Advanced SDN Applications using the P4 language

Bernardo Folques de Valente

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisor: Prof. Fernando Henrique Côrte-Real Mira da Silva

Examination Committee

October 2021

Acknowledgments

Firstly, I would like to thank my family for their support and by providing me the means to achieve higher education. Specially to my father Mário, for being a role model throughout the years and asking hard questions that got me thinking in the right direction.

Next, I would like to give a big thanks to my girlfriend, Sofia Bastos, which was always present, supportive, and understanding in this journey of my life.

I would also like to thank my university colleagues and friends, they were always there for study sessions, as well as adventures outside of university.

Last, but not least, I would like to thank my supervisor, Prof. Fernando Mira da Silva, for its outstanding supervision and constantly motivating me to do my best work.

Abstract

Software Defined Networking (SDN) aims to provide dynamic and programmable solutions for network management. In its original form, the SDN paradigm decouples the data and control plane of conventional network devices, centralizing all control functions of the SDN domain on a single server with a unified view of the network. Network devices were therefore simplified to a reduced set of data plane operations, programmed using a standard protocol. OpenFlow was possibly the most successful SDN protocol.

However, it soon became clear that most of the flexibility offered by the SDN model was limited by the capabilities and fields defined in the OpenFlow protocol itself. As a way of overcoming this limitation, an alternative model was proposed based on the concept of programmable data plane. P4 (Programming Protocol-independent Packet Processors) is a language developed for specification of how data packets should be processed and forwarded on compatible network devices. Since rules and packet actions are defined at the bit and byte level, P4 offers a more detailed control over network traffic than the one possible by OpenFlow solutions.

This work presents a novel load balancing solution using the P4 language which includes a production-grade SDN controller, a programmable data plane compatible with any number of connected servers, and a stateful load balancing algorithm with fault tolerance capabilities.

Keywords

Software Defined Networking; OpenFlow; Data Plane Programmability; P4; Load Balancing

Resumo

Redes Definidas por Software (RDS) tem como objetivo promover soluções dinâmicas e programáveis para gestão de redes. O paradigma original de RDS tem como objetivo separar o plano de dados e plano de controlo de dispositivos convencionais de rede, centralizando o plano de controlo num servidor com uma visão unificada da rede. Os dispositivos de rede foram transformados em dispositivos mais simples com apenas operações de plano de dados, programáveis com um protocolo standard. OpenFlow foi possivelmente o protocolo SDN mais bem-sucedido.

No entanto, a flexibilidade oferecida pelo modelo RDS foi limitada pelas capacidades do protocolo OpenFlow. Para superar estas limitações, um novo modelo foi proposto baseado no conceito de um plano de dados programável. P4 (Programming Protocol-Independent Packet Processor) é uma linguagem desenvolvida para especificar como pacotes de dados devem ser processados e encaminhados em dispositivos de rede compatíveis. Como regras e acções de pacotes são definidos aos níveis de bit e byte, P4 oferece um controlo mais detalhado sobre tráfico de rede comparando com o que é possível com soluções OpenFlow.

Este trabalho apresenta uma solução de balanceamento de carga original com a lingaugem P4 que inclui um controlador SDN a nível de produção, um plano de dados programável compatível com qualquer número de servidores conectados, e um algoritmo de balanceamento de carga *stateful* com capacidades de tolerância a faltas.

Palavras Chave

Redes Definidas por Software; OpenFlow; Plano de Dados Programável; P4; Balanceamento de Carga

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

API	Application Programming Interface
ASIC	Application-specific Integrated Circuit
BMV2	Behavioral model version 2
COTS	Commercial Off the Shelf
CSV	Comma-Separated Values
DPP	Data Plane Programmability
ECMP	Equal-cost Multi-path
gRPC	gRPC Remote Procedure Calls
HTTP	Hypertext Transfer Protocol
IPv4	Internet Protocol version 4
JSON	JavaScript Object Notation
MAC	Media Access Address
NBI	Northbound Interface
NFV	Network Function Virtualization
NV	Network Virtualization
OF	OpenFlow
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OS	Operating System

OVS	Open vSwitch
P4	Programming Protocol-Independent Packet Processors
PISA	Protocol Independent Switch Architecture
PSA	Portable Switch Architecture
SBI	Southbound Interface
SDN	Software-Defined Networking
STP	Spanning Tree Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time to Live
VNF	Virtual Network Function
VM	Virtual Machine

1 Introduction

Contents

Computer networks have evolved drastically since they were first born. For many years it was common to buy new hardware every time a new function was to be included in a network. This was the only option since dedicated hardware was much faster than using programmable chips. Luckily, due to Moore's Law, computer chips are now cheaper and faster than ever. General-purpose computer chips from nowadays can perform features almost as fast as dedicated chips. The versatility from these chips opens doors to a new technological revolution, the Virtualization era [1].

Virtualization is a technique to manage hardware used by multiple software systems. In networking it is used to run functions originally designed to run in dedicated hardware, in programmable chips. Its concept dates from the 1960's but has earned a lot of traction recently with the evolution of computer chips.

Nowadays, it is possible to virtualize nearly every network function originally designed to be performed in custom physical hardware. A regular network administrator with the help of network virtualization can add a new function to his network or reconfigure operational resources without the need to rewire the physical network or add new network devices, provided that the underlying physical computer and network hardware have the required capacity. It also brought great advantages for network development, allowing developers to design and test new network node architectures virtually, without the need to first create dedicated hardware [2].

Software Defined Networking (SDN) [3] is a new network paradigm that separates the data and control planes from the same network node. This allows for a centralized control plane application that can effortlessly manage and configure every node in a network. There was a great evolution in the control plane with the help of SDN. However, the original SDN model assumed rigid network devices, which often limited the capabilities of the data plane. This limitation led to the introduction of the concept of Data Plane Programmability [4].

1.1 Motivation

The conventional implementation of SDN networks assume non-programmable switches configured with a remote controller through a well-defined protocol. This is overall a good solution for some occasions; however, it has some known shortcomings that will be discussed in section 2.3.1. P4 was proposed [4], in the scope of data plane programmability, as a domain-specific language, designed to program network nodes, and overcome the limitations imposed by the traditional static SDN data plane architecture.

The P4 programming language offers many benefits, including high customizability of the data plane and control over the code running in the network nodes. One of the key aspects of P4 is to be protocol-independent. Switches can be loaded with large range of programs and each might have a different implementation of multiple protocols. Having one of these nodes in a network makes it very agile for future implementation of new protocols or network functions. The network administrator just needs to extend the P4 code in the already available and ready programmable-switches. Changes can be simple like changing how a protocol analyses a packet's header information, adding keys to be matched in a flow-table or adding new counters and sending the information to the network controller for statistics purpose either to have a better understanding of the network traffic, or to populate big data for network analysis and machine learning. Switches are also available to more complex network functions, like adding a firewall function to filter unwanted traffic or installing a load balancer in the network for better control of flows.

Shifting a network from fixed-function to programmable-switches is an investment for future network implementations, the network becomes agile and available to change how the data plane works. However, P4 still presents considerable challenges before it can be widely deployed. The motivation for this work was the need to explore and contribute to new P4 applications.

1.2 Objectives

The objective of this thesis is to bring Data Plane Programmability in the form of the P4 language to Software Defined Networks, in order to have a better understanding of the capabilities that P4 can introduce in a Software Defined Network.

The implementation should provide the network with benefits from Data Plane Programmability such as high flexibility to include new protocols, as well as provide a base platform to build and install new functions, making a transition to a future-proof, agile network.

To accomplish the work proposed, we have decided to develop a load balancing system. These are relatively easy to develop, while at the same time giving some room for improvements and creativity. Also, Load Balancers are crucial components of data centers, being therefore ubiquitous and fundamental to the day to day life of a modern technological society.

The goal of this work is to contribute to improve P4 solutions for LB applications. To accomplish that we propose to consult articles mentioning Load Balancing projects and develop an application based on these. The objective is to improve the solutions found by introducing new features and overcoming some of their obstacles an limitations. Later in this thesis we will make a further comparison between our work and the related work.

1.3 Contributions

In this section we describe the contributions achieved with this paper.

Our main contribution is to extend the work done in the paper by Chih-Heng Ke et al. [5]. Namely, we introduce a production-grade SDN controller with features like code modularity and configurability, improve the P4 code to make it compatible with any number of servers, and transition from a stateless load balancer to a stateful one.

Additionally, we presented a paper in the symposium INFORUM 2021 about the work described in this thesis.

1.4 Document Structure

The document is structured as follows: Chapter 2 presents the Background around Data Plane Programmability and the technologies that lead to this innovation. Chapter 3 describes the Proposed Solution of the application presented in the dissertation. Chapter 4 presents a detailed implementation of the application developed. In Chapter 5 we evaluate the developed application. Finally, Chapter 5 presents conclusions and discusses the main outcomes of this paper.

2 Background

Contents

With the fast evolution of technology in the 21st century, the telecommunication industry had to adapt to bring internet connectivity to every new device manufactured. In the beginning, whenever a network change happened, new physical proprietary hardware had to be deployed for the network to keep its quality and availability. However, hardware is immutable, and it reaches the end of life rather quickly, which forces network administrators to systematically buy new hardware. This is not profitable for owners of large data centers and is not a viable solution for the ever-expanding internet.

In this chapter, we present an introduction to Network Virtualization (NV), a technology that changes how today's networks are operated, as well as an overview of the technologies and paradigms that followed, including Network Function Virtualization (NFV), Software Defined Networking (SDN) and Data Plane Programmability (DPP).

2.1 Virtualization

Virtualization is the process of managing a system's hardware components to be shared by multiple software systems. Most resources can be virtualized and shared such as storage, memory, and processing power. Computer virtualization comes in many forms, from virtualizing whole Operating Systems (OS) to single applications running in an isolated environment.

It was first introduced by IBM in the 1960's to divide mainframe computers' resources to allow for multiple applications to run simultaneously. Since then, Virtualization has greatly evolved but the concept remains the same [2].

A Virtual Machine (VM) is an emulation of a computer system used to run an Operating System. The physical hardware is referred to as the host, while a VM is referred to as the guest. One host can emulate multiple guests. The host shares his resources with the VM, these are isolated from the main host's resources and other VMs. This is useful when a specific work environment is needed to develop and run an application. Although virtual machines are helpful and have a lot of advantages, users should be careful. Running multiple virtual machines in one physical machine can result in unstable performance if the host's computational or memory capacity is not enough for the number of running guests.

The lowest level of virtualization comes in the form of Hypervisors [6]. A Hypervisor is a software that manages other virtual machines. Hypervisors can come in two types. Type 1 hypervisors, also called bare metal hypervisors, run directly in the host's hardware. Big enterprises with large data centers prefer to use this type of hypervisors. They are considered Operating Systems themselves, even though its only purpose is virtualization. The Type 2 hypervisors run on top of another Operating System, as for example QEMU and VirtualBox. The second type provides a platform much easier to manage VMs than the first type. This is commonly used when a user occasionally needs to quickly run a Virtual Machine

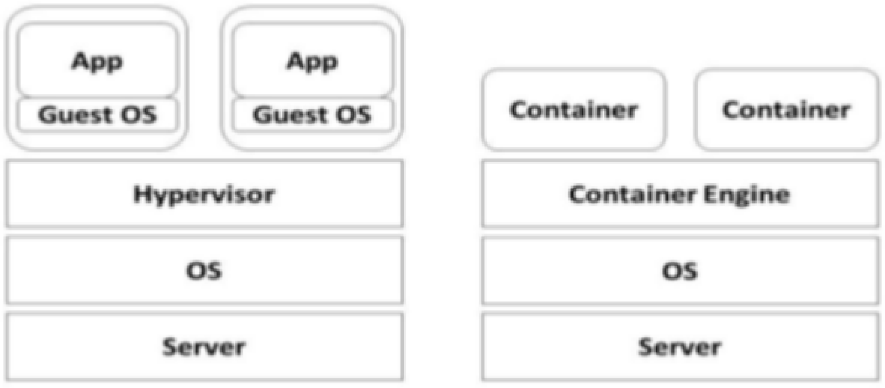


Figure 2.1: Virtual Machines vs Containers.
Reproduced from [6].

for a specific purpose. Type 2's disadvantage is that the performance is inferior since it has to run on top of a full working OS.

Containerization is another form of virtualization that creates isolated user spaces called Containers [6]. Unlike Virtual Machines, that virtualize a complete computer system, containers just virtualize the OS, these are used for running a specific program in an isolated environment. These have a much smaller footprint, and their setup time is faster. They share the same Linux kernel as the host and therefore are faster to deploy since the host does not have to run a new virtualized kernel on top of host operating system. Each container requires an image to run. An image contains all necessary source code and libraries needed for an application to run. Figure 2.1 illustrates the difference between running Virtual Machines and Containers. Although containerization is a good solution for running most applications, some require tools only found on VMs. Additionally, it has a weaker level of isolation than VMs, leading to security concerns in critical applications.

Docker is the most widely adopted container manager. Docker provides the world's largest repository of container images called Docker Hub [7]. Docker will be described in more detail in section 4.1.1.

2.2 Network Function Virtualization

A traditional network infrastructure is composed by conventional networking hardware and physical interfaces that connect servers between themselves and to the internet. This has been the standard since the internet was born and it is still the most used model in many networks.. However, the current size of today's internet is much larger than when it was first designed, and it is in constant change due to its never stopping expansion. Implementing physical network nodes for every change in a network is hard, expensive and is not future proof.

Hence, Network Virtualization (NV) was developed as a method for abstracting network resources. NV can either segment a physical network into multiple virtual networks or combine multiple physical networks into one virtual network.

Network Function Virtualization (NFV) [8] is a concept that uses virtualization to replace network functions from hardware appliances with virtual server. It reduces cost and accelerates service deployment for network operators by decoupling functions like firewalls or encryption. Virtual servers may be deployed on COTS (Commercial Off the Shelf) hardware.

With the rapid development of NFV applications, a new framework was required to identify functional blocks and interface points. ETSI proposed the ETSI NFV Framework to ensure a consistent approach to NFV across different vendors [8]. This gives a great flexibility for creating and managing networks. When a new function is needed, an administrator can easily create a new VM with the desired image and in a matter of minutes can implement the new function in the network, instead of buying hardware and setting it up manually. This approach saves equipment costs and configuration time.

NFV also presents other benefits such as reduced power consumption through consolidating equipment, easier resource managing considering there are less machines to operate and encourage system openness since the appliances are now software based [8].

As with any new technologies, implementation of NFV presents several challenges. Migrating from dedicated hardware to virtualized software may be a complex procedure; it requires a change in the network architecture. Furthermore, installing new Virtualized Network Functions (VNF) often means a co-existence with legacy platforms that are not always compatible. On top of that, virtualized software has usually lower performance than dedicated hardware implementations. Moreover, since NFVs are run on top of hypervisors, there is a general concern about security and resilience; there are more moving parts running in the same system that can cause system failures and security breaches. A security certified hypervisor is required to provide a safer and more stable base for the VNFs to run on [8] .

2.3 Software Defined Networking

Traditional switches have the data and control plane in the same physical device. This allows for a fast and reliable operation since each switch is independent from all others in a network. However, distributed architectures are much harder to optimize since they rely on local algorithms that usually do not have a global network view. Moreover, in large networks, managing and configuring every switch manually can be tiring and cumbersome.

Software Defined Networking (SDN) offers an architecture that decouples the control and data planes from switches, centralizing all control decisions in a single computing platform. The control plane oper-

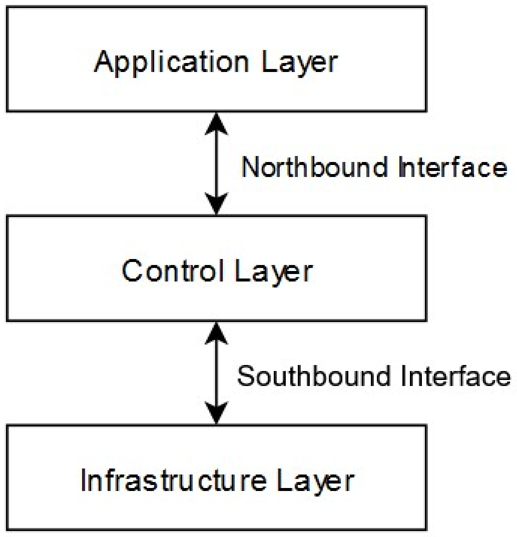


Figure 2.2: SDN architecture layers.

ates the logic and makes decisions about network traffic management and optimization, while the data plane focuses on forwarding the packets based on the control plane's decisions. The data plane uses flow tables that contain actions to execute when a match is made. Flow tables can be dynamically populated with rules by the control plane according to operation requirements. This architecture provides a strong foundation, since decisions depart form a single controller who has a global and unified view of the overall network, and the decisions are no longer distributed all over the network.

Separating the data and control planes from the same device originates a new network architecture. It consists of three layers vertically aligned. The central layer is the Control Layer, where network traffic is managed. It receives instructions from an Application Layer through the Northbound Interface. The bottom layer is called the Infrastructure Layer, it is built by network devices and contains the data plane. The Control Layer uses the Southbound Interface to communicate with the Infrastructure Layer sending instructions for how to manage network traffic. Figure 2.2 shows an illustration of the three layers present in a SDN architecture.

The SDN controller, also known as Network Operating System, is the key component for a SDN network to work properly. It can be found in the Control Layer and consists of an application that manages network flows [9].

SDN can work with hardware switches in a physical network. However, this approach conflicts with the spirit of SDN, which goal is to centralize network intelligence in software. It is commonly implemented in virtualized networks with software switches alongside NFV to benefits from virtualized network functions and have a fluid software-only network orchestration.

2.3.1 OpenFlow

The development of SDN was based on the decoupling of the control and data planes. The interface between these two planes is called the Southbound Interface (SBI). OpenFlow (OF) was the first widely adopted protocol for the SBI communication.

The OpenFlow protocol is managed by the Open Networking Foundation (ONF), a consortium leading the adoption of open network standards such as SDN and OpenFlow, as well as other technologies related to automatic network management. OpenFlow Development started in 2008 [10] at Stanford University and since its initial inception many versions have been launched. Each version introduced new network protocols which tried to complement previous versions. The first version only supported four common protocols, while the latest version of OpenFlow lets us add and delete forwarding entries for about 50 different header types [11].

OpenFlow can add, update, and delete entries in the flow tables of network switches. Each entry has a set of match keys and instructions to follow on matching packets. This can either be done proactively, by populating tables before packet hits the network node, or reactively, in response to arriving packets that do not match any of a preloaded set of rules.

OpenFlow is a powerful protocol that gives the user full control of packet forwarding rules and the network data plane through the control plane.

However, OpenFlow has some known shortcomings. Supporting many protocols makes the protocol almost universal, since it can be used in nearly every situation possible. However, it increases the development complexity. In fact, each OpenFlow version requires new OpenFlow switches, since these too should implement every protocol addressed by it. This makes it harder for vendors to adapt, especially if some of the protocols are very specific and rarely used. This results in bloated network switches which customers may not want due to configuration complexity. Usually, users only need a limited amount of network protocols to run in their network.

OpenFlow supports the communication between Controllers and Network Nodes and therefore the same OF version should be implemented in both. The control plane uses TLS on top of TCP with default port 6653 to populate the forwarding flow-tables of Network Nodes, TLS was chosen because of security concerns [10].

2.3.2 SDN Controllers

As stated above, SDN controllers are the main software application orchestrating SDN Networks. Controllers have a global view of the entire network and provide full control over every network node. Since a centralized controller is a single point of failure, it is common to see two or more controllers in large and sensitive networks. The second controller takes the role of a backup controller in case of failure of the first one.

Controllers have three modes of operation [12]:

Reactive The flow-tables of network switches are initially empty. When an unknown packet arrives in a switch interface it is sent to the controller, this analyses the packet and sends a flow table entry, also known as flow rule, as a response to the switch with rules with what to do with that specific type of packets. This approach has the most control over the flow rules installed but adds a setup time with any new transmission.

Proactive The proactive approach populates the flow rules of the switch on its startup. This does not have an additional setup time for new traffic; however, it requires harder management and beforehand knowledge about the type of traffic that flows through the network.

Hybrid This offers the best of the previous two and consists of installing a set of preconfigured flow table entries. At the same time, it has a real-time connection with the controller for new traffic that might not have an action in the switch's flow tables.

There are many SDN Controller applications written in a variety of programming languages. Some are open-source, and most have modular characteristics for adaptability with a wide range of network devices and protocols. In the next section we will introduce the P4 programming language, some controllers that currently support P4 are ONOS and OpenDaylight [13, 14].

2.4 Data Plane Programmability

Control plane programmability had a rapid evolution in the past 10 years and brought many benefits for SDN networks. However, data plane devices have remained nearly the same: a static platform that is essentially "dumb" and only forwards packets based on the input it receives from the control plane. This model has also contributed for limited data plane development.

In order to overcome these limitations, a new "top-down" model was proposed that introduces the concept of data plane programmability [11]. Instead of having a limited switch and adapting the control plane to its capabilities, as in a "bottom-up" model, this model suggests a dynamic data plane where the programmer can edit the forwarding plane with custom protocols and features. That way the control plane does not need to worry about the switches' or the SBI limitations.

Tofino [15] is probably the most successful switch with a programmable data plane. It was developed by Barefoot Networks, with the objective of giving network vendors the freedom to develop new switching software. Barefoot called it the world's fastest, at 6.5Tbps. Since then, Intel has acquired Barefoot Networks and launched Tofino2 [16].

Programming the data plane gives the user much more detailed control over how packets are processed. New protocols and features can be implemented and managed out of the scope of OpenFlow or other rigid SBI protocols.

The next section will focus on P4 and its features regarding data plane programmability.

2.4.1 P4 Programming Language

P4 (Programming Protocol-independent Packet Processors) is a programming language used to specify how a switch's data plane processes packet forwarding. While being protocol independent, it is also target independent, which means that the same code can run in different hardware platforms, provided that they adhere to a common hardware specification [4].

The P4 programming language was initially proposed in [4] and later developed by the P4 language consortium, a non-profit organization formed by a group of network engineers. It has since joined the Open Networking Foundation [17].

Originally proposed in 2014, P4's objective presented three main goals:

Reconfigurability Let programmers change how packets are processed.

Protocol Independence Free switches from unwanted protocol integrations.

Target Independence For hardware interoperability.

In 2016 a revision of the language was presented and originated a new version referred to as $P4_{16}$. The evolution to a new version was necessary because the original version was a complex language with over 70 keywords. Language refactorization resulted in a simpler language with less than 40 keywords. Also, the 2016 version was successful to decouple the core language from the switch architecture, a problem that kept the earlier version from being fully target independent. The original version is today in maintenance mode only and is referred to as $P4_{14}$ [4, 18].

When OpenFlow was first engineered there was a big difference in processing speed between fixed-function ASICs (Application-Specific Integrated Circuit), and programmable switches. The latter were not viable for networks with fast paced traffic since they were much slower. Nowadays however, there are programmable chips that are as fast as fixed-function ones, called PISA (Protocol Independent Switch Architecture) chips. These are very versatile, and its flexibility allows for programmers to implement and try new network protocols as soon as they are designed [11].

Because P4 is used to program the data plane in software switches, it is usually used in SDN networks. It takes advantage of the fact that SDN decouples planes to program the data plane without worrying much about the control plane. It is an appealing addition for SDN networks that need data plane flexibility. P4 can be used to modify already existing protocols and add extra features or include meters for network statistics. P4 switches can also be programmed to implement new protocols, which is an advantage over fixed-function switches for developers that cannot wait for hardware manufacturing to test their algorithms.

Although P4 is commonly used with SDN, it can also be used in standalone mode with both the data and control plane in the same system. This provides great flexibility, P4 can be implemented independently of the network architecture to bring the benefits of Data Plane Programmability.

The language is very versatile and can adapt to most network protocols including OpenFlow [11]. This is useful for compatibility with conventional SDN architectures based on OF. However, to control custom protocols programmed in P4 a new communication protocol needs to be engineered.

2.4.2 P4 Runtime

P4 Runtime creates and API based on a P4 Program and can be used to control any kind of switches, from fixed, to completely programmable. Although P4 Runtime and the P4 language have similar names and are related, they are separate open source projects. The P4 programming language is used to program the data plane of programmable switches, but it can also be used to specify the behavior of existing devices [11].

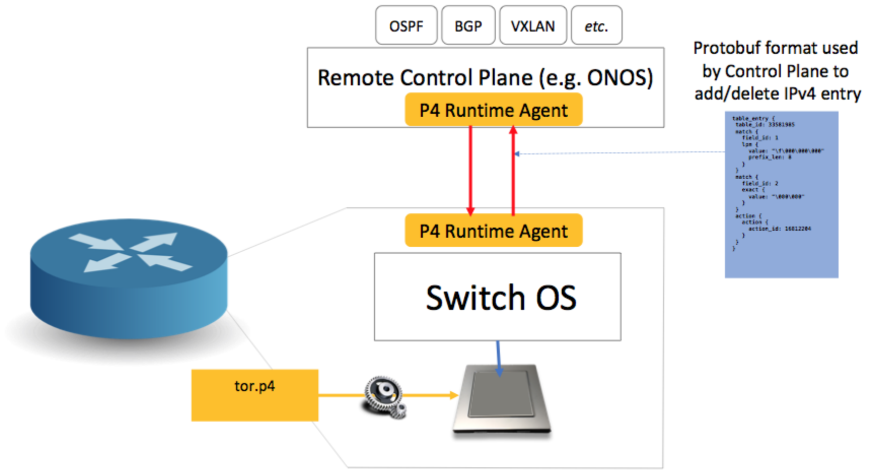
P4 Runtime is an API originally engineered for the SBI communication between a SDN controller and a P4 programmable switch [19]. It was designed to let programmers control any kind of forwarding plane, from fixed-function or programmable switch ASIC, to software switches running in a virtualized network. The master framework is fixed, which means the same API can be used to control a variety of different switches. When programming the data plane and adding new features and protocols to the P4 program, the P4 Runtime API is automatically updated. Its schema is extended to include the new features in the switch so it can be successfully managed by the control plane.

Proprietary APIs from closed switch chips are fixed and cannot be changed. This not only makes it harder to implement new protocols as it is also harder to manage networks that have devices from multiple manufactures with different APIs. P4 Runtime offers a customizable and standardized API for the Southbound Interface. This is a great advantage over OpenFlow, the predominant protocol for the SBI.

To implement the P4 Runtime API in a SDN network, the controller must install a service running gRPC with default TCP port 9559 to which network nodes can connect and exchange Protobuf messages [20]. Figure 2.3 shows an illustration of the communication between a Remote Control Plane and a programmable Switch running the P4 program “tor.p4”.

2.4.3 Switch Architectures

The switch architecture is what defines the building blocks that condition packet processing since they are first received through an input port, until they are discarded or sent to an output port. In physical switches, the architecture is implemented in the hardware and cannot be changed. It benefits from


OSPF
BGP
VXLAN
etc.
Remote Control Plane (e.g. ONOS)
P4 Runtime Agent
Protobuf format used by Control Plane to add/delete IPv4 entry
P4 Runtime Agent
Switch OS
tor.p4

```
table_entry {
 table_id: 33581985
 match {
   field_id: 1
   lpm {
     value: "\f\000\000\000"
     prefix_len: 8
   }
 }
 match {
   field_id: 2
   exact {
     value: "\000\000"
   }
 }
 action {
   action {
     action_id: 16812204
   }
 }
}
```

Figure 2.3: Using P4 Runtime API with a remote control plane. Reproduced from [19].

fast packet forwarding due to chips custom designed to analyze and make decisions based on packet information. Software switches are virtualized, and its architecture is defined through a programming language (*e.g.*, C++). These do not benefit from hardware acceleration, but they have the advantage of being flexible and can adapt to any switch architecture possible.

The first version of P4 was based upon an abstract forwarding model that could not be modified. This is the standard architecture for $P4_{14}$, it is also used in $P4_{16}$ referred to as V1Model. It consists of an initial parser that deconstructs the incoming packet, and a set of control blocks that are programmed with actions to perform based on the information of the packet being processed. Recently, a more mature and stable version of the language was released. $P4_{16}$ let the programmers define a custom architecture based on their needs. It is still compatible with the previous version of the language by supporting the V1Model architecture, which is included in the P4 compiler. This provides a straightforward auto-translation from the first version programs to the latter.

$P4_{16}$ can adapt to many architectures. The P4 language Consortium created a standard called Portable Switch Architecture (PSA). This is a community-developed architecture that describes common capabilities of a network switch. It was designed to be of general use across multiple architectures by extending its base code, or to be used alone. The first half of the model has a similar architecture to V1Model, it consists of an initial parser that identifies headers in a packet, followed by control blocks with tables that perform lookups based on header information until it matches with a table entry, every entry is associated with an action to be performed. The packet is then deparsed and delivered to the next block. Unlike the first model, PSA is divided in two segments. The first one is the ingress pipeline, when this is over the packet is processed by a Traffic Manager, in this case the Packet buffer and Replication Engine, where a packet can optionally be replicated to multiple egress ports and stored in the packet buffer. The second one is the egress pipeline with similar control blocks to the ingress pipeline that a

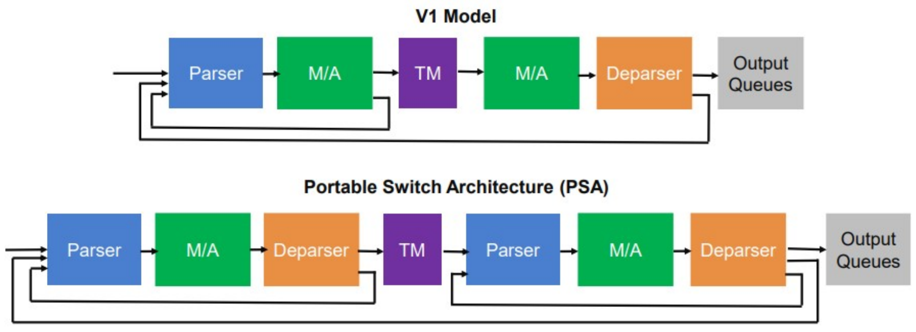


Figure 2.4: V1Model and PSA target architectures.
Reproduced from [22].

packet must pass through before it is deparsed and queued to leave the pipeline. Figure 2.4 shows an illustration of the differences between the V1Model and PSA architectures [21].

The P4 Consortium built a reference software switch, Behavioral Model v2 (BMv2), capable of running P4 programs. The switch consists of three variations. The first variation, called `simple_switch`, consists of a Thrift interface for communication between the control and data planes. Complementing this variation, `simple_switch_grpc` was created to include a more modern and standardized P4 Runtime API. It was also created a variation for running PSA model programs, `psa_switch`. BMv2 is only meant for development purposes and it has significantly less performance than other software switches such as Open vSwitch. Therefore, it is not recommended to be used in production [23].

For production ready solutions, Stratum is an open source switch OS managed by the Open Network Foundation, capable of running P4 programs. It will be analyzed in higher detail in section 4.1.3 [24].

2.4.4 P4 Language Specification

P4 is a domain-specific language for expressing how network packets are processed by the data plane. Its syntax is loosely inspired on the C programming language, including reserved keywords designed for packet analysis. These are some of the most important core abstractions provided by the P4 language [18]:

Types Because P4 operates with packet processing, it often requires changing parameters at bit level, it includes primitive types such as `bit<N>` that allocates **N** bits for an unsigned integer. Internet packets are composed of headers and payload, headers are essential for packet processing and P4 has a dedicated keyword dedicated to these. Listing 2.1 shows an example of an Ethernet header defined in $P4_{16}$.

Parser Every P4 pipeline must include a parser. This is usually the first programming block. Here the programmer expresses a state machine to extract packet headers. It begins with the `start` state, in each state the program analyses the header, extracts necessary information, and transitions the

Listing 2.1: Ethernet header definition in P4$_{16}$

```
header ethernet_t {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}
```

program to another state until it reaches the `accept` or `reject` state, these are final, and the program moves to the next pipeline block.

Tables Fundamental in any switch program, tables are the main component that correlate a set of keys with an action. Users can define a set of keys such as a packet MAC destination address or the ingress port to associate with an action. Additional arguments can be given such as the maximum table size, for memory allocation, and a default action for packets that do not have any entry in the table. Table entries can be hardcoded in the program for well-known behaviors, or they can be updated in real time through either a Thrift interface or a P4 Runtime service running in a SDN controller. Tables are very versatile and can adapt to any kind of scenario like simple routing tables, access-control lists, or even complex multi-variable key decisions for advanced protocols.

Actions In a table entry hit, the program executes the corresponding action on the packet being processed. These are code fragments that describe how packet header fields and metadata are manipulated. P4 has primitive actions like `drop()`, that drops the current packet and no more processing is done. Usually, actions are the most complex part of a P4 program as they include the most logic. In listing 2.2 an example action `ipv4_forward` performs the actions necessary to forward an IPv4 packet to the corresponding interface port. The parameters `dstAddr` and `port` are saved in the table entry together with the keys that trigger this action. In this action, first the switch is told what port to send the packet to by modifying the `standard_metadata`, then the packet's MAC address is updated, and finally it decrements the TTL parameter in the IPv4 header.

Listing 2.2: IPv4 packet Forwarding action

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

Control Flows Switch architectures like V1Model define a pipeline structure with an initial parser block and control blocks, the latter have a dedicated control keyword. Inside these blocks the user can define tables and actions that are managed by the control plane, editing the tables can change the behavior of the data plane. Every control has an apply block, in this block is defined a set of if-else statements to decide which table to apply the packet to, based on header inspection. Control blocks also support deparsing packets when ready to depart from the switch pipeline.

Extern Extern objects are provided by vendors and work as the interface for P4 programs to interact with the switch's core. Functionalities provided in externs are not programmed in P4, they can be considered as "black boxes" for P4 programmers. In BMv2 the externs are programmed in C++ and are public since it is an open source project. Often architectures use externs as an interface for P4 programs to access variables defined in the switch OS. V1Model has a simple Standard Metadata set useful for basic P4 programs, it can be accessed any time inside a control block and has variables such as `ingress_port` that indicate the port on which the packet arrived, and `egrees_spec`, used in listing 2.2 example, to indicate the port to which the packet should be sent to.

Main Declaration Finally, every P4 program must define an instance of the architecture called `main`. The parser and controls are defined in the order they are executed and must be coherent with the switch architecture.

2.5 Load Balancing

The great expansion of computer networks in recent years brought much more traffic than the conventional web servers could handle. Traditional services only had one server handling all incoming requests, server administrators quickly understood that a new method was bound to be implemented to distribute the high load between more resources. Thus came Load Balancing.

Load Balancing refers to the technique to distribute load among a set of resources to get the best performance possible of the global system. In computer networks, load balancing can be viewed at two levels: to distribute network traffic through multiple paths, or to balance client requests to multiple servers. In this project we will focus on the latter.

There are many load balancing algorithms, which can be grouped in **stateless** or **stateful** algorithms [25]. Stateless algorithms are those that do not take into consideration the state of the system, while Stateful algorithms, on the other hand, consider the load of the system and make real time decisions to distribute the load with the objective of achieving the best overall performance. Stateful algorithms are usually harder to implement but offer better load balancing results, these can take advantage of one or more parameters from the network as the decision factor for the algorithm. Some of the parameters can be associated with properties of the servers, for example: the CPU load time, the number of connections with clients, or the average request response time.

Probably some of the most common load balancing algorithms are the following [26]:

Round Robin The Round Robin algorithm consists of ordering all the servers in a list, and distributing the client requests in rotation. If we have three servers, the first three requests will go, in order, to the three servers, considering that we have servers one, two and three. When the fourth request arrives, as there is no fourth server, the list will start at the beginning and the request will go to the first server.

Weighted Round Robin Similar to the Round Robin algorithm, the Weighted Round Robin algorithm also orders the servers in a list and serves them in order and rotation. Unlike the previous algorithm, the requests are not distributed equally among the servers. The network manager may set different weights to each server. In an example with three servers and four requests, where the first server has a weight of 2 (two), and the others have a weight of 1 (one), the first server will handle the first two requests, while the third request will go to server two, and the fourth request to the third server. Like the previous algorithm, this process repeats when the list reaches the end. This strategy is an improvement of his predecessor since more powerful servers can be attributed with more requests.

Least Connections As the name of the algorithm suggests, this algorithm is considered stateful and relies on the number of open client connections. The load balancer keeps track of how many connections each server has and, when it receives a new client request, assigns it to the server with least connections. Unfortunately this algorithm has a downside, it takes into consideration the number of active connections of each server, but some connections might be much heavier than others, making it imbalanced if a server is attributed too many heavy connections.

Resource Based This algorithm listens for reports of agents installed in the servers that share the load status of the servers. This load is dependent of the load balance application and can take the form of one or several parameters, such as CPU load time, or the server memory. When the load balancer receives the report from all the servers, it processes the values and makes a load balancing decision.

2.6 Related Work

Data Plane Programmability is a powerful concept that completely changes the process of designing a network. This section presents papers related to P4 and load balancing.

In the paper by Katta el al. [27], a distributed load balancing algorithm implemented in P4 is presented. The algorithm is called HULA. Equal-cost multipath (ECMP) is a routing strategy that forwards packets through multiple paths with the same cost, this is a common routing strategy found in data center networks. ECMP has some known limitations like degraded performance and poor reaction to link failures. Some work has been done to overcome ECMP's limitations by developing congestion aware load balancing techniques. HULA focuses on overcoming two key limitations of these techniques, first by reducing the amount of switch memory required in each node, and second by implementing the algorithm in programmable switches instead of custom hardware. In conclusion, this paper presents a stateful load balancing algorithm in a decentralized network.

Another article was later published by Eyal Cidon et al. [28] presenting AppSwitch, a load balancer switch for key-value storage systems. AppSwitch does not implement a load balancing algorithm itself, it rather works like a cache system. This project's topology consists of one client, one server, one proxy, and a switch connecting all of them together. The system uses a communication protocol called Memcached, the client uses the protocol to performs key-value requests. First the requests are routed to the proxy, where a hash operation finds a suitable server for every key. When the client receives that information, it proceeds to connect with the server indicated by the proxy, which sends the value corresponding to the requested key. This paper proposes an application implemented in P4 that eavesdrops the network traffic for packets using the Memcached protocol, and for every key discovered it stores the servers indicated by the proxy. With this algorithm, when the client performs a request with a key that was already used in the past, the switch loads from cache which server the client should connect to, therefore reducing the traffic exchanged with the proxy. By greatly reducing the communication with the proxy, this procedure results in nearly half the average request latency. Therefore, this paper introduces a data plane cache solution to increase the performance of a stateful load balancing mechanism.

Recently in 2020, Chih-Heng Ke et al. [5] released a paper demonstrating a load balancing system with P4 switches in a Software Defined Network, using a centralized controller. The test topology consists of a client connected to a P4 switch that manages the flow to up to four servers. The motivation for this paper relies on the time consuming encapsulation and decapsulation of conventional load balancers like Linux Virtual Server (LVS) and HAProxy. It proposes the use of Data Plane Programmability with the P4 language to make load balancing faster, by moving the load balancing algorithm from the control plane to the data plane. During their work, a series of load balancing algorithms were used to test which has the best performance. The algorithms tested were: connection hash, random, round-robin, and weighted round-robin. They used the average request response time to compare the algorithms, and reached the conclusion that the best algorithms were the round-robin and weighted round-robin, depending if the CPU speed of the servers were equal or not, respectively. This work presents some interesting features like periodic health checks done by the controller, which can inform the data plane for server faults, and the ability for the load balancing switch to continue working if the control plane fails. However, after some inspection to the project's source code, we found that the algorithms performed are hard coded in P4 and very hard to alter in case the topology changes. In summary this paper presents a project with four different stateless load balancing algorithms in a centralized network capable of fault tolerance.

It is interesting to note that the first two papers mentioned in this section were co-authored by Jennifer Rexford and Nick McKeown, respectively. These two authors are also co-authors of both the original

OpenFlow and P4 whitepapers [4, 10]. This gives some credibility to the work mentioned in this section, and shows that our work makes a contribution to the technological advancement of computer network topics like Software Defined Networking and Data Plane Programmability.

3

Proposed Solution

The objective of this work is to develop a load balancing application with SDN and P4. Based on the articles mentioned in section 2.6, we propose to develop a load balancing application that benefits from the best features of the mentioned articles, while at the same time trying to overcome some of their limitations. We will focus mainly on the paper by Chih-Heng Ke et al. [5].

Considering the work by Ke, we propose some modifications. Firstly, by introducing a production-grade SDN controller with features like code modularity and configurability. Secondly, by improving upon the data plane P4 code, making it more generic and compatible to any number of servers in the system. Finally, by replacing the stateless load balancing algorithm, by a stateful one, which should offer better performance. In summary, this work shall consist of a stateful load balancing algorithm in a centralized network with fault tolerance capabilities.

The main scope is to produce a system with high quality that could be incorporated in a data center network. Since data centers are commonly found to work with centralized networks and load balancing systems, we consider that it would be a great achievement if our project could be ready for an environment like that. With that in mind, we aim to use SDN, since it is very common among data centers and has useful functionalities like a unified view of the network. Also, we want to incorporate a modern controller that is widely accepted and has modular applications, in order to ease the implementation and the change between multiple load balancing algorithms in a transparent way to the network.

In order to develop a load balancing SDN application, it is a good practice to separate the logic between the control and data planes. The control plane contains the part of the algorithm that is responsible for choosing the weight of each server, while the data plane has a flexible code that distributes the load between multiple servers considering the weight of each server, but is transparent to the decision algorithm itself.

With this approach, the data plane does not have to communicate with the controller for each packet it receives, since it already has a table indicating the weight of each server. In one hand, the data plane is responsible for having global control variables that manage the balancing flows independently, this way the decision algorithm is abstracted. In the other hand, the control plane is responsible to update the data plane every time there is a change in the decision algorithm. Additionally, if a modular controller is implemented, it should be easy for the user to change the load balancing decision algorithm in real time without the knowledge of the data plane.

To simulate a real world scenario, we propose that the servers run a HTTP socket listening for GET requests. This means that the switch handles TCP traffic. The data plane program needs to have a basic understanding of how TCP connections works, this means that the switch is not performing load balancing packet wise, but instead request wise. In other words, for each new packet inbound to the switch, this performs a hash calculation using the source IP address and the source TCP port, on the first packet of each connection it attributes a new server for the hash and saves the hash-server pair.

That way, when a packet arrives and the hash calculation is already saved in switch memory, it loads what server is performing the request and sends the packet to that server. It is also important to note that in the end of each TCP connection, the header's FIN flag is active, when the switch detects this flag it proceeds to delete the hash-server pair from switch memory since it will no longer be used.

Since the application is balancing load between a set of servers, the internal server IP addresses need to be invisible to the clients. A strategy is implemented where the switch is attributed a public virtual IP address to be used by the clients. For each client request, the destination IP address is the switch's virtual IP address, instead of the physical IP address of the servers. The switch is responsible for modifying the packet headers of each request to change the destination IP address for the server's IP address that is assigned by the load balancing algorithm. Also, when a server responds to the client, the response packet also needs to be modified, the switch replaces the source IP address by its own virtual IP address. It is important to note that these header modifications make both the IP header and the TCP header checksums invalid. Therefore, before the switch sends the packet through the outbound interface, it needs to recalculate the header checksums. If this last step is skipped, the recipients of the packets mark them as invalid and discard the packets.

Regarding the load balancer algorithm, we decided to take the most benefit out of SDN and develop a stateful load balancer, since these often offer a better performance than stateless load balancers. The chosen algorithm is the resource based type, and the decision parameter to use is the average server request response time. Ideally the data plane should have a metric to calculate the average response time of each TCP connection, but since the P4 language does not support this functionality, we decided to use a probe based approach from the servers. In our approach, every server is responsible for sending a periodic report to the controller with the average response time of the their most recent requests. These probes are sent with the UDP protocol to the switch, that redirects them to the controller. When the controller has received a report from all the online servers, it processes them and makes a decision about how the weights should be distributed to each server. When the decision ends, the controller installs new load balancing flow rules in the switch, replacing the previous flow rules.

Lastly, our application is also capable of supporting fault tolerance. We use the advanced SDN capabilities of our remote controller to detect changes in the topology, these changes can be the addition or removal of servers connected to the switch. When the switch detects one of these changes, it proceeds to restart the load balancing algorithm. When the algorithm restarts, it forgets about the weights previously attributed to each servers and assumes that all the servers have the same initial weight. When the time passes, the controller receives the periodic server probes and adjusts the server weights accordingly.

A proposed topology for the proposed application can be consulted in fig. 3.1.

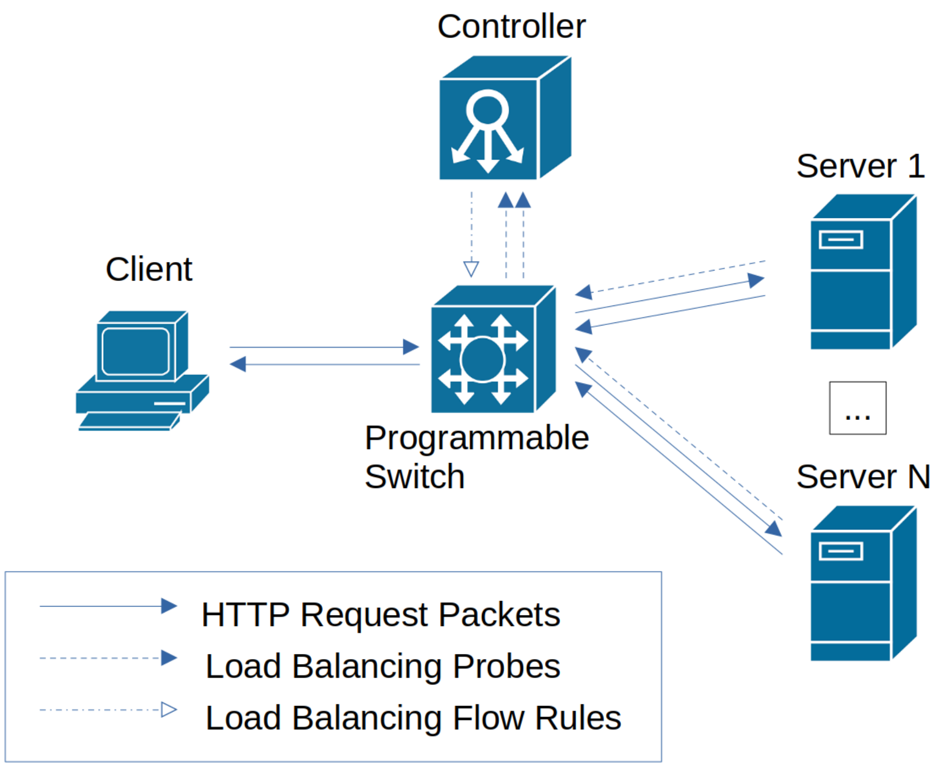


Figure 3.1: Proposed Topology.

4 Implementation

Contents

This projects aims to develop and test a Load Balancing algorithm in a SDN network with P4 compatible switches. This Chapter describes the details of the implementation used to develop this project. In section 4.1, we introduce the tools used to create a development and testing environment. In section 4.2, we present the initial setup necessary for the tools described in the previous section work together. Section 4.3 shows the steps taken to enable packet routing. Finally, in section 4.4, we explain the details of the load balancing algorithm.

The application was developed in GitHub and can be found in [29].

4.1 Development Environment

As explained in the previous section, this project consists of an advanced SDN application. The first step to develop this project is to setup a solid development environment with all the tool necessary. We decided to use a virtualized environment since it would be easier to develop and test than a physical environment.

While we were researching for similar projects and improving our skills with the P4 language before starting our project, we found a GitHub repository that met all the necessary requirements to develop a SDN application with a programmable data plane network using the P4 language [30]. The development environment is illustrated in fig. 4.1.

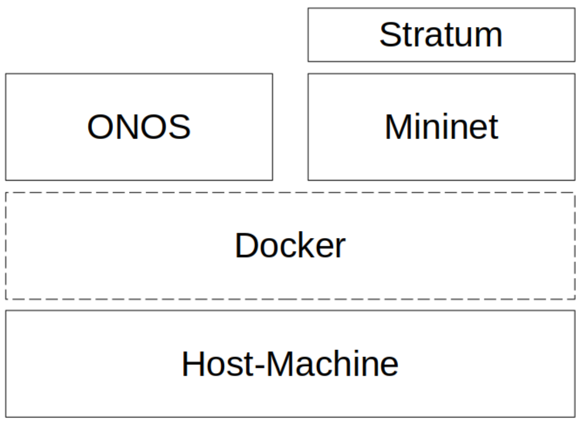


Figure 4.1: Development and Testing Environment

4.1.1 Docker

For the virtualization of the environment, Docker is the obvious choice. It works with containerization, a much lighter approach to virtuazliation than virtual machines.

Docker is a set of tools to create and manage containers. These containers are lightweight forms of virtualization commonly used to virtualize a set of programs and services in an isolated environment.

Every docker container starts with a Dockerfile, this is a text file with syntax that includes instructions to build docker images. The instructions can setup environmental variables, files to be included, and what commands to execute on startup. The build command can be used to create an image; most images can be found in Docker Hub [7], the world's largest library for container images.

Docker containers can be easily set up and managed with the help of Docket Compose, a tool for managing multiple containers simultaneously. We used Docker Compose to set an environment that runs two services: a network simulator, and a SDN controller. Thanks to the ease of use of Docker, both containers are in the same network and can communicate with each other, making it ease to connect the switches inside the network simulator to the SDN controller.

4.1.2 Mininet

One of the containers used in this project runs Mininet. Mininet is a network emulator designed to run on Linux using features such as network namespaces and virtual ethernet interfaces. Written in Python, Mininet is portable and can be run in most laptops by either using the Mininet command-line, or the Python API. It can run large networks with custom switches, hosts, and controllers. Customizing a network is easy by creating custom node types and designing your own topology. The programmer can create his own custom topology with a Python script or use one of Mininet's parametrized topologies. When generating a network, Mininet creates a new Linux local namespace for each host where processes are isolated from the global namespace, each namespace has its own set of network interfaces, IP addresses and routing tables. [31]

By default, Mininet provides Open vSwitch (OVS), an open-source switch that supports multiple protocols, it is lightweight which makes it perfect for most Mininet emulation runs. OVS supports OpenFlow, making Mininet compatible with SDN networks and the default testbed for running, testing, and debugging this kind of networks with a controller. Although Mininet contains a custom simplified controller, it is often used with an external controller.

Due to Mininet's high configurability via its open source Python code, any programmer can extend its code using Object Oriented Programming (OOP). This is helpful for network node developers testing their applications. For instance, one can extend the Switch class to implement BMv2, a software switch capable of running P4 programs.

4.1.3 Stratum

Stratum is an open source switch operating system developed for Software Defined Networks. Developed by ONF it was built to run on data programmable networks, it supports P4 programs, as well as exposing SDN interfaces including P4Runtime and OpenConfig. The motivation behind Stratum is to deliver a white box solution to avoid the vendor lock-in of proprietary silicon interfaces and closed software APIs, while at the same time enabling an easy integration of devices. In conclusion, Stratum proposes to include interfaces that give full lifecycle control, configuration and operations, therefore giving SDN a broader scope. [24] Also, it is worth mentioning that Straum's source code provides a Docker image that can execute Mininet using Startum as the default switch [32].

4.1.4 ONOS

The second container used in this project runs an image of ONOS. Also known as Open Network Operating System, ONOS is a open source SDN controller developed by ONF and compatible with P4 switches and the P4 Runtime API. Developed in Java with modular applications, it follows four design goals [33]:

Code Modularity ONOS includes a set of independent applications that can be used to introduce new functionalities. The network manager can choose which applications to activate and also develop their own applications to introduce new functionalities in the SDN controller.

Configurability Thanks to its REST API, web dashboard and ssh console, ONOS is higly configurable with the ability to load and unload various features either at startup or at runtime.

Separation of Concern Each module should have clear boundaries to create independent subsystems.

Protocol agnosticism The controller itself is independent of communication protocols. ONOS should be compatible with all protocols through the southbound API. If it needs to support a new protocol, it should be possible to build a new plugin and load it to the system.

With this design goals in mind, ONOS is a solid SDN controller supporting high configurability trough its interfaces, and great application modularity. With a well documented Java API and a unified view of the network, ONOS provides a high level abstractions through which application can have a good understanding of the network's state. In addition, ONOS API supports network modification events, applications can implement listeners to process these events and have real-time reactions.

4.2 Initial Setup

With the development environment decided, there is still some work to be done before the Load Balancing application can be built. In this section we describe the work done to congregate the technologies mentioned in the previous section, as well as some functionalities built to automate processes that facilitate the development described in future sections. The setup explained in this section was already developed in the base repository, this is an introduction that we found relevant to explain in order to have a better understanding of how the system works.

Firstly, as with any project involving computer networks, the network topology needs to be defined. With Mininet as the network simulator, the network topology was defined with a Python script using the Mininet API. The script is organized in two sections. The first section defines the network nodes using OOP by extending the base classes provided by Mininet and customizing them for compability with data plane programmable networks. Luckily, as mentioned in section 4.1.3, there is not much work to be done in this section since Stratum already has a Mininet switch extension in a Docker image that is being used in this project. By including this Docker image in the Docker Compose, the only thing left to do is to extend the base classes related to the Hosts and Servers. The second section is the definition of the network topology. Here the Mininet API is used to add new nodes such as switches, hosts and servers, and furthermore connect them together with links. When adding new nodes, important setup information should be mentioned, including the IP and MAC addresses, as well as a string identifier.

After the setup of the network simulator, the SDN controller needs to be configured, this project uses ONOS, which is also running in a Docker container. ONOS provides many functionalities out of the box in the form of application modules. The built-in applications loaded in ONOS are the following:

- **gui2**: ONOS web user interface.
- **drivers.bmv2**: BMv2/Stratum drivers based on P4Runtime, gNMI, and gNOI.
- **lldpprovider**: LLDP-based link discovery application.
- **hostprovider**: Host discovery application.

Each application can define other applications as dependencies, it is normal to have more applications activated than only the ones mentioned above when ONOS ends booting. In addition to the built-in applications, a new ONOS application was developed with the Java API to have custom control over the switch's flow tables, and to manage the network. This application is responsible for managing all types of flows in the system, from ARP and IPv4 packets, to the load balancing algorithm.

Since this new ONOS application is built to be generic and run with any kind of network topology, it is not aware of the system built in this project. To instruct ONOS with the details of the application and how it should communicate with the Stratum switches, a configuration file is installed with NETCONF. NETCONF, also known as Network Configuration Protocol, is a protocol used by ONOS as a Northbound interface to install, manipulate, and delete the configuration of network devices. NETCONF it was developed by IETF and is published as RFC 6241 [34].

The Programmable Switch is obviously programmed using the P4 language. A file named `main.p4` contains the code used to control the switch. The structure of this file includes header definitions, and the program pipeline, which consists of control blocks normally found in a V1 Model. In the Ingress control block it can be find the definition of all the flow tables, as well as the logic responsible for processing and routing incoming packets.

To automate all of necessary processes to install and manage the above setup, a `MAKEFILE` file was created. In this file it is defined a series of helpful commands, the most useful and commonly used are described here:

- **start**: Start Mininet and ONOS containers using Docker Compose.
- **stop**: Stop the containers.
- **onos-log**: Show the ONOS log.
- **mn-cli**: Access the Mininet CLI
- **app-build**: Compile the ONOS program and the P4 code.
- **app-reload**: Install and activate the ONOS app.
- **netcfg**: Push the configuration file to ONOS.

In summary, this section shows how the tools and programs chosen for this project can cooperate together and create a Software Defined Network with a programmable data plane.

4.3 Routing

With the environment tools selected and the setup complete, the project can be compiled and deployed. The result is a simulated Software Defined Network with a programmable data plane in a custom network topology. However, no matter the topology, there is no communication between the network nodes. That happens because both the P4 code and the ONOS app are incomplete. On one hand, the data plane needs to include the definition for the packets that are going to be used, as well as to implement the routing tables and develop logic to process and forward network packets. The control plane, on the other hand needs to process some incoming packets from the data plane and install all the flow rules in the data plane's tables.

As it will be explained in section 4.4, the dominant communication protocol used in this project is HTTP. HTTP communications take place on top of TCP/IP connections. For the scope of this project we don't need to parse HTTP header contents, thus the only headers that need to be parsed are TCP and the ones below that, including IPv4 and Ethernet. Additionally, it is also used some communication with UDP, and during the development process it was used the *ping* tool which requires ICMP packets. Finally, in order for the network to function properly, it is also important to include ARP. In conclusion, our network processes a total of six kinds of different headers. In listing A.1 it can be found the definition with the P4 language for all the network headers used in our project.

In order to accomplish communication with the protocols defined above, it was defined a series of match-action tables and logic in the P4 code to process the packets, as well as develop modules in the ONOS application to process topology modification events and install flow rules in the data plane's tables. In the end we could use the *ping* tool and verify basic connectivity between multiple nodes connected through a switch. This was an important milestone in the project, it meant that the communication was successful and all the moving parts of the project were working in harmony.

4.4 Load Balancing Algorithm

With successful communication established in a preliminary testing environment, we could change our focus to the development of the Load Balancing application.

For the load balancing algorithm we decided to use a Resource Based algorithm using the average server response time as the deciding factor.

This section takes into consideration all the work previously described, as well as the solution presented in Chapter 3, and explains in high detail all the work developed to obtain the finishing product of this project, which takes the form of a stateful Load Balancer.

The algorithm can be segmented in two sections, the data plane, and the control plane. In the data plane, a program was developed to be the most generic and transparent possible to the type of load balancing. The control plane, which has modular features thanks to ONOS, is responsible for choosing the type of load balancing algorithm and installing flow rules in the data plane.

4.4.1 Data Plane Algorithm

Firstly, it was developed the Data Plane section of the algorithm. This is programmed in P4 and uses registers and match+action tables to save and load data crucial to the load balancing logic.

A decision was made to implement the logic in the data plane in order to be the most transparent possible to the type of load balancing. Therefore, we have developed a generic P4 program that can be used by any type of load balancing algorithm in the control plane. This was done by defining a concept of weights that we called **flows**. In the beggining of the data plane program it is defined how many flows are being used by the application, each flow connects to one server. The P4 language is oriented for bitwise variables, thus the number of flows can be any number to the power of 2. In Chapter 5 we present evaluation tests that will use between 16 and 128 flows.

As an example: If the program is using 64 flows, and the topology consists of 4 servers, the control plane algorithm could initially assign 16 flows to each server. This means that the first 16 requests go to the first server, while the following 16 flows go to the second server, and so on. When the 65th request arrives, the program starts from the beginning assigning that request to the first server.

Having in consideration the data plane program as a whole, it is important to note that the P4 code processes every packet that arrives to the system. In this section it is only described the procedure for packets that meet the criteria for load balancing, *i.e.* TCP packets that belong to a HTTP connection between a client and a server. In the case that the TCP packets have the client as the source, the destination IP address is that of the switch's virtual IP address, these packets are subject to the load balancing algorithm where in the end the destination IP address is changed by the chosen server's IP address. Otherwise, the packets have a server as the source and in this case the switch is responsible for swapping the source IP address of the servers by switch's virtual IP address. In either case, the switch modifies packet headers, and therefore the packets need to go through a re-computation of the header checksums.

To keep track of how many flows have been attributed in each iteration of the algorithm, there is a register named `next_server_register` which contains a value between zero and the total amount of flows. This register has the same amount of bits as the total amount of flows and is responsible for keeping track of the server to be attributed for the next incoming request. When the register reaches the maximum value, it is performed a binary overflow and it begins again at the first value. In other words, this register has a similar behaviour to a counter of a loop in modern programming languages.

As previously described, the algorithm has a total amount of flows available, to which is server is attributed a different amount. To improve performance, a `range` key was used, instead of the most common `exact` key. A table named `load_balancer_table` was created to keep track of what values are attributed to each server. The `range` key matches a range of values to a single action. In this case, a server can be attributed multiple flows, and therefore there will be multiple keys in the form of the `next_server_register` that will have the same server as output. This table is constantly being updated by the control plane.

Since we are working with HTTP requests, when an inbound packet arrives, it is determined a hash with the source IP address and TCP port. The algorithm then checks in the registers if there is already a server attributed to this flow, if there is then the server is read from the registers and the packet is forwarded to the right server. In the case that the hash does not have any flow attributed, then the load balancing table is used to retrieve the next server to be used. The new flow is saved in the registers for future packets with the same hash, and the `next_server_register` is incremented. Additionally, in the case that a packet has a flow but the FIN flag in the TCP header is active, the algorithm forwards the packet and then deletes the flow from the register since it will no longer be used.

The algorithm implemented in the data plane is detailed in algorithm 4.1.

4.4.2 Control Plane Algorithm

With the data plane logic implemented with a generic algorithm, it's up to the control plane to decide the type of load balancing algorithm to implement. The control plane is also responsible for installing flow rules in the flow tables used by the data plane algorithm.

The algorithm used is a Resource Based Load Balancer. In Resource Based algorithms, the load balancing decision is based upon the resources of the system. In this application, an agent was installed

Algorithm 4.1: Data plane load balancing algorithm for HTTP requests

```
Input: Packet header $hdr$
if $hdr.ipv4.dst\_addr = virtual\_ip$ then
    $tcp\_hash \leftarrow$ hash( $hdr.ipv4.src\_addr$, $hdr.tcp.src\_port$ )
    if $tcp\_hash$ has flow then
        $hdr.ethernet.dst\_addr \leftarrow$ READ
        $hdr.ipv4.dst\_addr \leftarrow$ READ
        if $hdr.tcp$ has FIN flag then
            delete $tcp\_hash$
    else
        $next\_server \leftarrow$ READ
        $hdr.ethernet.dst\_addr \leftarrow$ WRITE
        $hdr.ipv4.dst\_addr \leftarrow$ WRITE
        save $tcp\_hash$
else
    $hdr.ipv4.src\_addr \leftarrow virtual\_ip$
```

in the HTTP servers to periodically send reports to the controller through the switch. A special ACL rule was installed in the switch to forward this types of probes to the controller. The probes are supposed to be lightweight messages that should not increase the traffic between the servers and the switch, additionally some packets can be lost without interfering with the good behaviour of the algorithm, therefore the communication protocol used is UDP instead of the most common TCP.

In our application we decided to use the server's average HTTP response time as the deciding factor for load balancing. We chose this parameter because our objective was to focus on the best User Experience possible.

The whole algorithm was developed around the probe packets sent from the servers. A new module was developed in the ONOS application that processes every probe received by the controller. Every time a new probe is processed, it is saved in an array. When the array contains one probe for each online server, the algorithm filters the array and obtains the servers with highest and lowest average response times. If the difference between the two values is inferior to a certain ratio, the algorithm considers that the load is well balanced and aborts this iteration of the algorithm. Else, if the difference is above said ratio, it removes one flow to the server with highest latency, and add one flow to he server with lowest latency, in an attempt to balance the system.

The output of the algorithm is then processed to generate flow rules compatible with the data plane. These rules have a higher priority than the rules of the previous iteration of the algorithm. When installing the new rules, the old rules are removed from the data plane's flow table. Finally, the algorithm clears the storage of the probes received in this iteration and waits for new probes.

Algorithm 4.2 demonstrates how the Control Plane algorithm works.

Algorithm 4.2: Control plane load balancing algorithm for UDP probes

```
Input: Server name $serverName$, Average server latency $latency$
$serverLatencyStorage$.put($serverName$, $latency$)
if $serverLatencyStorage$ contains all online servers then
    $maxLatency \leftarrow serverLatencyStorage.getMaxLatency()$
    $minLatency \leftarrow serverLatencyStorage.getMinLatency()$
    if $diff(maxLatency, minLatency) < 15\%$ then
        abort
    else
        $maxLatency.server.flows \leftarrow maxLatency.server.flows - 1$
        $minLatency.server.flows \leftarrow minLatency.server.flows + 1$
    $serverLatencyStorage$.clear()
```

4.4.3 Fault Tolerance

This section provides a brief description of how the Fault Tolerance feature was implemented.

ONOS's API provides event listeners that trigger with topology changes. The load balancing application was developed with the ability to change the status of a server when these events are triggered. Anytime an event happens, the control plane decides to interrupt the ongoing load balancing algorithm and restart the load distribution. For example: if there are three servers running and one goes offline, the control plane restarts the algorithm and distributes all the flows by the two remaining servers.

This feature could also have been implemented using the report probes from the agents or by implementing a health check routine by the controller. Instead, we decided to take benefit of the functionalities already developed by ONOS.

5 Evaluation

Contents

In this chapter we demonstrate the tests done to evaluate our load balancing system. In section 5.1, we introduce the objectives that we aim to achieve in this chapter. Section 5.2 introduces the scenarios where the tests will be done. Lastly, in section 5.3, we present the results of the tests described in the previous section.

5.1 Test Objectives

With any algorithm developed in software, there must be a evaluation section to verify its operability as well as its performance. These are the objectives we aim to achieve with our tests.

- Prove that the algorithm developed is properly working by demonstrating connectivity tests.
- Test the algorithm with various types of variables to evaluate their impact on different network situations
- Place the developed system in faulty environments to assess its fault tolerant behaviour

5.2 Test Scenarios

To evaluate the good behaviour and performance of our algorithm, we used a set of topologies and variables to have the necessary tests to cover the minimum amount of possibilities tat we consider enough to evaluate our system for a near real world environment.

For every test performed the topology consists of one client, that simulates enough requests to act like a large set of clients, a load balancing switch connected to a SDN controller, and one to four servers, which will process the client requests assigned by the load balancer.

Since we are performing tests in a virtualized environment, we also decided to develop an artificial system for our servers in order to have a behaviour closer to the real world environment.

The load balancer itself has a series of parameters that will be tested in multiple topologies to test the algorithm and hopefully come to a conclusion about which are the best values for each parameter.

5.2.1 Artifical Server Load

Since we are working in a virtualized environment, it can sometimes be hard to evaluate a system with real world performance values. We decided to make a basic artificial load algorithm based on the work done in [35] and [36]. The resulting algorithm was developed in the python HTTP servers and has a response time distribution that can be seen in fig. 5.1.

Since our system can sometimes have more than two servers and multiple types of tests, the two distribution lines do not always correspond to servers one and two of the system. Instead, servers with

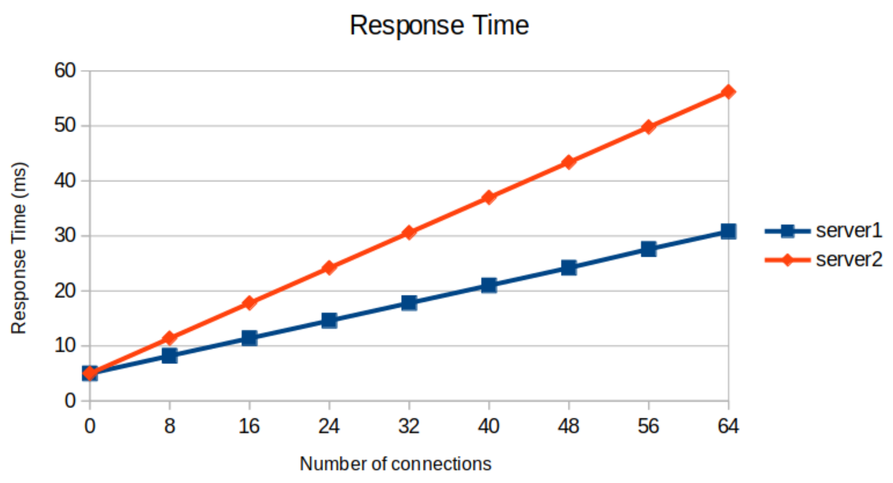


Figure 5.1: Artifical load, request response time by number of active connections

an odd number have the performance of **server1** of the graphic, and servers with an even number have the performance of **server2** of the graphic.

5.2.2 Test Topologies

To test a server load balancing algorithm, the most simple case scenario is to have a system with one client, one load balancer, and at least two servers. We based our testing topology in this simple case, but extended the amount of servers connected to the load balancer to have more broad tests. As such, our testing topology can be analysed in fig. 5.2, where the amount of servers connected to the load balancer can vary based on the tests being performed. The number of connected servers can be between one and four.

While the load balancing algorithm can not work with a topology that only includes one server, since there is only one path to where the traffic can go, we have this option because we are performing comparative case scenarios to prove that having two or more servers is better than only having one single server.

5.2.3 Test Variables

As explained in chapter 4, our application uses a load balancing algorithm that allocates weights represented as **flows** to each server based on their response time. These flows are controlled by the P4 program, as well as the ONOS controller, and are defined before compilation. The amount of flows that the program uses is defined by the programmer before starting the program, and can take any value in the form of a power of 2. We are testing these values for the total amount of flows with different

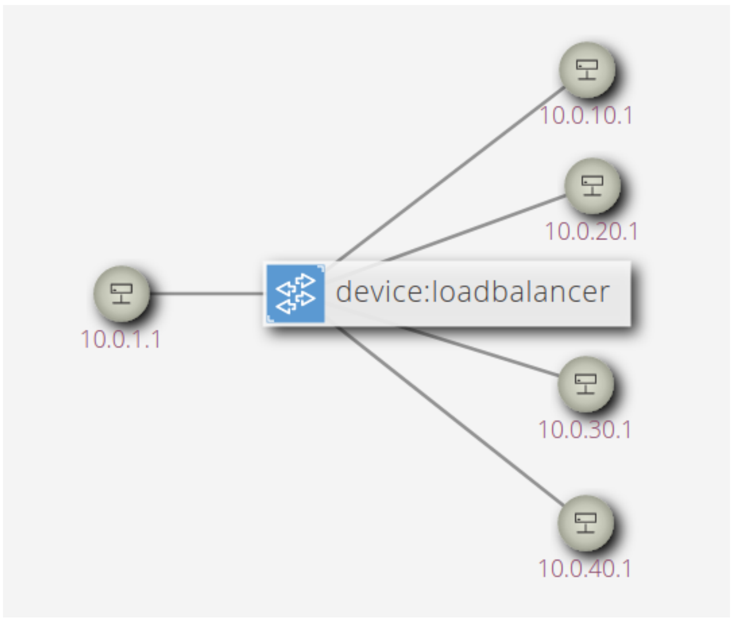


Figure 5.2: Load Balancing Testing Topology.
One client on the left, a load balancing switch in the middle, and four servers on the right.

Servers / Flows	1	2	3	4
16				X
32				X
64	X	X	X	X
128				X

Table 5.1: Number of flows tested for each topology.

topologies, each with a different number of servers, the tests are described in table 5.1 where each test is marked with an **X**.

5.2.4 Test Script

To thoroughly test our application, we decided to develop our own script which will be executed by the client. Our script was written in Python and includes a library to perform HTTP GET requests to our servers, as well as a library to write the output of our tests in a Comma-Separated Values (CSV) file.

The client targets the load balancer virtual IP to perform the requests. It sends a set of 32 sequential batches of 128 requests, and between each batch of requests it waits either for user input, or for a *sleep* function. This wait between requests is unnecessary and would never happen in the real world, we decided to implement this feature so that between each batch of requests the servers can have time to send the load balancing report to the controller and this can evaluate if there are necessary adjustments to balance the system. With this behaviour we can have a more clear understanding of what is happening behind the hood of our application and show test results that are easier to read.

Lastly we save all of the test results in a CSV file with the values gathered for each requests. The values are the following:

- **id** The overall ID of the request, this can be a value between 1 and 4096.
- **run** The batch of the request, a value between 0 and 31.
- **runID** The ID of the quests for the corresponding batch, a value between 0 and 127.
- **server** The server that processed this request, for example *server1*.
- **elapsed** Response time of the request.

With all these values successfully saved in a CSV file, we used LibreOffice Calc to create Pivot Tables, and plot the output with visual Charts.

5.3 Test Results

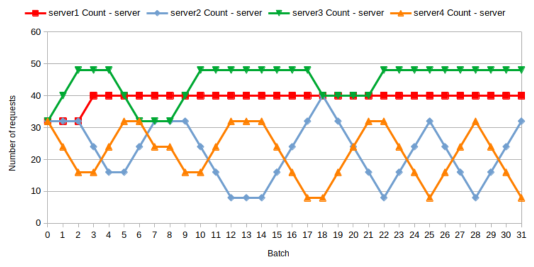
This section presents the results obtained in the tests described in section 5.2.

As previously stated, we designed our application algorithm with a couple of variables in mind, with this tests we aim to explore multiple values for these variables and reach a conclusion about which are the values that give the best performance. We will also test the overall performance of our system given a different number of servers to load balance.

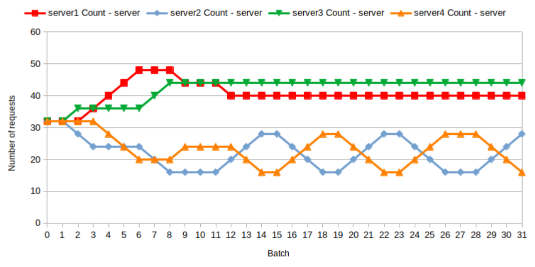
5.3.1 System Tests

Our application balances the load between multiple servers using a weight variable, in our project we refer to this as *flows*. There are two important aspects to this component, the flows attributed to one server, and the total amount of flows installed in the system. Since the P4 application is working with bitwise variables, we found both easier to developed and more performant to use values of the power of 2 for the total amount of flows in the system.

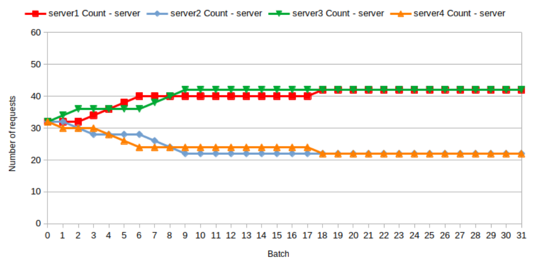
For testing the best values for the total amount of flows, we decided to use our standard load balancing testbed topology with four servers. The values that we will be testing will be 16, 32, 64 and 128.



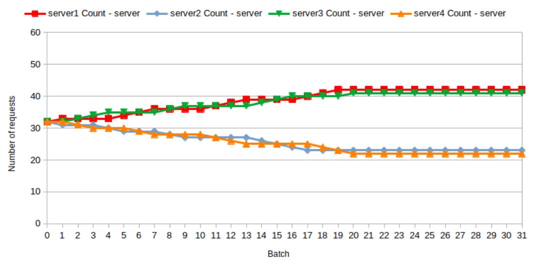
(a) 16 flows



(b) 32 flows



(c) 64 flows



(d) 128 flows

Figure 5.3: Load balancing behaviour based on the number of flows with 4 servers.

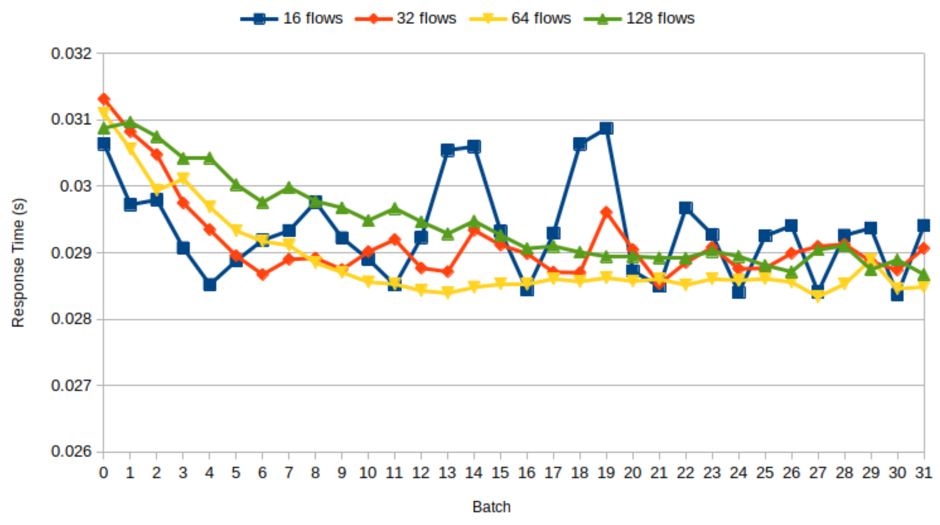


Figure 5.4: Average Response Time comparison between flows using 4 servers.

Figure 5.3 shows the output of the client script, here one can find the behaviour of the load balancing algorithm under load. For each chart, the **X** axis represents the progression of request batches, while the **Y** axis shows the number of requests distributed for each server. It is to note that for every batch of requests, the sum of the requests of all servers equals 128, which is the same number of requests that our client script sends in each batch.

As a reminder of section 5.2.1, server1 and server3 have the same performance in terms of response time by the number of requests. Server2 and server4 also have the same performance, but these have a higher response time than server1 and server3.

We can take some conclusions after analysing this charts. Server1 and server3 have the best performance, and therefore are attributed more flows, while server2 and server4 are attributed less flows. This happens to all the tests presented in this charts, however it is important to note that the first chart with 16 flows has a lot more instability compared to the other experiments.

As expected, with more flows introduced in our application, the more precision it has. In other words, the more flows we introduce in our system, the gap between flows becomes smaller. In one hand, the more flows the better, because we have more detail and the application can produce a more accurate load balancer. However, on the other hand, with each increment of flows, the heavier the application becomes, it consumes one more bit per register used, which is considered insignificant in our application since we are working with a small topology of servers, but can be significant in a mass scale topology.

The higher amount of flows also increases the time that our algorithm takes to balance the load until it reaches a stable solution. This is a significant aspect of our application, because we only change a maximum of one flow per batch of requests. There can be a work around this problem by introducing a more advanced flow distribution algorithm in the control plane.

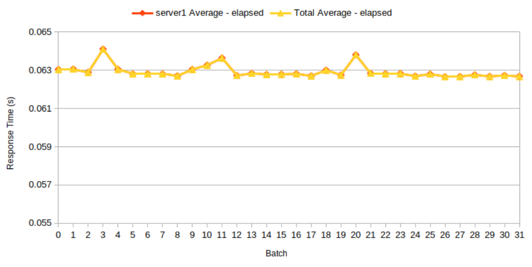
In fig. 5.4 we can consult the Average Response Time for each test shown in fig. 5.3. As expected we can take the same conclusions than by analysing the four charts individually. First, with less flows, the system has less stability. The tests with 64 and 128 flows have more precision than the tests with 16 and 32 flows. Secondly, with higher flows, the system takes longer to reach a stable value. Th test with 128 flows needs more batches of requests until it reaches the same value than the test with 64 flows, and the latter one also needs more requests to reach the same values of the tests with 16 and 32 flows.

Therefore we can conclude that, with 4 servers, the best outcome of our algorithm happens when we have a total of 64 flows. With 64 flows we have the best compromise between response time stability, and time necessary for the algorithm to find the best number of flows for each server.

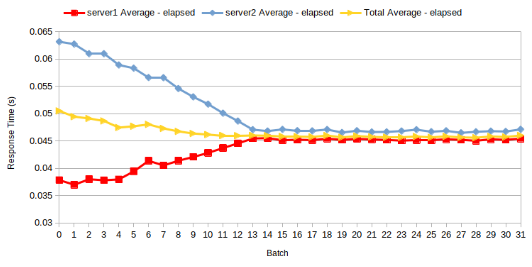
5.3.2 Performance Tests

To test the overall performance of our application we used our system with 64 flows and tried multiple topologies that have between one and four servers connected to our switch.

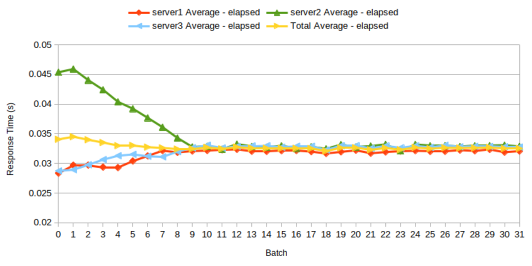
In fig. 5.5 we can consult how the response time evolves based on the number of servers connected to the load balancer. We can easily understand that when there is only one server connected, the load balancer doesn't work. Either way this is a relevant scenario in order to have a response time value of reference.



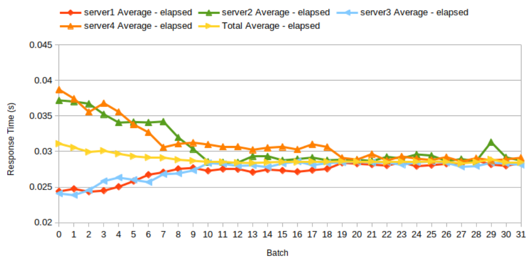
(a) 1 server



(b) 2 servers



(c) 3 servers



(d) 4 servers

Figure 5.5: Response Time evolution based on number of servers.

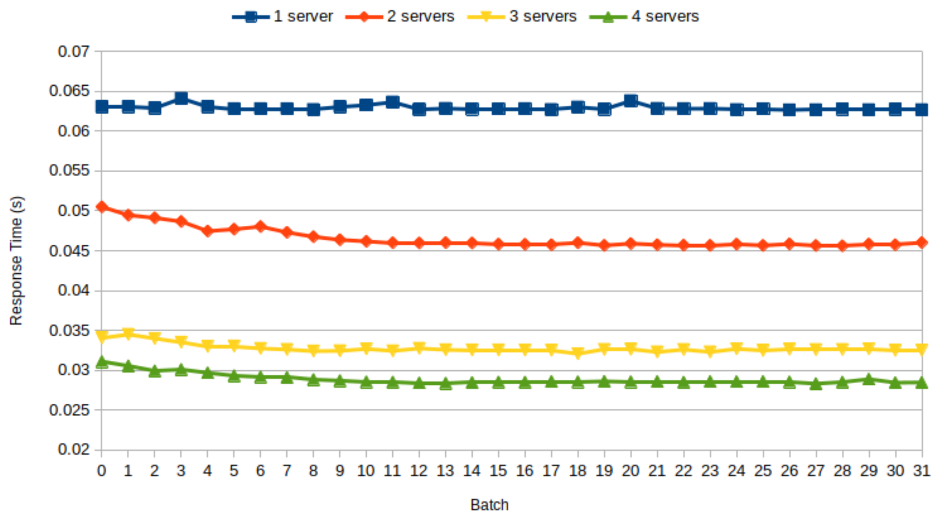


Figure 5.6: Average Response Time comparison between topologies with different number of servers.

When comparing the scenarios we understand that in fact the load balancing is working towards reducing the total average response time of the system. In all the cases were the load balancing takes effect we see a decrease in the average response time of the system, as well as a significant decrease in the maximum response time.

The expected outcome of a load balancing system is to reduce the average system response time when introducing new servers. We can verify that our application gives results in accordance with the expected in fig. 5.6. In this figure we compile the average response time of all the charts presented in fig. 5.5. We can also verify that, with the increase of servers in the topology, the response time gain decreases. Therefore, it may not be worth the cost to introduce 5 servers if the increase in performance is very slow.

It is also important to add that all of these tests were performed with 64 flows installed in our application because that was the scenario with best overall performance for a topology with four servers. We then used the application with 64 flows and tried topologies with different number of servers connected to have a uniform testbed for all topologies.

5.3.3 Fault Tolerance Test

Software Defined Networks have the ability to have a unified view of the network. Advanced controllers such as ONOS have APIs prepared with entities that listen for a wide span of events, including events like the addition or removal of hosts.

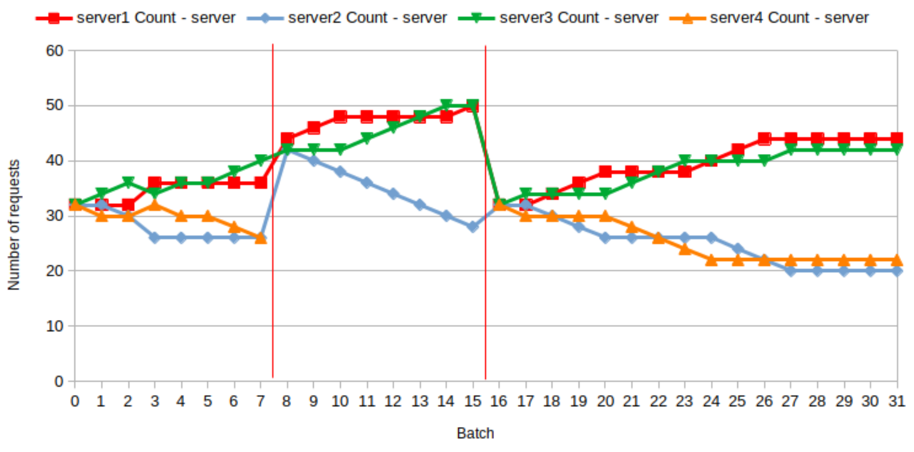
With these kinds of entities we can create triggers that react to these events and change the behaviour of our load balancing application in real time.

We decided to use a topology with four servers and a total of 64 flows to test this functionality. During our test we used the capabilities of the Mininet cli to turn down, and later back up, the connection between the load balancer switch and the server4. The results can be seen in fig. 5.7, where the first chart represents the evolution of the number of requests attributed to each server, and the second chart shows the average response time of the each server as well as the overall average response time of the system.

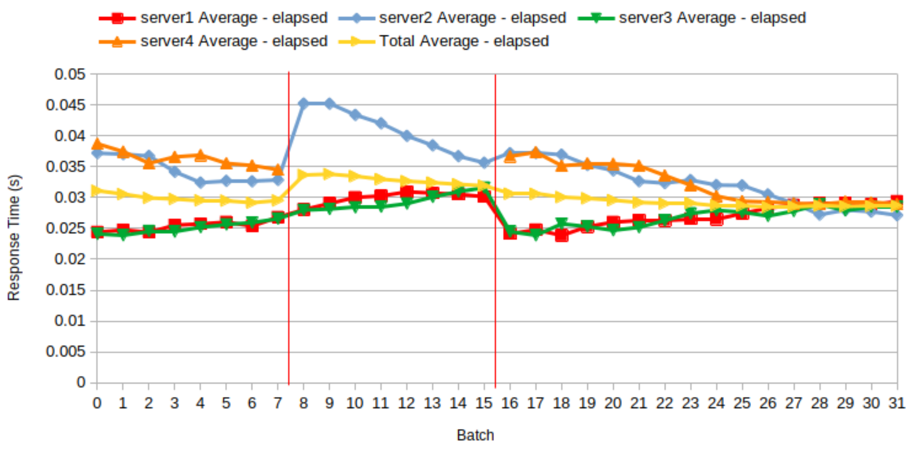
In this test we start with a topology consisting of four servers. As soon as the controller detects all servers, it distributes the 64 flows equally, meaning that every server is attributed 16 flows. In fig. 5.7(a) we can see that the servers start by receiving 32 flows each, that's because our test script sends 128 requests per batch, meaning that our algorithm loops two times and attributes 16 flows to each server twice.

The load balancer then starts the algorithm and begins to balance the load received until the batch 7, where the server4 crashes. At this point the controller detects the failure and reset the algorithm with three servers in mind, meaning that the previous load balancing evolution is forgotten and it start the algorithm from the start.

When the algorithm starts by the second time, it now only has three connected servers. It distributes the load again, but this time each server will be attributed with 21 flows (one server will randomly be attributed with 22 so the sum can give 64 flows). The algorithm then starts again and begins balancing the load with three servers online until the batch 15.



(a) Number of requests



(b) Response Time

Figure 5.7: Fault Tolerance Test.
Server4 fails between batches 7 and 16.

Server4 comes back online between batches 15 and 16. The controller detects again a change in the number of connected servers and decides to restart the algorithm. This time there will be no more interruptions and the program will balance the load until there it reaches a stable balance around batch 27, at which point the servers are considered to have requests with similar enough response times and the algorithm stops adjusting the flows.

5.3.4 Summary

The system was tested with different topologies and various kinds of flows.

We conclude that the number of total amount of flows can be adjusted considering the number of servers connected. In a topology with four servers, the best performance came with 64 flows, while with less servers a better performance may be achieved with less flows, as well as more servers may benefit from more flows.

Additionally, we reached the conclusion that adding more servers decreases the average response time, but the gap gets smaller with the addition of each server.

We also showed that the system has a near optimal behaviour upon the loss of one server, and that the adopted solution is fault tolerant.

6 Conclusion

Contents

In this chapter we present the conclusions about the project done in this thesis. Section 6.1 makes a brief summary of the paradigms addressed in this papers. Section 6.2 presents the achievements accomplished during the development of this project. Finally, section 6.3 references some relevant issues that may be addresses in future work.

6.1 Summary

SDN was developed to improve network configuration in traditional networks, it successfully decouples the control and data planes from the same network devices. The control plane was centralized in a controller that manages flow control by providing a global view of the network. There was a significant evolution in control plane technology, however, the data plane was left unchanged.

To address that issue and revolutionize the data plane, the P4 programming language was developed. It brings data plane programmability to switches by modifying how packet forwarding is processed. This can bring many benefits to SDN networking like custom defined network protocols and features.

This work aims to develop a load balancing application that benefits from the best features of both Data Plane Programmability, as well as Software Defined Networks.

6.2 Achievements

With our work we successfully improved upon the works mentioned in the Related Work, namely the article by Chih-Heng Ke et al. [5], by introducing a production-grade SDN controller, revamping the P4 code, and transforming a stateless algorithm, into a stateful algorithm.

Introducing ONOS in the application system bring a number of features like code modularity. This feature allows for the ONOS application to be higly configurable and interchangeable at runtime. With a generic load balancing scheme introduced in the data plane, the control plane can change the parameter of the Resource Based Load Balancer without the knowledge of the data plane. This is an improvement over the article by Ke since the work mentioned has the load balancing algorithms hard-coded in the data plane logic.

The project implemented in this thesis also supports an ambiguous number of servers trough configurability with NETCONF, while the related work has the servers explicitly in the data plane code, changing the topology by adding or removing a server is a heavy task.

Additionally, by introducing more control in the control plane, we were able to migrate a stateless system to a stateful one, introducing the advantages of the latter.

However, the focus of Data Plane Programmability is to increase the control of the network in the data plane. With our work we are splitting the control in between the data and control planes.

One of the conclusions of the paper by Ke was that by providing the load balancing algorithms in the data plane, their system was independent of the controller. Our is more dependent of the controller since it is constantly trying to balance the load between the servers based on their average response time. If the control plane fails in our solution, we lose the functionality of realtime analysis of the system, but the state of the load balancer when the control plane fails is saved in the data plane, meaning that the control plane could fail and our load balancer would still work, while being stuck in the same load balancing state.

If we consider that most of the time the system is in a stable state, that it only converges between servers in a small window of time, we can assumes that work solution remains working properly even on a control plane failure. Instead, if the system is unstable, and the control plane is constantly changing weights in between servers, then we could assume that our soultion is not perfect, in a scenario where the control plane fails.

6.3 Future Work

On one hand, we defend the SDN paradigm which states that the control should remain in the control plane, where ONOS with a unified vision of the network has a lot of visibility and can manage the data plane by installing flow rules that work as ”guide lines” for the data plane to work independently. With the help of ONOS modularity we propose as future work to improve the control plane application to have a broader amount of load balancing algorithms which the user could chose and switch between at runtime. Also, we propose to improve the situation where the algorithm takes a long time to converge when a high number of flows is introduced in the system. This implementation could extend the ONOS GUI component to include custom buttons and commands in the web application.

On the other hand, we would also like to explore the capabilities of P4 to the maximum, and propose to try and bring all the algorithmic logic in the control plane to the P4 code. This would be a hard task, the hardest part would be to maintain a generic code for any arbitrary number of connected servers.

Bibliography

[1] R. R. Schaller, “Moore’s law: past, present and future,” *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[2] N. M. K. Chowdhury and R. Boutaba, “A survey of network virtualization,” *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[3] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, pp. 14–76, 2015.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *Computer Communication Review*, vol. 44, pp. 87–95, 2014.

[5] C.-H. Ke and S.-J. Hsu, “Load balancing using p4 in software-defined networks,” *Journal of Internet Technology*, vol. 21, pp. 1671–1679, 2021.

[6] A. M. Joy, “Performance comparison between linux containers and virtual machines,” *Conference Proceeding - 2015 International Conference on Advances in Computer Engineering and Applications, ICACEA 2015*, pp. 342–346, 2015.

[7] “Docker hub - container image library,” accessed 18 December 2020. [Online]. Available: https://www.docker.com/products/docker-hub

[8] N. W. Paper, “Network functions virtualisation: An introduction, benefits, enablers, challenges call for action. issue 1,” Oct. 2012.

[9] F. Hu, Q. Hao, and K. Bao, “A survey on software-defined network and openflow: From concept to implementation,” *IEEE Communications Surveys and Tutorials*, vol. 16, pp. 2181–2206, 2014.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 69–74, 2008.

[11] N. McKeown and J. Rexford, “Clarifying the differences between p4 and openflow - open networking foundation,” accessed 21 December 2020. [Online]. Available: https://opennetworking.org/news-and-events/blog/clarifying-the-differences-between-p4-and-openflow/

[12] M. P. Fernandez, “Comparing openflow controller paradigms scalability: Reactive and proactive,” *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, pp. 1009–1016, 2013.

[13] “P4 brigade,” accessed 21 December 2020. [Online]. Available: https://wiki.onosproject.org/display/ONOS/P4+brigade

[14] “Opendaylight releases oxygen, with new p4 and container support,” accessed 21 December 2020. [Online]. Available: https://www.opendaylight.org/about/news/blogs/opendaylight-releases-oxygen-with-new-p4-and-container-support

[15] “Barefoot networks may have built the world’s fastest networking switch chip,” accessed 29 October 2021. [Online]. Available: https://www.computerworld.com/article/3083761/barefoot-networks-may-have-built-the-worlds-fastest-networking-switch-chip.html

[16] “Intel tofino2 next-gen programmable switch detailed,” accessed 29 October 2021. [Online]. Available: https://www.servethehome.com/intel-tofino2-next-gen-programmable-switch-detailed/

[17] P4.org, “The onf and p4.org complete combination to accelerate innovation in operator-led open source,” 2019, accessed 3 January 2021. [Online]. Available: https://p4.org/p4/p4-joins-onf.html

[18] “P4-16 language specification,” accessed 3 January 2021. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.1.html

[19] N. McKeown, T. Sloane, and J. Wanderer, “P4 runtime – putting the control plane in charge of the forwarding plane,” accessed 18 December 2020. [Online]. Available: https://p4.org/api/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane.html

[20] “P4runtime specification,” accessed 23 December 2020. [Online]. Available: https://p4.org/p4runtime/spec/v1.3.0/P4Runtime-Spec.html

[21] “P4-16 portable switch architecture (psa),” accessed 23 December 2020. [Online]. Available: https://p4.org/p4-spec/docs/PSA-v1.1.0.html

[22] G. Brebner, “Extending the range of p4 programmability,” 2018. [Online]. Available: https://p4.org/assets/P4WE_2018/Gordon_Brebner.pdf

[23] “p4lang/behavioral-model: The reference p4 software switch,” accessed 3 January 2021. [Online]. Available: https://github.com/p4lang/behavioral-model

[24] “Stratum,” accessed 3 January 2021. [Online]. Available: https://opennetworking.org/stratum/

[25] O. Shyshkov, “Load balancing - oleksii shyshkov’s blog,” 2018, accessed 18 October 2021. [Online]. Available: https://oshyshkov.com/2018/07/20/load-balancing/

[26] “Load balancing algorithms and techniques,” accessed 30 September 2021. [Online]. Available: https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/

[27] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, “Hula: Scalable load balancing using programmable data planes,” *Symposium on Software Defined Networking (SDN) Research, SOSR 2016*, 2016.

[28] E. Cidon, S. Choi, S. Katti, and N. McKeown, “Appswitch: Application-layer load balancing within a software switch,” *ACM International Conference Proceeding Series*, pp. 64–70, 2017.

[29] B. Valente, “bvalente/ngsdn-tutorial at loadbalancer.” [Online]. Available: https://github.com/bvalente/ngsdn-tutorial/tree/LoadBalancer

[30] “opennetworkinglab/ngsdn-tutorial: Hands-on tutorial to learn the building blocks of the next-gen sdn architecture,” accessed 5 October 2021. [Online]. Available: https://github.com/opennetworkinglab/ngsdn-tutorial

[31] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks.” Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1868447.1868466

[32] “stratum/tools/mininet at main · stratum/stratum,” 2021, accessed 20 October 2021. [Online]. Available: https://github.com/stratum/stratum/tree/main/tools/mininet

[33] “Onos : An overview,” accessed 5 October 2021. [Online]. Available: https://wiki.onosproject.org/display/ONOS/ONOS+%3A+An+Overview

[34] “Network configuration protocol (netconf),” accessed 20 October 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6241

[35] D. A. Menascé, “Load testing of web sites,” *IEEE internet computing*, vol. 6, no. 4, pp. 70–74, 2002.

[36] B. Boucheron, “An introduction to load testing — digitalocean,” 2017, accessed 7 October 2021. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-load-testing

A

Appendix

Listing A.1: Header Definitions in P4.

```
header ethernet_t {
    bit<48>  dst_addr;
    bit<48>  src_addr;
    bit<16>  ether_type;
}

header ipv4_t {
    bit<4>   version;
    bit<4>   ihl;
    bit<6>   dscp;
    bit<2>   ecn;
    bit<16>  total_len;
    bit<16>  identification;
    bit<3>   flags;
    bit<13>  frag_offset;
    bit<8>   ttl;
    bit<8>   protocol;
    bit<16>  hdr_checksum;
    bit<32>  src_addr;
    bit<32>  dst_addr;
}

header tcp_t {
    bit<16>  src_port;
    bit<16>  dst_port;
    bit<32>  seq_no;
    bit<32>  ack_no;
    bit<4>   data_offset;
    bit<3>   res;
    bit<3>   ecn;
    bit<6>   ctrl;
    bit<16>  window;
    bit<16>  checksum;
    bit<16>  urgent_ptr;
}

header udp_t {
    bit<16> src_port;
    bit<16> dst_port;
    bit<16> len;
    bit<16> checksum;
}

header icmp_t {
    bit<8>   type;
    bit<8>   icmp_code;
    bit<16>  checksum;
    bit<16>  identifier;
    bit<16>  sequence_number;
    bit<64>  timestamp;
}

header arp_t {
    bit<16>   hwType16;
    bit<16>   protoType;
    bit<8>    hwAddrLen;
    bit<8>    protoAddrLen;
    bit<16>   opcode;
    bit<48>   hwSrcAddr;
    bit<32>   protoSrcAddr;
    bit<48>   hwDstAddr;
    bit<32>   protoDstAddr;
}
```