet removes *.out files from /tmp. We use
        # this as a signal to terminate the switch instance (if active).
        self.keepaliveFile = '/tmp/%s-watchdog.out' % self.name

        # Remove files from previous executions
        self.cleanupTmpFiles()
        os.mkdir(self.tmpDir)

    def getOnosNetcfg(self):
        basicCfg = {
            "managementAddress": "grpc://localhost:%d?device_id=%d" % (
                self.grpcPort, self.nodeId),
            "driver": "stratum-bmv2",
            "pipeconf": self.pipeconfId
        }

        if self.longitude and self.latitude:
            basicCfg["longitude"] = self.longitude
            basicCfg["latitude"] = self.latitude

        netcfg = {
            "devices": {
                self.onosDeviceId: {
                    "basic": basicCfg
                }
```

```
            }
        }

        return netcfg

    def getChassisConfig(self):
        config = """description: "stratum_bmv2 {name}"
chassis {{
  platform: PLT_P4_SOFT_SWITCH
  name: "{name}"
}}
nodes {{
  id: {nodeId}
  name: "{name} node {nodeId}"
  slot: 1
  index: 1
}}\n""".format(name=self.name, nodeId=self.nodeId)

        intf_number = 1
        for intf_name in self.intfNames():
            if intf_name == 'lo':
                continue
            config = config + """singleton_ports {{
  id: {intfNumber}
  name: "{intfName}"
  slot: 1
  port: {intfNumber}
  channel: 1
  speed_bps: 10000000000
  config_params {{
    admin_state: ADMIN_STATE_ENABLED
  }}
  node: {nodeId}
}}\n""".format(intfName=intf_name, intfNumber=intf_number, nodeId=self.nodeId)
            intf_number += 1

        return config

    def start(self, controllers):
```

```
if not self.stopped:
    warn("*** %s is already running!\n" % self.name)
    return

writeToFile("%s/grpc-port.txt" % self.tmpDir, self.grpcPort)
with open(self.chassisConfigFile, 'w') as fp:
    fp.write(self.getChassisConfig())
with open(self.netcfgFile, 'w') as fp:
    json.dump(self.getOnosNetcfg(), fp, indent=2)

args = [
    STRATUM_BMV2,
    '-device_id=%d' % self.nodeId,
    '-chassis_config_file=%s' % self.chassisConfigFile,
    '-forwarding_pipeline_configs_file=%s/pipe.txt' % self.tmpDir,
    '-persistent_config_dir=%s' % self.tmpDir,
    '-initial_pipeline=%s' % self.json,
    '-cpu_port=%s' % self.cpuPort,
    '-external_stratum_urls=0.0.0.0:%d' % self.grpcPort,
    '-local_stratum_url=localhost:%d' % pickUnusedPort(),
    '-max_num_controllers_per_node=%d' % MAX_CONTROLLERS_PER_NODE,
    '-write_req_log_file=%s/write-reqs.txt' % self.tmpDir,
    '-bmv2_log_level=trace',
]

cmd_string = " ".join(args)

try:
    # Write cmd_string to log for debugging.
    self.logfd = open(self.logfile, "w")
    self.logfd.write(cmd_string + "\n\n" + "-" * 80 + "\n\n")
    self.logfd.flush()

    self.bmv2popen = self.popen(cmd_string, stdout=self.logfd, stderr=self.logfd)
    print "     %s @ %d" % (STRATUM_BMV2, self.grpcPort)

    # We want to be notified if stratum_bmv2 quits prematurely...
    self.stopped = False
    threading.Thread(target=watchdog, args=[self]).start()
```

```
            except Exception:
                StratumBmv2Switch.mininet_exception = 1
                self.stop()
                self.printLog()
                raise

        def printLog(self):
            if os.path.isfile(self.logfile):
                print "-" * 80
                print "%s log (from %s):" % (self.name, self.logfile)
                with open(self.logfile, 'r') as f:
                    lines = f.readlines()
                    if len(lines) > BMV2_LOG_LINES:
                        print "..."
                    for line in lines[-BMV2_LOG_LINES:]:
                        print line.rstrip()

        def cleanupTmpFiles(self):
            self.cmd("rm -rf %s" % self.tmpDir)

        def stop(self, deleteIntfs=True):
            """Terminate switch."""
            self.stopped = True
            if self.bmv2popen is not None:
                if self.bmv2popen.poll() is None:
                    self.bmv2popen.terminate()
                    self.bmv2popen.wait()
                self.bmv2popen = None
            if self.logfd is not None:
                self.logfd.close()
                self.logfd = None
            Switch.stop(self, deleteIntfs)


    class NoOffloadHost(Host):
        def __init__(self, name, inNamespace=True, **params):
            Host.__init__(self, name, inNamespace=inNamespace, **params)

        def config(self, **params):
            r = super(Host, self).config(**params)
```

```
        for off in ["rx", "tx", "sg"]:
            cmd = "/sbin/ethtool --offload %s %s off" \
                  % (self.defaultIntf(), off)
            self.cmd(cmd)
        return r

class NoIpv6OffloadHost(NoOffloadHost):
    def __init__(self, name, inNamespace=True, **params):
        NoOffloadHost.__init__(self, name, inNamespace=inNamespace, **params)

    def config(self, **params):
        r = super(NoOffloadHost, self).config(**params)
        self.cmd("sysctl net.ipv6.conf.%s.disable_ipv6=1" % (self.defaultIntf()))
        return r


# Exports for bin/mn
switches = {'stratum-bmv2': StratumBmv2Switch}

hosts = {
    'no-offload-host': NoOffloadHost,
    'no-ipv6-host': NoIpv6OffloadHost
}
```