Efficient Flow Routing From Programmable Data Plane

Undergraduate Degree

Nicole Venachi Pizo
Jorge Manuel Castillo Camargo

Advisor: PhD. Oscar Mauricio Caicedo Rendón
Co-Advisor: PhD. Cristhian Nicolás Figueroa Martínez

Department of Telematics
Faculty of Electronic and Telecommunications Engineering
Universidad del Cauca
Popayán, August 2022

Efficient Flow Routing From Programmable Data Plane

Nicole Venachi Pizo
Jorge Manuel Castillo Camargo

Undergraduate degree presented to the Faculty of Electronic and Telecommunications Engineering of the University of Cauca to obtain the degree of:
Electronics and Telecommunications Engineer

Advisor: PhD. Oscar Mauricio Caicedo Rendón
Co-Advisor: PhD. Cristhian Nicolás Figueroa Martínez

Department of Telematics
Faculty of Electronic and Telecommunications Engineering
Universidad del Cauca
Popayán, August 2022

Acknowledgements

Our families, especially Jorge Casitllo's father, our mothers, and siblings, for their unconditional support. Thanks to all the people who were part of our university life, friends, colleagues, and teachers of the Electronic Engineering and Telecommunications program who were essential support throughout our formative stage. Thanks to our Advisor, Oscar Mauricio Caicedo, and Co-Advisor, Cristhian Nicolas Figueroa, who became our academic and professional mentors. We would also like to thank the engineer Daniela Casas who gave us the theoretical guidance to perform the solution evaluation. Finally, we are proud to have performed our academic formation at the distinguished University of Cauca, where we managed to create bonds of friendship that will last throughout our lives and meet professionals with excellent training and human values, thus becoming the place where we achieve goals and dreams.

Abstract

The growing demand for data storage, transmission, and processing has become a challenge for network operators. Data centers suppose the core infrastructure to meet these needs, providing the environment to deploy rising network services and applications (e.g., big data, video streaming, and cloud computing). Multi-rooted topologies (e.g., Fat-Tree) rise as Data Centers' main architectures to mitigate the impact of the demanding workloads. These topologies allow massive multipathing to distribute the traffic workload. Moreover, Equal-Cost Multipathing (ECMP) is used as a de-facto load-balancing routing solution to split traffic through multiple paths evenly. Nevertheless, uniformly spreading the workload can degrade the network performance since the traffic is balanced without awareness of the network congestion, leading to bottleneck-specific links.

Novel P4-based congestion-aware and weighted load-balancing mechanisms such as HULA, CONGA, and DASH have successfully addressed this situation. Still, existing solutions introduce certain limitations. First, Conga is implemented in custom hardware. Second, HULA and CONGA easily bottleneck the best routing path since they only select and record the less congested path. Third, existing data-plane ECMP routing alternatives compromise the network performance since their congestion estimation is limited to links-state information, disregarding the switches' queue occupancy.

This Thesis introduces a P4-based load-balancing mechanism that overcomes previous limitations. In particular, the proposed mechanism computes multiple optimal paths to avoid the single best path quickly congestion. Moreover, this solution fits links and devices state information in the load-balancing algorithm to split arri-

ving flows based on weighted probabilities. Using the device-state information (i.e., queue occupancy) allows the mechanism to accurately characterize the network's congestion and consequently improves the routing performance.

This solution has been extensively evaluated in typical Fat-Tree data center topology using realistic workloads models from production data centers to generate traffic. Experimental results show that the proposed solution outperforms ECMP regarding the delay, throughput, and packet loss. Moreover, our solution reduces the packet loss up to 1.33x compared to alternatives using solely the link-state information to model the network congestion.

Resumen

La creciente demanda por almacenamiento, transmisión y procesamiento de datos se ha convertido en un reto para los operadores de redes. Los centros de datos surgen como la infraestructura central para satisfacer estas necesidades, proporcionando un entorno para desplegar servicios y aplicaciones de red en crecimiento (por ejemplo, big data, streaming de vídeo y computación en la nube). Las topologías multiraíz (por ejemplo, Fat-Tree) suponen una de las principales arquitecturas de los centros de datos para mitigar el impacto de las altas demandas en tráfico. Estas topologías permiten múltiples caminos masivos para distribuir la carga de trabajo del tráfico. Además, ECMP se ha posicionado como la solución de enrutamiento de equilibrio de carga por defecto para dividir el tráfico a través de múltiples caminos de manera uniforme.

Sin embargo, la distribución uniforme de la carga de trabajo puede degradar el rendimiento de la red, ya que el tráfico se equilibra sin tener en cuenta la congestión de la red, lo que lleva a cuellos de botella en enlaces particulares.

Los mecanismos de equilibrio de carga emergentes basados en P4, como HULA, CONGA y DASH, han abordado con éxito esta situación. Sin embargo, las soluciones existentes presentan ciertas limitaciones. En primer lugar, CONGA se implementa en hardware personalizado. En segundo lugar, HULA y CONGA congestionan fácilmente el camino de enrutamiento elegido, ya que sólo seleccionan y registran un único camino (aquel que presenta menor congestión). En tercer lugar, las alternativas existentes de enrutamiento en el plano de datos comprometen el rendimiento de la red, ya que estiman la congestión limitándose a parámetros del estado de los enlaces, sin tener en cuenta la ocupación de las colas de los conmutadores.

Esta tesis introduce un mecanismo de equilibrio de carga basado en P4 que supera las limitaciones anteriores. En particular, el mecanismo propuesto computa múltiples caminos óptimos para evitar cuellos de botella al emplear un único camino óptimo de enrutamiento. Además, esta solución incorpora la información del estado de los enlaces y dispositivos en el algoritmo de equilibrio de carga para dividir los flujos en la red utilizando probabilidades ponderadas. El uso de la información el estado de los dispositivos (como la ocupación de cola) permite al mecanismo caracterizar con precisión la congestión de la red y, en consecuencia, mejorar el rendimiento del enrutamiento.

Esta solución se evaluó ampliamente en una topología típica usada en los centros de datos, llamada Fat-Tree, además se modeló el tráfico en la red empleado de cargas de trabajo realistas, obtenidas de centros de datos de producción. Los resultados conseguidos muestran que la solución propuesta supera a ECMP en cuanto al delay, throughput y pérdida de paquetes. Además, nuestra solución reduce la pérdida de paquetes hasta 1,33 veces en comparación con las alternativas que utilizan únicamente la información del estado de los enlaces para modelar la congestión en la red.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
bmv2	behavioral model version 2
CPU	Central Processing Unit
DCN	Data Center Networks
D-ITG	Distributed Internet Traffic Generator
ECMP	Equal-Cost Multipathing
FCT	Flow Completion Time
gRPC	google remote procedure calls
IP	Internet Protocol
JSON	JavaScript Object Notation
P4	Programming Protocol-Independent Packet Processors
p4c	P4 compiler
p4-hlir	P4 High-level Intermediate Representation
PISA	Protocol-Independent Switch Architecture
PSA	Portable Switch Architecture
QoS	Quality of Service

RTT	Round Trip Time
SDN	Software Defined Networking
TCP	Transmission Control Protocol
WECMP	Weighted Equal-Cost Multipathing

Chapter 1

Introduction

Nowadays, the growing demand for cloud computing and Internet-based services has raised data centers as the core for providing a variety of cloud-based services such as Web-Hosting, Video Streaming, Social Networking, etc. [1, 6]. Data centers are facilities that house a wide number of computing and storage devices interconnected by a network infrastructure [1, 7].

This network infrastructure handles communication across tens to hundreds and even thousands of servers. Hence network architects have developed multi-rooted topologies (*e.g.,* Fat-Tree and Clos) to meet data centers' communication needs. Multi-rooted topologies provide a large bisection bandwidth by splitting the traffic through multiple paths [8, 9]. Therefore, data centers use load-balancing schemes to leverage the multi-path bandwidth. The main multipathing scheme for load-balancing in data center networks is ECMP. ECMP evenly splits traffic flows among all available next-hops using a hash function applied to the packet header to map packets into the available paths. Nonetheless, the flows forming the network traffic differ significantly, so evenly splitting the traffic results in non-optimal use of the bisection bandwidth since the routing devices can forward multiple large flows across the same path [10, 11]. Consequently, uniformly spreading the traffic compromises the network performance. Motivated by ECMP's shortcomings, network engineers have developed various research efforts from diverse perspectives.

Several research efforts address ECMP shortcomings from a centralized perspective. In, [12–19] use a centralized controller to load balance the Data Center Networks (DCN) traffic, splitting packets through optimal paths computed using the network traffic congestion or the traffic demands, instead of randomly distribute the load. Hence, previous alternatives improve ECMP performance by exploiting the links' status information to use the multipath bisection bandwidth efficiently. Nevertheless, centralized mechanisms do not fit bursty and unpredictable DCN traffic as they require a high convergence time to update the congestion information [20]. Furthermore, they increase communication overhead, consequently deteriorating the communication throughput. Moreover, new-arriving flows demand higher Central Processing Unit (CPU) usage, overloading the controller process facilities as the inter-arrival flow ratio increases [21]. Indeed, high inter-arrival rate scenarios, such as *Internet of Things* networks, or *Distributed Denial-of-Service* attacks, introduce a large controller load, compromising the scalability, availability, Quality of Service (QoS) in end-to-end communications, and the security of the entire infrastructure [22–24]. Additionally, most Software Defined Networking (SDN) centralized schemes rely on OpenFlow [25] protocol. Nevertheless, OpenFlow-based networks match packets regarding a set of fixed protocols (*e.g.,* Transmission Control Protocol (TCP), Ethernet, and IPv4). Therefore, OpenFlow-based centralized routing solutions are unsuitable for the novel routing techniques arising in DCN as it does not offer the flexibility to add new and customized protocol headers [26].

Emerging data plane-based routing solutions [27, 28] address the SDN centralized schemes issues leveraging distributed implementations. Moreover, distributed solutions based on Programming Protocol-Independent Packet Processors (P4) [29–32] surpass the OpenFlow inflexibility to adopt arising packet headers efficiently. Nevertheless, existing distributed solutions adopt link-state metrics as parameters to choose the best routing path, ignoring the forwarding devices' status. However, device-state parameters such as buffer occupancy or queue length are important indicators since they reveal the switches' capability to deal with arriving flows. Disregarding this metric leads to overflowing the flow entries, causing packet loss and traffic congestion.

In summary, to address load-balance routing mechanisms in SDN, researchers ha-

ve focused on a centralized perspective. This perspective can degrade the overall network performance and security. Moreover, a centralized perspective does not suit traffic volatility in DCN and is constrained to a set of fixed header protocols. Furthermore, emerging SDN distributed load-balancing routing solutions compromise the network performance as they disregard the forwarding devices' status, causing packet loss and deteriorating the switches' capability to deal with arriving flows. Therefore, this undergraduate proposal was guided by the following research question:

How to provide an efficient routing mechanism from a programmable data plane?

1.1. Objectives

1.1.1. General Objective

- Propose a routing mechanism based on data plane programmability, link-state, and device status.

1.1.2. Specific Objectives

- Design a routing mechanism based on data plane programmability, link-state, and device status.
- Implement the proposed routing mechanism using P4 to achieve in-band routing.
- Evaluate the effectiveness of the mechanism regarding delay, packet loss, and throughput.

1.2. Contributions

This undergraduate work's main contributions are mentioned below, which were obtained from the previously mentioned objectives 1, 2, and 3.

- A routing mechanism based on data plane programmability, link state, and device state.
- An implementation of the proposed mechanism using P4, as well as an extensive evaluation of its performance regarding the delay, packet loss, and throughput.
- A comparative evaluation among solutions modeling the network congestion using solely link-state or both link-state and device-state information to establish the device-state impact on the routing performance.
- To our knowledge, our mechanism is the first data-plane load-balance solution splitting traffic through optimal paths computed using both link-state and device-state information to estimate network congestion.

1.3. Outline

This undergraduate research document is composed of five chapters which are described below.

- Chapter 1 presents the **Introduction** that includes the problem statement, objectives, contributions, and this document outline.
- Chapter 2 presents the **State Of the Art**, consisting of two subsections. First, a **Backgorund** on main topics related to this undergraduate research (including Network Programmability, Data Center Networks, SDN, P4). Second, an overview of the **Related Works** presents previous works within the study area of this undergraduate research and identifies gaps around the research problem.
- Chapter 3 introduces the **Probe-based weighted load-balancing** mechanism, presenting the **Probe's Header Formatting**, the **Probe-based congestion information collection** and the **Weighted path selection** algorithm.
- Chapter 4 introduce the proposed load-balancing mechanism **Test Environment** and the corresponding **Results Analysis**.
- Chapter 5 presents the research **Conclusions** highlighting the main contributions. Furthermore, it also presents relevant recommendations for the development of **Future Work**.

Chapter 2

State Of Art

This chapter illustrates background on the main topics related to this thesis. First, we introduce the *Data Centers Networks*, its common network topologies, and routing mechanisms. Second, we explain the *Network Programmability*, its evolution, and present. Third, we review *Software Defined Networking* and its development regarding the network programmability. Finally, we introduce P4 language, its main features, architecture, and ecosystem.

2.1. Background

2.1.1. Data Centers Networks

A *Data Center* is a set of physical facilities working together to store, share and process large amounts of data. The DCN suppose the communication infrastructure used in a data center. This infrastructure includes different routing devices connected following specific network topology to communicate the entire infrastructure using a variety of network protocols [1].

In DCN, the network topologies typically consist of either a two- or three-tier tree of routing devices (*e.g.,* switches and routers). Figure 2.1 shows the *Fat-tree* network

topology, whose design is a hierarchical three-tier layer system: *core*, *aggregation*, and *edge*. The k parameter defines the number of *pods*: a two-layer subsystem composed of edge and aggregation switches. Each core switch connects the different k pods to provide communication in the network infrastructure. Hence, the DCN topologies supply bisection bandwidth using many equal-cost paths between each host pair. Therefore, they use load-balancing routing schemes to leverage the multi-path bandwidth [8, 9].

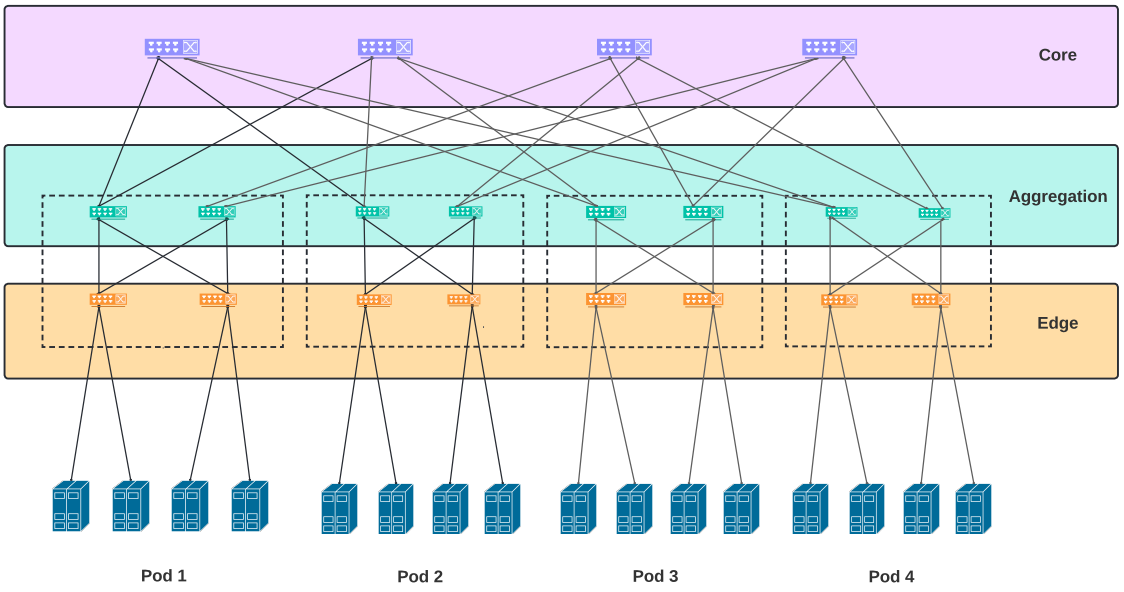

Figure 2.1: Fat-tree topology, K = 4 - adapted from [1]

The best-known multipath routing technique is ECMP [33], a hash-based flow forwarding technique that performs load balancing by equally distributing the traffic along several paths. Practical experience shows that dealing with high network loads ECMP operates well enough in mice flows (i.e., small flows) traffic scenarios numbers. Nevertheless, working with large flows (i.e., elephant flows) leads ECMP to traffic imbalances, and consequently link congestion [34].

2.1.2. Network Programmability

Network programmability is a set of tools and practices to deploy, manage and troubleshoot network devices. The network programmability allows network engineers to specify and control the network behavior by defining customized algorithms in the control or data plane. Hence, network engineers deploy networks regarding the user needs [35].

Traditionally, the hardware vendors' specifications have constrained the development of the control and data plane algorithms since providers did not allow customized execution environments running directly in the routing devices' hardware, preventing hardware damage and security issues.

Leveraging the resource isolation provided by the *virtualization*, SDN rises as the first wide-scale solution for network programmability accepted by academics and the industry. Nowadays, SDN supports network programmability from centralized and distributed perspectives.

The centralized network programmability perspective employs a centralized programmable control plane, which communicates with the data plane through well-known protocols, such as OpenFlow [**?**] and ForCES [**?**]. Hence, network engineers run custom-made algorithms in the control plane through an external agent. Next, the control plane communicates the match-action rules to the data plane devices using the above protocols[35].

The distributed network programmability perspective runs customized algorithms directly in the different data plane devices. Thus, network operators can modify the switches' packet processing behavior using the data plane programmability without incurring an external API. Data plane programming languages adapt to specific models to express algorithms abstractly. The PISA model is the programmable data plane model for P4 language [3], which will be discussed further in the following sections.

2.1.3. Software Defined Networking

SDN is an emerging paradigm that undoes vertical integration by separating network intelligence (control plane) from routing devices (data plane). The traditional SDN architecture consolidates a centralized controller to define the network operation. In this centralized paradigm, the programmable control plane communicates the routing devices' policies through a southbound Application Programming Interface (API).

The traditional SDN centralized architecture widely-extended southbound is well-known protocol OpenFlow [36]. OpenFlow switches support different routing tables to handle incoming packets. Thus, each arriving flow matching a predefined rule follows the same routing policies [2, 37].

The SDN centralized network architecture offers the following advantages in the routing process:

- The centralized controller provides a global view of the network, enhancing the routing path selection process.
- The decoupling of intelligence from the routing devices enables network programming and consequently adapts the behavior of the traditional routing protocols or even offers new routing alternatives.

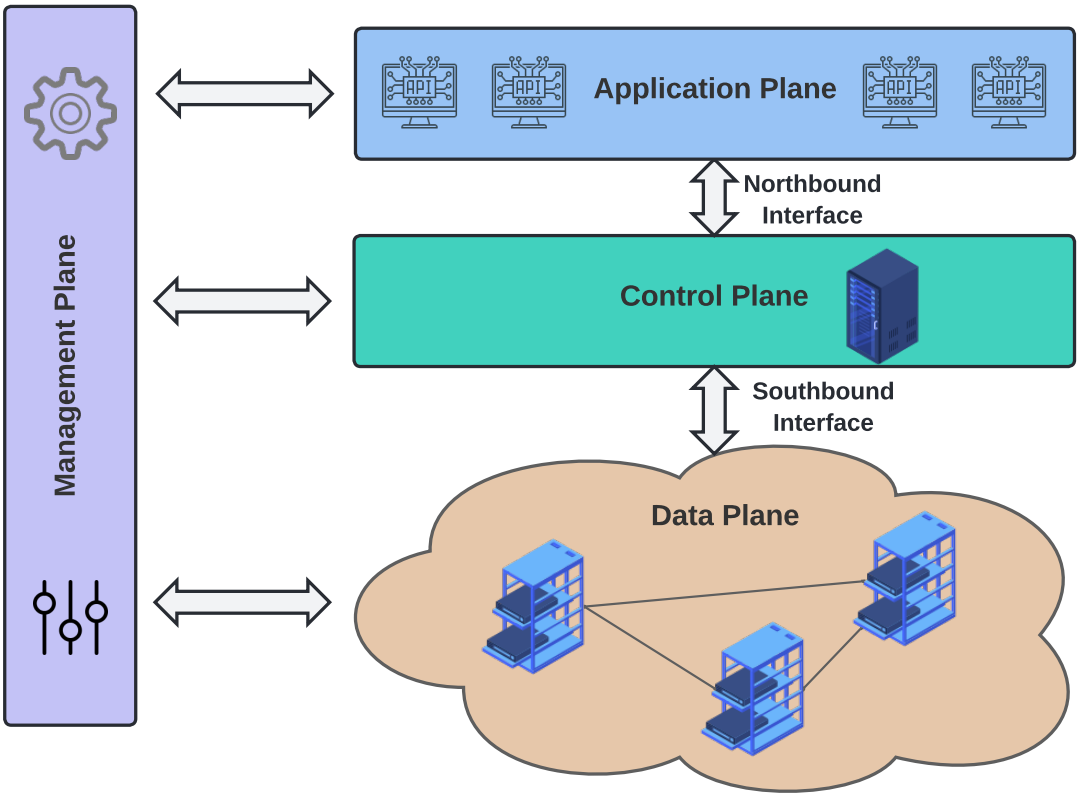

Figure 2.2: SDN Architecture - adapted from [2]

Figure 2.2 shows the SDN centralized network architecture, which is constituted as follows:

- Data plane: It is composed of different interconnected forwarding devices.
- Control Plane: It manages the network logic. In particular, it sets the network policies communicated to the forwarding devices.
- Application plane: It is a set of services communicating their desired network behavior and requirements to the controller.
- Management plane: It is a set of applications coordinating each plane individually.

- Northern Border Interface: Defines an API that communicates between the control and application planes. It typically abstracts the southern border interface's instructions to program routing devices.
- Southern Border Interface: The southern border API defines the communication protocol between the data and control planes. This protocol gives access to the forwarding devices from the centralized controller.

2.1.4. P4

P4 is an open-source language that provides network programmability directly on the data plane forwarding devices. P4 allows network engineers to specify how the forwarding devices process incoming packets. Although P4 does not interfere in the control plane operation, it defines a general communication interface between the control and data plane [38, 39]. The P4 main features are:

- Flexibility: This allows reconfiguring the switch behavior through a P4 program previously installed on it.
- Protocol independence: Unlike OpenFlow, P4 switches are not tied to any packet formats. Instead, they allow custom-made packet encapsulation protocols through a parsing packets mechanism to extract specific header fields and a set of match-action tables to process these custom headers.
- Target independence: P4 programs do not require any underlying hardware. They only need a device capable of translating the P4 programs.

PISA arises as the base architecture model for P4 data plane network applications (2.3).

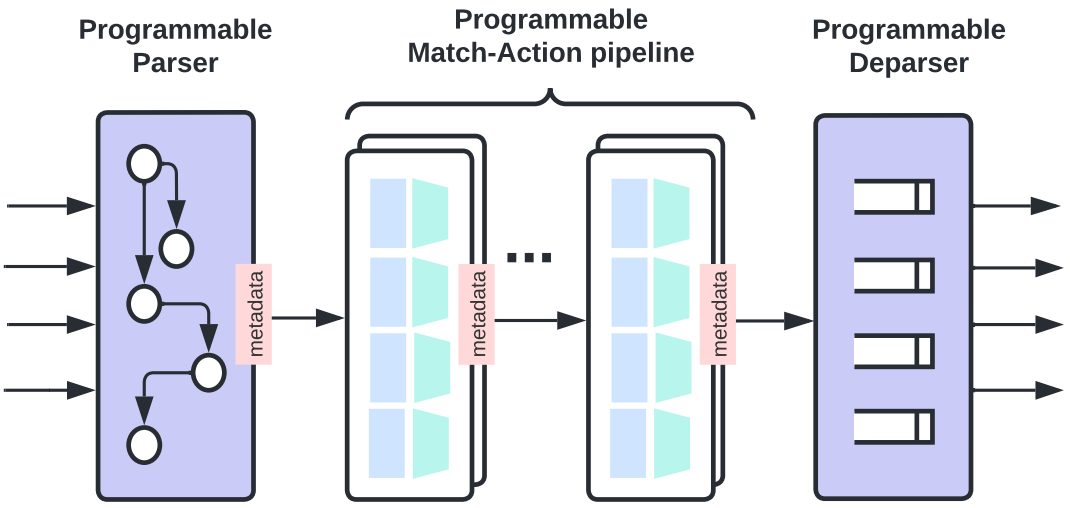

Figure 2.3: Protocol-Independent Switch Architecure (PISA) - adapted from [3]

PISA is a single pipeline forwarding architecture that offers custom programmable packet parsing, packet processing using match-action pipelines, and general purpose registers for stateful operations [3]. Thereby, PISA monitors and controls switch queuing directly in the data plane. Additionally, PISA switches allow different simultaneous queries, reducing the amount of data sent to the stream processor.

Figure 2.3 shows the three main components of the architecture. It includes a programmable parser, a deparser, and a match-action pipeline. The parser describes the network's header sequences allowed, the process of identifying these header sequences, and the process of extracting the header fields from incoming packets. The parser and deparser are reconfigurable to support user-defined packet header formats. The deparser determines how packets are serialized. On the other hand, the input and output pipelines process packages through match-action tables organized by stages. The match-action tables perform the specific actions on the packets matching the headers fields using predefined matching rules [3].

Incoming packets consist of both the payload and the packet metadata. Nevertheless, PISA focuses on processing the packet metadata. PISA splits the packet metadata into:

- Packet headers.
- Intrinsic metadata.
- User-defined metadata.

PISA use these metadata fields to incorporate into the header different helpful information for processing packets throughout the parser until reaching the deparser [3].

On the other hand, P4 comprises two standards which are: $P4_{14}$ and $P4_{16}$.

$P4_{14}$ allowed network programmers to write data plane algorithms using a set of 70 reserved words. However, different limitations appeared over time, caused partly by imperative programming constructs. Moreover, $P4_{14}$ weakly supported programs' modularity. Hence, $P4_{16}$ was introduced to overcome these limitations, so $P4_{14}$ development and support were interrupted.

$P4_{16}$ was established as the leading standard. Various architectures, such as *V1model* and Portable Switch Architecture (PSA) support this standard. A P4 architecture is a programming model that allows the implementation of the logic and processing capabilities of a P4 switch [38].

V1model is one of the most extended $P4_{16}$ architectures used in P4 switches. Figure 2.4 shows the V1model. This architecture model is similar to PISA as it consists of a programmable parser/deparser, and an input/output match-action pipeline. Still, it aggregates a traffic manager responsible for packet buffering, queuing, and scheduling. Moreover, V1model allows the developers to migrate $P4_{14}$ programs into $P4_{16}$ [3].

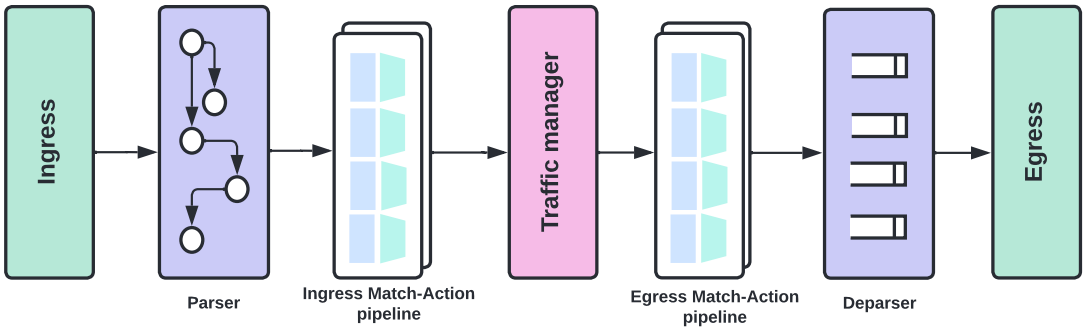

Figure 2.4: V1model Architecture - adapted from [3]

Software-based P4 targets are packet routing programs that run on a standard CPU. The two most popular are:

- P4c-behavioural.
- behavioral model version 2 (bmv2).

P4c-behavioral was the first version of a P4 target. It consisted of a P4 compiler and a P4 switch. P4c-behavioral translates $P4_{14}$ programs into a C program. It also relies on a *python* module called P4 High-level Intermediate Representation (p4-hlir).

bmv2 was introduced to support $P4_{16}$ programs and to resolve the limitations of P4c-behavioral switches. Eventually bmv2 switches replace P4c-behavioral switches entirely. To run bmv2 programs it is required a P4 file (containing the P4 program) and a JavaScript Object Notation (JSON) file to compile the P4. The P4 compiler (p4c) is written in C++, and it is the responsible for generate the JSON file [40]. From this model, there have been deployed the following P4 switches:

- simple_switch: Contains all the features of $P4_{14}$ and also supports the *v1model* architecture of $P4_{16}$ using an API that improves the exchange of data plane information. Hence, it uses the Thrift API to communicate the forwarding devices and the control plane at the runtime.

- simple_switch_grpc: It has the same characteristics as simple_switch. Nevertheless, it implements the P4RunTime API.
- psa_switch: It is similar to simple_switch, but it supports the PSA architecture instead of v1model
- simple_router and L2_switch: They do not support $P4_{16}$. They are used to experience different architecture implementations in bmv2.

The P4 APIs ease the runtime management of the P4 devices. Also, they enhance the exchange of information among the data and control planes. One of the most common APIs in the data plane is P4Runtime which was designed for the $P4_{16}$ version. Figure 2.5 shows how P4Runtime works. The *p4runtime.proto* file specifies the operating API, it also describes how the controller accesses the P4 entities using the *P4Info* metadata. The P4 devices have a google remote procedure calls (gRPC) server and controllers have a gRPC client. The interaction between the controller and p4 devices is as follows. First, a P4 compiler generates the *P4Runtime* configuration consisting of the P4 device configuration and its P4Info metadata. Second, the controller uses this information to establish a gRPC connection with the P4 device to access the entities and configuration of the P4 device [3, 4].

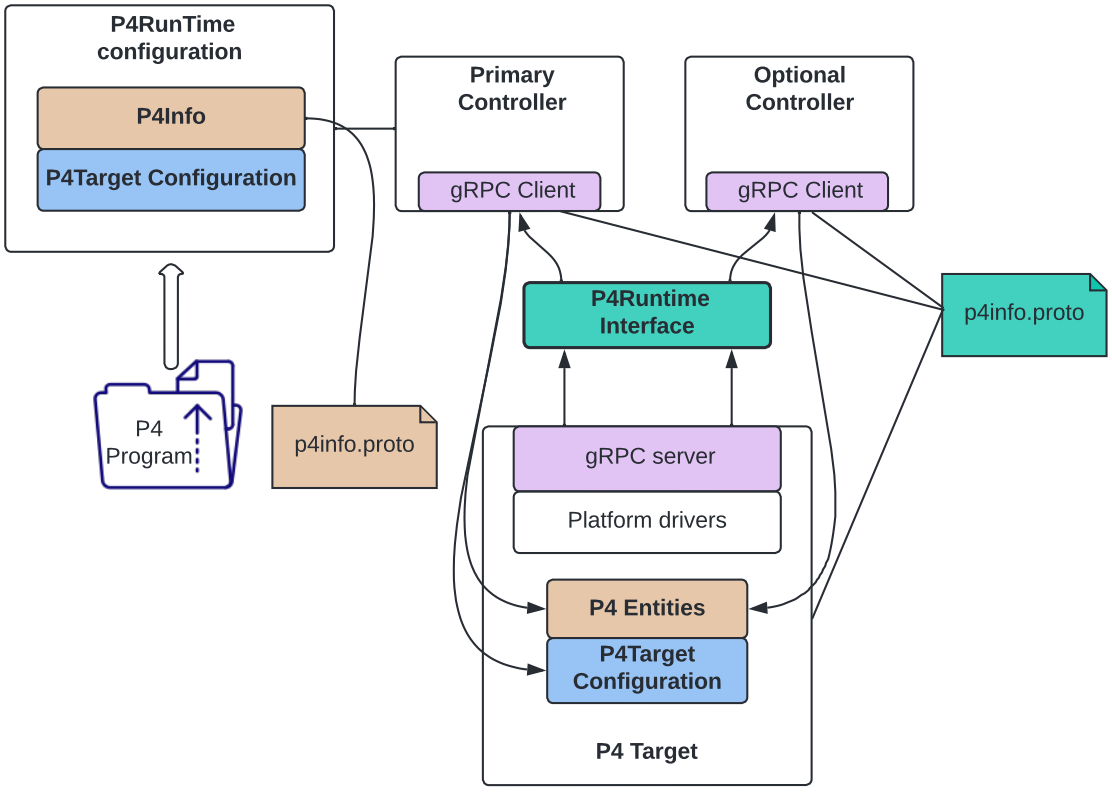

Figure 2.5: P4Runtime architecture - adapted from [4]

2.2. Related work

This section reviews research works in the context of SDN-based ECMP solutions. Typically routing mechanisms developed in SDN networks implement centralized logic techniques to manage flows through networks. Still, rising SDN routing solutions focus on a distributed architecture to overcome centralized scheme shortcomings (e.g., controller overhead, security issues, etc.) and OpenFlow limitations.

2.2.1. Control Plane-based routing

[12] presents HEDERA, a dynamic flow scheduling system to overcome the multi-pathing protocol's non-efficient use of network resources. HEDERA collects incoming

flow utilization information from all constituent switches and computes multiple optimal routing paths to maximize the bisection bandwidth utilization. Thus, HEDERA dynamically schedules network flows, moving them from highly-utilized links to the less utilized links (*i.e.,* optimal paths). The scheduler mechanism allocates flows by combining the path congestion information and the flow's natural bandwidth demands, achieving non-conflicting paths for each flow. HEDERA defines a TCP flows natural bandwidth demands as the mean rate it would grow to in a fully non-blocking network. Thus, HEDERA reallocates the flows using its scheduling mechanism whenever a flow persists over time and its bandwidth demands grow beyond a defined limit. Moreover, HEDERA's design aims to mitigate the impact of active flows on the network, minimizing the overhead scheduler information. HEDERA implementation fits on the data center's multi-stage switch topologies. Experimental results show that Hedera affords 4 x more bisection bandwidth than ECMP's static load-balancing methods.

[13] introduces B4, a private wide area network connecting multiple Google's data centers worldwide. B4 is built upon an SDN centralized architecture to control traffic splitting through multiple paths regarding the demands of the applications. B4 supplies centralized traffic engineering services to allocate the network bandwidth across diverse competing applications. B4's services use an objective function to compute the application bandwidth given the flow's priority. The relative application priority arises from administrator-specified static weights. B4 leverages the available network capacity load balancing the traffic along multiple paths according to application priority. Non-SDN standard wide area networks are limited to work at moderate utilization (*e.g.,* constrained at 30-40 % utilization for the busiest links) to avoid packet loss and reserve backup capacity in case of link failures. Still, the operational experience over the first three years of deployment shows that B4 enables cost-effective bandwidth management, running most links at near 100 % utilization over extended periods (*i.e.,* achieving 2-3 x efficiency improvements compared to most wide area networks).

[14] proposes SWAN, a highly efficient and flexible system to enhance the links utilization in inter-data-center wide area networks. SWAN controls the traffic each service sends based on the current service demands and the network utilization.

Next, SWAN updates the switches' tables to carry traffic according to these demands. Maintaining high utilization involves frequent data plane updates to meet current traffic demands consistently. Consequently, SWAN periodically re-configures the switches' tables to match the traffic demands. However, it leverages a small amount of the link's capacity to handle the updates without compromising latency-sensitive traffic performance due to the transient congestion caused by the switches' non-synchronized updates. Also, SWAN scales to large networks since it faces limited forwarding table capacities by greedily selecting a small set of entries that can optimally match the current demands. The data-driven simulations of two production networks show that SWAN can improve the utilization by around 60 % compared to prevailing methods for inter-data-center wide area networks.

[15] presents CAMOR, a routing strategy that combines multipath routing and a congestion-aware mechanism to obtain the optimal path for every new flow between source and destination. CAMOR controller computes the optimal path regarding the status of each participating link, using a congestion-aware mechanism that collects the load of each available link. Moreover, CAMOR uses multipath routing to distribute the traffic on multiple optimal routes, preventing congestion due to routing on a single best path. Experimental results show that CAMOR traffic splitting favors routing paths with higher available throughput, improving the overall network performance.

[16] introduces a global multipath load-balanced routing algorithm based on machine learning. This algorithm leverages on SDN controller global view to collect enough information on network traffic state (*i.e.*, links and bandwidth utilization) from all possible flows within the network to generate a global routing policy using reinforcement learning. Then, a load balancing mechanism uses this policy to find the best routing paths for each pair of source-destination nodes on the network. The algorithm performance presents superior delay and network utilization results compared to the shortest path and round-robin algorithms.

[17] proposes a load-balancing technique using the transport layer protocol multipath TCP [41] coupled with Internet Protocol (IP) aliasing [42] as a solution to the links under-utilization issues from the load-balancing process. This technique reduces

the maximum links load by splitting connections between different sub-flows and multiple streams through diverse TCP sessions. IP aliasing enables the formation of various sub-flows depending on the IP availability addresses and redirects traffic in the sub-flows across all available paths. The proposed techniques implement the multi-path TCP specifications to address the challenges of load-balancing different sub-flows. Moreover, this technique enhances network utilization using Hamiltonian paths to forward each subflow among the network. This technique improves the congestion control in the available paths, the under-utilization of network links, and the network's overall throughput compared to regular TCP.

[18] presents a multipath routing mechanism to meet both multiobjective QoS demands and provide efficient path optimization. This mechanism consists of a two-phase procedure to achieve both requirements. The first phase uses an analytic hierarchy process to capture the varying QoS requirements and reduce them to a new cost function to designate different link weights. Hence, reducing the traffic processing complexity and improving the network's quality control. The second phase uses a two-step algorithm to select the optimal routing path. At first, the controller obtains a set of candidate paths between source and destination using a customized version of the Dijkstra algorithm. Then, the controller decides the optimal path among the set of candidate paths using a greedy algorithm regarding the QoS cost function computed in the first phase. Thereby, this two-phase procedure improves the load balancing of the network core by dynamic path planning subject to multiple QoS requirements. It is noteworthy that the mechanism authors did not present any experimental evaluation of the mechanism in this proposal.

[19] introduces an SDN flexible and dynamic load balancer to improve network performance reducing the response time. The load balancer computes the shortest alternative routes for each incoming flow and dynamically balances traffic between different paths calculated using the Dijkstra algorithm. Moreover, the load balancer design provides network flexibility since its a non-customed solution. Hence, the load balancer operation is not constrained to any network topology. The network flexibility is tested by conducting performance evaluations in small and large SDN networks. Experimental results show that the proposed load balancer improves the network's transfer rate and response time.

2.2.2. Data Plane-based routing

[27] presents a data-plane multipath routing alternative that relies on Flowlet [43] to manage the network flows division and monitoring. The solution exploits Flowlet bursts elastic data size to distribute the traffic adequately according to link conditions, achieving resilient load balancing. In particular, the endpoint switches dynamically monitor latency to find the minimum interval that mitigates the packet reordering. Hence, using the monitoring information, the endpoint switches alternate bursts of packet flows within multiple available paths, avoiding packet reordering and reducing link congestion. Additionally, this solution enhances the *"Flowlet timeout"* estimation by applying in-band telemetry and monitoring to measure the latency dynamically, enabling proper flow splitting. The overall traffic evaluation shows that this alternative achieves similar Flow Completion Time (FCT) for different workloads than naive LetItFlow.

[28] proposes CONGA, a DCN congestion-aware enterprise solution to efficiently balance actual TCP workload into Flowlets without requiring any TCP modifications. CONGA uses a congestion feedback mechanism based on periodical probes carrying the link utilization of every path between leaf switches. Each probe updates hop-by-hop path utilization as it travels through the entire path. Hence, each probe information is stored at the destination leaf switch and fed back to the source leaf switch in a reverse direction. Then, leaf switches use the per path congestion information to compute the best routing path for each arriving flow. Leaf switches select the optimal path on the first packet of each Flowlet. As long as the Flowlet remains active, packets use the same recorded path. Testbed experiments show that CONGA outperforms 5 x the ECMP delay. Also, CONGA is more suitable for the DCN bursty traffic conditions compared to centralized schemas (e.g., Hedera) as it improves several times the congestion information update frequency. Moreover, CONGA is an enterprise solution that is permanently tested in real traffic conditions.

[29] introduces HULA, a DCN congestion-aware load-balancing algorithm that aims to cover the switch's memory limitation. HULA uses unique probes to gather global link utilization information. These probes travel periodically throughout the network and cover all desired paths for load balancing. Then, link utilization information is

summarized and stored as a table at each switch, providing an optimal next-hop to any destination. Each switch updates the HULA probe with its view of the best next-hop to any destination and sends it to other upstream switches. Then, each HULA switch tracks congestion using the downstream information to disseminate the best path in the entire network, similar to a distance vector protocol. Compared to CONGA, HULA fits switches' memory limitations since switches only store the best next-hop information to track the best path instead of maintaining a per-path congestion state at the leaf switches. Besides, the authors designed HULA to run on data-plane programmable chipsets without requiring custom hardware, programming it on P4 thinking on emerging programmable switches. Experimental results show that HULA outperforms ECMP and CONGA regarding the FCT for different workloads.

[30] presents a Weighted ECMP routing alternative for DCN, implemented directly in the data plane using P4. Like the well-known DCN solutions, CONGA and HULA, this alternative collects link congestion information to solve the non-aware traffic balance in DCN. Nonetheless, in HULA and CONGA, each switch only selects and records the best routing path dynamically, easing this path quickly congestion. Unlike them, the proposed solution uses weighted probabilities to choose a path among different optimal routing solutions to overcome the single best path issues. Moreover, the proposed routing mechanism encapsulates path utilization data directly into the regular traffic, reducing the bandwidth utilization and convergence time (*i.e.,* increase the path weights update frequency). Hence, improving the FCT overall performance compared to HULA at different probe frequencies.

[31] proposes DASH, a hash-based load-balancing mechanism implemented in P4. DASH entirely balances the load on multiple paths in the data plane to react on a small timescale to traffic dynamics. DASH's adaptive weighted routing strategy splits traffic at line rate as a service of the current path load (calculated using link utilization and delay) using periodical probes to dynamically adjusts the load changes at data-plane speeds. Unlike HULA and Conga, DASH overcomes data-plane dynamic adaptative multipath implementations challenges, routing across different optimal paths instead of a single best path recording. Compared to ECMP and HULA, DASH presents a desirable FCT value in symmetric and asymmetric topologies

achieving less time in user interaction with the network. It also reaches less convergence time, generating faster routing decision-making as it takes fewer packets to adjust new path weights.

[32] introduces CONTRA, a general, programmable, and novel system to perform performance-aware routing at hardware speeds. Unlike other data-plane routing solutions (*e.g.,* CONGA, HULA, etc.), CONTRA is a general solution that fits any network topology and a broad number of routing policies (e.g., shortest path, minimum utilization, congestion-aware routing, weighted links, etc.). Thereby, CONTRA supplies a configurable environment where the network administrator customizes the network operation (*i.e.,* topology and policy). Once the network is customized, CONTRA sets up different distributed local P4 programs to run in the programmable network switches. In particular, CONTRA's routing strategy is a probe-based specialized distance-vector protocol to forward traffic based on routing constraints. This protocol uses periodical compact probes to collect performance metrics across diverse network paths. Switches analyze the performance metrics to compute the best policy-compliant next-hop to any destination. Experimental results show that CONTRA's is competitive with hand-crafted systems (*i.e.,* HULA) regarding FCT in DCN.

2.2.3. Gaps

Table 2.1 summarizes the works discussed through the related works section to address efficient load balance in SDN. This table briefly describes each mechanism and highlights its research gaps.

Control plane-based solutions [12–19] use a centralized controller to generate the load balancing decisions. In centralized controller mechanisms, incoming flows that do not match any switch flow entry produce extra communication among the switch and the controller to define the rules matching the received flow. However, introducing extra communication between the forwarding and the control plane increases the control information overhead, consequently deteriorating the communication throughput and QoS in end-to-end communications. Furthermore, new-arriving flows

increase the controller CPU usage overloading the controller process facilities as the inter-arrival flow ratio increases ([21]). As a result, high inter-arrival rate scenarios, such as *Internet of Things* networks or *Distributed Denial-of-Service* attacks, introduce a large controller load due to continuous communication messages between planes, compromising the scalability, availability, QoS in end-to-end communications, and the security of the entire SDN infrastructure [22–24]. Moreover, centralized mechanisms respond too slowly to the traffic volatility in DCN as they require a high convergence time to update the congestion information [20].

CATEGORY	WORK	MECHANISM	SHORTCOMING
Control Plane Perspective	[12]	Uses a dynamic flow scheduling system to overcome the multipathing protocol's non-efficient use of network resources	They decrease the overall network performance and security
	[13]	Uses a SDN centralized architecture to control traffic splitting through multiple paths regarding the demands of the applications.	
	[14]	Uses a highly efficient and flexible system to enhance the links utilization in inter-data-center wide area networks	They demand higher CPU usage
	[15]	Uses multipath routing and congestion-aware mechanisms to compute the optimal path for new flows between source and destination	
	[16]	Uses a global multipath load-balanced routing algorithm based on *Machine Learning* to find the best routing paths	They do not fit bursty and unpredictable DCN traffic
	[17]	Uses a load-balancing technique using the transport layer protocol multi-path TCP coupled with IP aliasing	They do not offer the flexibility to add new and customized protocol headers
	[18]	Uses a multipath routing two-phase mechanism to meet both multiobjective QoS demands and provide efficient path optimization	
	[19]	Uses a dynamic load balancer based on Dijsktra algorithm to improve network performance reducing the reponse time	
Data Plane Perspective	[27]	Uses a multipath routing alternative that relies on Flowlet to manage the network flows division and monitoring	Higher packet loss and traffic congestion since they dismiss devices status metrics.
	[28]	Uses a DCN congestion-aware enterprise solution to efficiently balance real TCP workload into Flowlets without requiring any TCP modifications	
	[29]	Uses unique probes to gather global link utilization information to implement acDCN-aware routing	
	[30]	Uses a Weighted ECMP routing alternative for DCN implemented directly in the data plane using P4	
	[31]	Uses a a hash-based routing mechanism to dynamically balance traffic regarding paths load, improving the FCT and convergence time of different Weighted ECMP algorithms.	
	[32]	Uses a novel programmable system to perform performance-aware routing at hardware speeds	

Table 2.1: Comparison of routing methods in software-defined networks

Additionally, previous works show another gap that makes them unsuitable for novel load balance techniques emerging in production data-centers networks. In particular, most SDN centralized schemes rely on OpenFlow [25] as the standard communication protocol between SDN controllers and the data plane. The network programmability

using OpenFlow allows network operators to program the flow forwarding given a set of fixed protocol headers on which OpenFlow operates (*e.g.,* TCP, Ethernet, and IPv4). Nonetheless, data-centers network operators increasingly intend to apply new forms of packet encapsulation (e.g., NVGRE [44], VXLAN [45], and STT [46]) fitting DCN rising challenges. Hence, network operators resort to deploying software switches that are easier to extend with new functionalities rather than repeatedly await until OpenFlow extends its specification to support new header fields [26].

Emerging data plane-based routing solutions [27, 28] address the SDN centralized schemes issues leveraging distributed implementations. These solutions handle the traffic variations in-band, applying network programmability directly in the Data Plane, and deploying low-memory algorithms in the forwarding devices. Thus, they release the controller's load by critically reducing the communication messages among the data and control planes since the controller adopts passive routing tasks. Moreover, P4 language supports a programmable packet parser for extracting header fields with particular names and types, allowing the network operator to define new and customized headers. Consequently, distributed solutions based on P4 [29–32] surpass the OpenFlow inflexibility to efficiently adopt arising packet headers. Nevertheless, existing distributed solutions adopt link-state metrics as parameters to choose the best routing path, ignoring the forwarding devices' status. However, device-state parameters such as buffer occupancy or queue length are important indicators since they reveal the switches' capability to deal with arriving flows. Disregarding this metric leads to overflowing the flow entries, causing packet loss and traffic congestion. Thus, routing alternatives must consider link-state and device-state metrics to identify the network's real state and perform accurate routing decisions.

The previous solutions externalize a research gap located at the intersection between the SDN distributed routing mechanisms and the typical routing metrics they exploit. This thesis meets this gap by proposing an innovator approach to carry out a data-plane routing mechanism based on links-state and devices-state.

Chapter 3

Routing from the Programmable Data plane

In this undergraduate work, we propose a probe-based mechanism for weighted load balancing. The mechanism uses links' and devices' state information to estimate the overall network congestion. We leverage the data-plane programmable features offered by P4 to implement the proposed mechanism. So, in this chapter, we present the mechanism's functional operation. It is noteworthy that the entire mechanism proposed in this section is our original work, except the *paths mapping strategy to binary numbers* shown in Fig. 3.4 adapted from [30].

Figure 3.1 introduce a high-level view of the proposed mechanism. At its core, this mechanism consists of the following subsystems working together to achieve a weighted load-balancing scheme:

- A probe-based congestion information collection subsystem: This subsystem relies on Algorithm 1 presented in section 3.2 to estimate the congestion of each existing path among source and destination using the links and devices' state information. Also, this subsystem progressively computes each path's weight as the probes travel through the paths. It is noteworthy that this subsysmtem runs until it probes each path among source and destination. Still, it restarts

periodically to update the network congestion systematically. Moreover, it is run by core, aggegation and edge switches.

- A weighted path selection subsystem: This subsystem relies on on Algorithm 2 presented in section 3.3 to compute multiple optimal paths for arriving flows. It sets a probabilistic interval of choosing each path using the path weights computed in the forehead subsystem. It is noteworthy that this subsysmtem runs once all paths have been already probed (i.e., once the previous subsystem has finished). Moreover, it is run entirely on the edge switches.

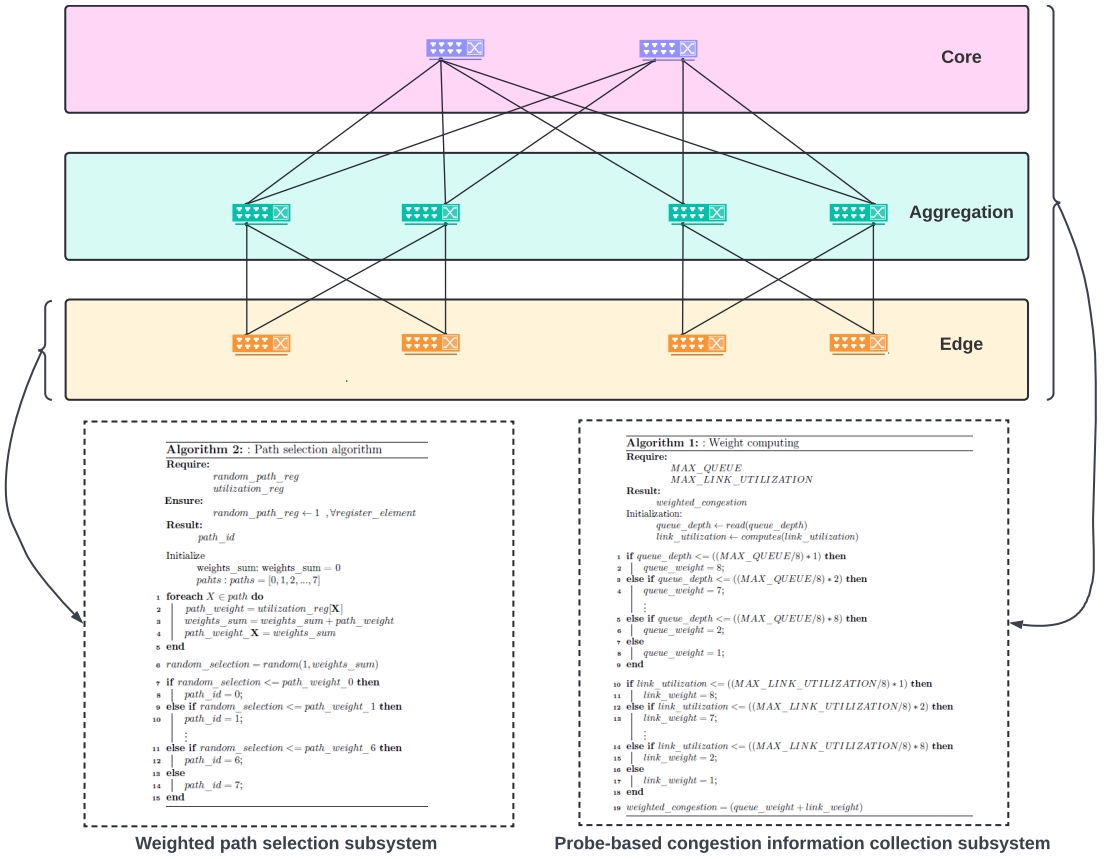

Figure 3.1: High-level view of the weighted load-balancing scheme - original work

The following sections extensively discuss each subsystem design.

3.1. Probe's Header Formatting

A P4 program design begins with the specification of header formats. Hence, before discussing the mechanism design, we need to set out the customized header format used in the mechanism operation. Compared to centralized SDN protocols (*e.g., OpenFlow*), P4 supports customized header fields to specify how to process packets instead of rules matching existing header fields [26]. Hence, Fig. 3.2 shows our customized probe's header formatting. In this subsection, we explain in detail each header field usage:

- *pod_dir* (8bits): This field is set in the edge switches to identify the source communication pod in the core switches.
- *selected_path_id* (8bits): This field is set in the edge switches using the congestion path information to indicate the selected path to arriving flows. Also, both core and aggregation switches use this field to compute the next-hop port.
- *output_tag* (8bits): This field is update in each switch once the next-hop is computed. Its value (*i.e.,* 0 or 1) indicates the upstream link followed at the switch. It is used to track the path taken.
- *tag_pat_id* (8bits): This field is built hop-by-hop in each switch among the selected routing path to communicate this path in the destination edge switch. Besides, this field, combined with *output_tag* allows tracking the path taken.
- *edge_flag* (8bits): This field is set in the destination edge switch as 1. It is used to distinguish returning probes at the source edge switch. Futhermore, it helps to characterize probed paths.
- *weighted_congestion* (32bits): This field is used to compute the weight of the routing path taken by the probe, using the hop-by-hop congestion information of each switch across the path. Moreover, edge switches compute the path for arriving flows using the *max_utilization* of each path collected by the probes traveling through the network.

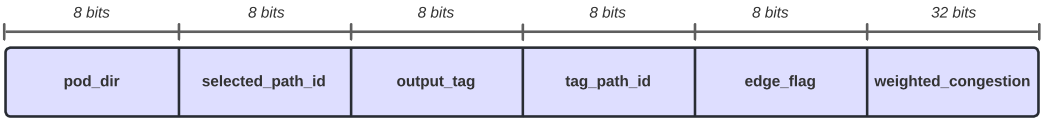

Figure 3.2: Probe's Header Formatting - original work

3.2. Probe-based congestion information collection

In this section, we propose a probe-based congestion information collection subsystem. We addressed this subsystem's functioning from two complementary points of view, one regarding the edge switches and the other regarding aggregation and core switches. Fig. 3.3 introduces a small-scale model of the implemented topology. This model is used to explain the mechanism in the following sections. It is noteworthy that the model rearranges the topology switches providing a different perspective of the same topology to ease the mechanism explanation. Despite using a scaled model, the following explication applies between each pair of source-destination edge switches.

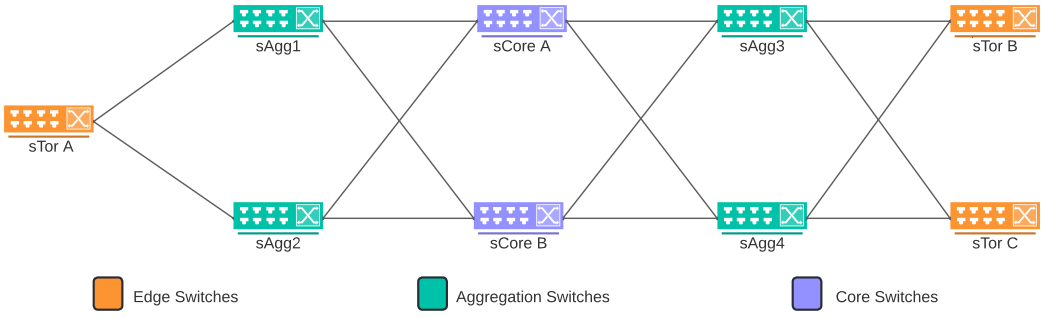

Figure 3.3: Small-scale topology model - original work

3.2.1. Probe-based congestion information collection - Edge switches view

The congestion information collection starts in the edge switches. Essentially, edge switches duties are to send probes along the different paths connecting source and destination. Hence, these switches must orchestrate the following tasks:

- First, they randomly select a routing path between source and destination to probe its congestion.
- Second, they select the probe's next hop according to the routing path selected.
- Third, they forward back the probe through the same path at the destination pod.
- Fourth, once a path is probed, they repeat the previous steps to probe the congestion through all available paths. Therefore, edge switches require a mechanism to identify which paths have already been probed.
- Finally, once all paths have been probed, the edge switches will know the congestion of the entire network. Nevertheless, edge switches must re-update the network congestion due to the changing traffic conditions. Hence, edge switches will repeat the previous steps according to an established *probe's frecuency.*

The routing path selection is a simple process. Edge switches only need to randomly select a path among the eight multiple paths connecting the source edge switch and destination aggregation switch. This path is stored as a binary number in the probe's header field *selected_path_id*. Fig. 3.4 illustrates the strategy used to map a path into a binary number. First, let's assume that the path chosen is the red one. Second, since every switch is connected to two different upstream links, we can differentiate each link by a binary number (*i.e.,* 0 and 1). Hence, links going upward are mapped as 0, and links going downward are mapped as 1. Third, the previous process is wielded in every switch among the source edge switch and the last aggregation switch

in the destination pod. It is noteworthy that the link between the aggregation and edge switches in the source pod is not mapped using the previous steps since it can be easily discovered using the destination address (in this example, the destination switch will be the *sTor B*). Furthermore, mapping this link as a binary number would increase the number of entries maintained in the edge switches to store the path's congestion. Finally, each bit is concatenated from right to left to produce the path's binary number. This number is stored in the *selected_path_id* field (see Fig. 3.5).

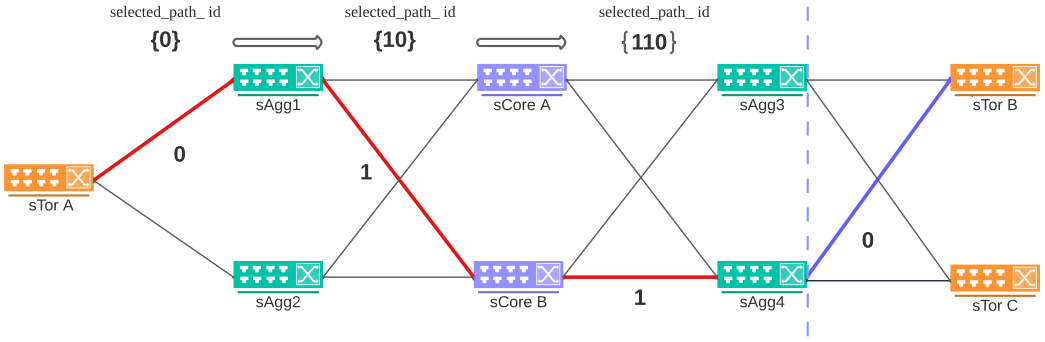

Figure 3.4: Paths mapping strategy to binary numbers - original work

Next, the edge switch computes the next hop among the two possible upstream links, using the previously selected path to send the probe through this path. Besides, the switch sets the probe's *output_tag* field (see Fig. 3.5) according to the upstream link selected (*i.e.,* 0 or 1). The next-hop computation process is explained in detail in the section 3.2.2 since it is the same process handled by the aggregation and core switches.

Fig. 3.5 depicts an overview of the probe process of any path. This process begins by recording the selected path and the output port at the source edge switch, as described in the previous steps. Then, each aggregation and core switch computes the congestion into weights and records its value at the probe *weighted_congestion* field. They also update the *tag_path_id* field to track the selected path. Finally, the destination edge switch also updates the *weighted_congestion* field with its congestion, sets the *edge_flag* field into 1, and forward back the probe to the source edge switch using the same path.

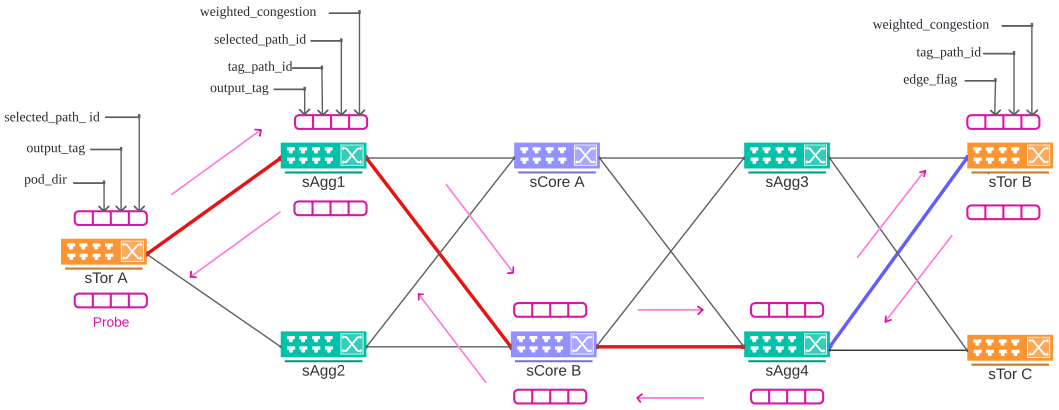

Figure 3.5: Probe process overview - original work

Then, the source edge switches must repeat the previous processes to send collection probes along all available paths. Consequently, edge switches use a mechanism to identify probed paths. Hence, as the probes packets return to the source edges switches, they hash the fields *tag_path_id* and the *edge_flag* to record probed paths. We used an eight-bit register called *random_path_reg* to store this information for the different routing paths. Also, we create an 32-bit register called *utilization_reg* to store the path weights hashing the incoming probe's fields *edge_flag* and *weighted_congestion*.

Finally, once the network congestion is probed, the mechanism will route arriving flows regarding the computed weights (see section 3.3). However, as time pass, network congestion conditions will change. Thus, the mechanism must probe the path's congestion anew to recompute the weights. Therefore, the edge switches must periodically repeat the probe process explained in this section. This job is effectuated using timers to restart the register *random_path_reg*, forcing the edge switches to probe the network congestion again. We provide the mechanism with a called *probe's frecuency* used to establish the time interval to recalculate weights periodically. Moreover, the network operator can customize *probe's frecuency* according to the traffic conditions.

3.2.2. Probe-based congestion information collection - Aggregation and Core switches view

Once the probes have been sent from the edge switches, we need to define the switching mechanism to transport the probes across the set path. This section describes the core and aggregation switches mechanism to handle receiving probes.

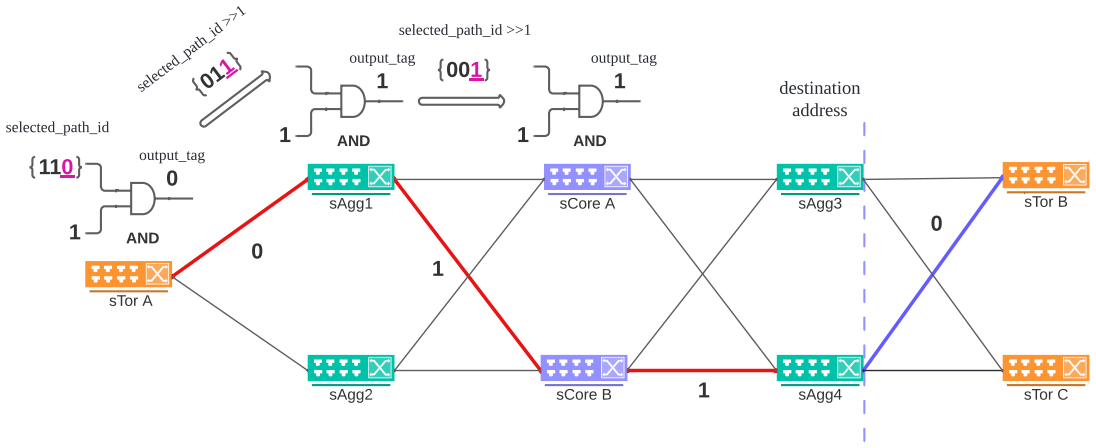

Figure 3.6: Switching mechanism - original work

First, each switch along the selected path needs to find the next-hop link regarding this path. Hence, the switches read the *selected_path_id* field from the receiving probe to compute the next hop. As shown in Fig 3.6, each switch operates the selected path binary representation using the boolean AND operator. Thus, switches operate the *selected_path_id* AND the binary number 1 to isolate the last bit of the selected path. This bit represents the next hop in each switch since the *selected_path_id* was built from right to left. Thereby, we leverage the path's binary mapping strategy used (see Fig. 3.4) to find the next hop along every switch in the path as each *selected_path_id* bit represents the two possible upstream links in every switch.

Second, each switch must update the *selected_path_id* field since the following upstream switches use the last bit from this field to find its next-hop link. Consequently, each switch operates the *selected_path_id* value by an arithmetic 1-bit right shift.

The result from this operation is updated in the *selected_path_id* probe's field before switch it to the next-hop (see Fig. 3.5).

The combination of the two previous steps allows switches to track the next hop along the path using the last bit. As shown in Fig. 3.4, each switch finds the probe's next hop through this mechanism. Nevertheless, the last aggregation switch from the destination pod computes the next-hop link using the edge switch destination address. Moreover, each switch can distinguish the *upstream* switches using the *edge_flag* field and the destination address.

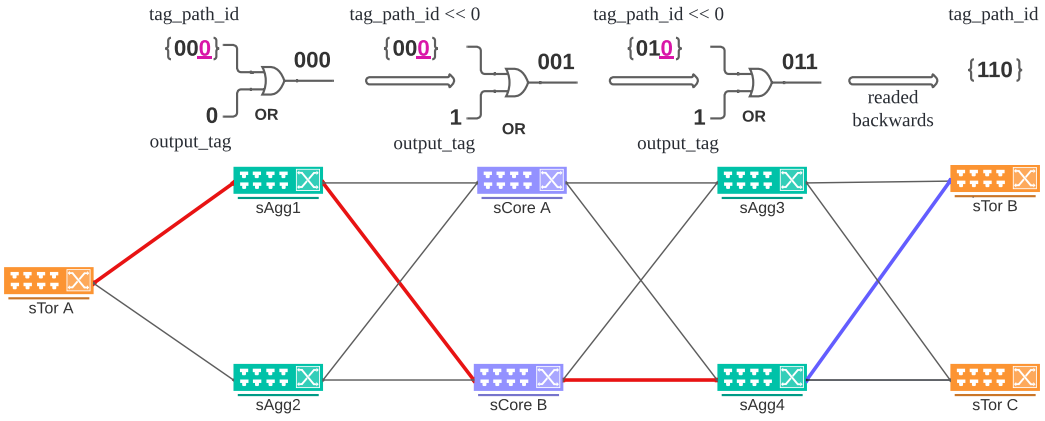

Figure 3.7: Path tracking mechanism - original work

Third, switches need to track the selected path to communicate it in the destination edge switch, forward back the probe, and record probed paths at the source edge switch. Nevertheless, switches rewrite the *selected_path_id* value hop-by-hop to find the next hop. We provide core and aggregation switches with a system to track the selected path. This mechanism is performed hop-by-hop overwriting the probe's *tag_path_id* and *output_tag* fields. Fig. 3.7 depicts this mechanism. Firstly *output_tag* fields tracks the upstream link taken in the previous switch (*i.e.,* the next hop found in each switch). Then, each switch operates the *output_tag* with the the *tag_path_id* field by using the boolean OR operator. This operation allows to store this *output_tag* value in the *tag_path_id* last bit. Next, the switch operates the *tag_path_id* value by an arithmetic 1-bit left shift to record from right to left the *output_tag* taken by each switch. Later, the switch rewrite into the probe the

updated *tag_path_id* value and the new *output_tag* computed corresponding to its next-hop direction. The switch sends this probe to the upstream next-hop switch, which repeats the previous process to rebuild hop-by-hop the path taken. Finally, the destination edge switch reads backward the probe's *tag_path_id* and stores it in the *selected_path_id* field. Thus, the destination switch can forward the probes to the source edge switch.

Algorithm 1: : Weight computing

Require:
MAX_QUEUE
$MAX\_LINK\_UTILIZATION$

Result:
$weighted\_congestion$

Initialization:
$queue\_depth \leftarrow read(queue\_depth)$
$link\_utilization \leftarrow computes(link\_utilization)$

```
1  if queue_depth <= ((MAX_QUEUE/8) * 1) then
2  |   queue_weight = 8;
3  else if queue_depth <= ((MAX_QUEUE/8) * 2) then
4  |   queue_weight = 7;
   |   :
5  else if queue_depth <= ((MAX_QUEUE/8) * 8) then
6  |   queue_weight = 2;
7  else
8  |   queue_weight = 1;
9  end

10 if link_utilization <= ((MAX_LINK_UTILIZATION/8) * 1) then
11 |   link_weight = 8;
12 else if link_utilization <= ((MAX_LINK_UTILIZATION/8) * 2) then
13 |   link_weight = 7;
   |   :
14 else if link_utilization <= ((MAX_LINK_UTILIZATION/8) * 8) then
15 |   link_weight = 2;
16 else
17 |   link_weight = 1;
18 end
19 weighted_congestion += (queue_weight + link_weight)
```

Fourth, as each switch receives the probe, it will update the *weighted_congestion* field. Algorithm 1 illustrates the weight computing process (regarding the link utilization and the switch's queue). First, the algorithm initializes two different variables

called *queue_depth* and *link_utilization*, containing respectively the switch reads of the queue depth and the link utilization when the packet was enqueued. To read this queue and link utilization information, we leverage the switches' intrinsic metadata offered by P4 switches. Then, the edge switch assigns a weight from 8 to 1 for these queue readings (lines 1 to 9). We use a descending strategy to map the queue reading to the weight. That is to say, if there were no queued packets, the switch would map this reading as a weight of eight. On the contrary, if the queue were full when the probe was processed, the switch would map this reading as a weight of one. Notably, this descending strategy assigns higher weights to optimal readings. We repeat this normalization previous process to map the link utilization reading up to eight different weights (lines 10 to 18). Finally, the switch updates the probe's *weighted_congestion* field adding to its current value the sum of the link utilization and queue weights (line 19). This entire process is wielded in every core and aggregation switch until the destination edge source. Moreover, to characterize the last link-device utilization, the destination edge source also calculates its weights following this algorithm. The algorithm inputs are integer numbers containing the maximum link utilization (*i.e., MAX_LINK_UTILIZATION*) and the maximum queue length (*i.e., MAX_QUEUE*), and the algorithm output is the updated weighted congestion across the chosen path until the device running the algorithm (*i.e., weighted_congestion*).

It is to noteworthy that the *swicthing* (Fig. 3.6), *path tracking* (Fig. 3.7), and weight calculation mechanisms runs parallel hop-by-hop along switches found at the selected path (Algorithm 1). Moreover, core switches use the *pod_dir* field to distinguish upstream switches in the communications arriving from the different pods in the evaluated topology (see 4.1). Besides, the switches write their weights *weighted_congestion* just in the going path from the source to the destination. Therefore, they use the *edge_flag* field to avoid rewriting the path weights once the probe is forwarded back to the source.

3.3. Weighted path selection

Our proposed load-balancing mechanism operations consist of two different subsystems. The section 3.2 introduced the first subsystem, the probe-based congestion information collection. This section will introduce the second subsystem, the weighted path selection.

Algorithm 2: : Path selection algorithm

Require:
random_path_reg
utilization_reg

Ensure:
random_path_reg $\leftarrow 1$, $\forall$*register_element*

Result:
path_id

Initialize
weights_sum: weights_sum = 0
pahts : *paths* = $[0, 1, 2, ..., 7]$

```
1  foreach X ∈ path do
2  |  path_weight = utilization_reg[X]
3  |  weights_sum = weights_sum + path_weight
4  |  path_weight_X = weights_sum
5  end
6  random_selection = random(1, weights_sum)
7  if random_selection <= path_weight_0 then
8  |  path_id = 0;
9  else if random_selection <= path_weight_1 then
10 |  path_id = 1;
   |  :
11 else if random_selection <= path_weight_6 then
12 |  path_id = 6;
13 else
14 |  path_id = 7;
15 end
```

The Algorithm 2 presents the weighted path selection process. This process relies on a probabilistic load balancing method to compute multiple optimal paths for arriving flows. This algorithm runs entirely in the edge switches. The algorithm inputs are the registers containing the probed paths (*i.e., random_path_reg*) and the paths weights (*i.e., utilization_reg*). Moreover, it must ensure that every path has already

been probed (*i.e.,* this algorithm starts once each *random_path_reg* element have been set as 1). The output from the algorithm is the chosen path (*i.e., path_id*). Also, the algorithm initializes two different variables, one called *weights_sum* is initialized as 0, and the other called *paths* is initialized as a vector whose elements represent each path. After these previous considerations, the weighted path selection process begins (line 1). First, performing the following steps, the edge switches sets an interval of choosing for each path according to its weight (lines 1 to 5):

- The edge switch reads the *utilization_reg* weight from the current path and stores it in the temporal variable *path_weight* (line 2).
- The edge switch adds to the *weights_sum* variable the current path weight (lines 3 to 4).
- The edge switch creates for each path a variable named *path_weight_X*, where X represents the current path (line 4). Hence, this variable will store the accumulated path weights until the X path. Thus, this variable will represent the interval of choosing the X path.

Once the previous *for-each* loop has finished, the variables *weights_sum* and *path_weight_X*, will respectively represent the sum of all paths weights and the probabilistic interval of choice for each path. Then, the proposed algorithm performs the following steps to select a routing path for each arriving flow (lines 6 to 14):

- The edge switch generates a random number among 0 and *weights_sum* (i.e the sum of all paths weights). They store this value in a variable named *random_selection* (line 6).
- The edge switch compares the *random_selection* value with each *path_weight_X* variable, where X represents the possible paths (each *if-elseif-else* condition from lines 7 to 14). Hence, if the *random_selection* value fits on the interval of two consecutive paths, the edge switches will execute the statements from the selected *if*, *elseif* or *else* condition.

- The edge switch choose the routing path corresponding to the *path_ weight_ X* selected variable, where X represents the possible paths. The *path_ id* variable stores this value.

The edge switch will route the flow through this selected path using the previous algorithm. This routing process is the same followed by the probing packets, explained in section 3.2.2.

	path_0	path_1	path_2	path_3	path_4	path_5	path_6	path_7
weight	64	24	24	50	27	8	0	30

Table 3.1: Sample path weights

	path_ weight_0	path_ weight_1	path_ weight_2	path_ weight_3	path_ weight_4	path_ weight_5	path_ weight_6	path_ weight_7
value	64	88	112	162	189	197	197	227
interval_ path_weight	64	24	24	50	27	8	0	30
probability	$\frac{64}{227}$	$\frac{24}{227}$	$\frac{24}{227}$	$\frac{50}{227}$	$\frac{27}{227}$	$\frac{8}{227}$	$\frac{0}{227}$	$\frac{30}{227}$

Table 3.2: Accumulated path weights and probability of choicing for each path

To illustrate the algorithm functioning, let's assume that each path presents the Table 3.1 weights. As shown in Table 3.2, the edge switches will use these paths weights to create eight different variable named *path_ weight_ X* (where X represents the possible paths) to record the accumulated path weights until the X path. Hence, the edge switches *random_ selection* value will be compared to these *path_ weight_ X* to select the routing path. The Table's *interval_ path_ weight* row represents the interval of choice for each path. Thus, we can calculate the probability of choosing for each path, as shown in the Table's *probability* row.

Futhermore, by analyzing Table 3.2, we can see that despite exists most probable paths, the algorithm will not always choose them to avoid bottlenecks. Furthermore, the paths with lower weights have fewer possibilities to be selected since they present

high congestion levels. Moreover, completely congested paths (*i.e.,* paths presenting 0 as weight) are not even choosable since they would drop incoming packets.

Several data-plane based load-balancing solutions [28–30], leverage the *flowlet* granularity to enhance the load-balancing process. Our mechanism performs the previous weighted path selection for the different *flowlets* into a flow to load-balance the packets from arriving flows along multiple paths. Edge switches shape *flowlets* from arriving packets comparing the inter-packet gap (*i.e.,* elapsed time between consecutive packets) and a pre-establish threshold. Then, if the inter-packet gap is higher, incoming packets will shape a *flowlet*. Thereby, edge switches build new *flowlets* as the inter-packet gap surpass the threshold. We set a high threshold to avoid packet reordering since it ensures that the time interval between consecutive*flowlets* is higher than the packet's delay. We set the threshold regarding the order of the network Round Trip Time (RTT). Then, the threshold value is a few hundred microseconds for DCN.

3.4. Summary

This section summarizes the load-balancing mechanism functioning and illustrates how its design meets the proposed contributions in section 1.2. The proposed load-balancing mechanism consists of two subsystems working together to achieve the weighted load-balancing scheme.

The section 3.2 introduced the first subsystem, the probe-based congestion information collection. This subsystem estimates the congestion of each existing path among source and destination using the links and devices' state information. Hence, this subsystem is decisive in building the proposed routing mechanism (*i.e.,* to meet the first and second contributions from section 1.2). Initially, we designed an algorithm to compute the weights coefficients regarding the links and devices' state information. Still, we intended to characterize the impact of the device-state information on the routing performance. Thus, we built the Algorithm 1 as a flexible algorithm that allows two different configurations in the weights computing (*i.e.,* using solely

link-state or both link-state and device-state information). Consequently, this flexible design allows us to implement our proposed weighted load-balancing mechanism in any of these configurations to meet the third contribution proposed in section 1.2. To adjust the Algorithm 1 to work in any of its two possible configurations its just needed to modify its final line (line 19) and set the *weighted_ congestion* variable as the *link_ utilization* or as the sum of the sum of the *queue_ depth* and *link_ utilization* respectively.

The section 3.3 introduced the second subsystem, the weighted path selection. This subsystem relies on the Algorithm 2 to set an interval of choosing for each path among source and destination regarding the weights computing in the first subsystem. Therefore, the interval of choosing maps the network congestion into the probability of choosing each path. We design the Algorithm 2 as an alternative to solutions computing only the best routing path. Consequently, our algorithm surpasses the shortcomings of these solutions (*e.g.,* HULA and CONGA), which easily bottleneck the best routing path since they only select and record the less congested path.

Chapter 4

Evaluation and Analysis

4.1. Test Environment

This section presents the probe-based mechanism test environment to evaluate its performance regarding the delay, throughput, and packet loss. The section 4.1.1 introduce the evaluation topology, section 4.1.2 describes the mechanism design parameters, section 4.1.3 presents the performance metrics evaluated, section 4.1.4 describes the traffic generation procedure, and section 4.1.5 details the performance monitoring tools and process.

Moreover, the test environment and its different processes described in this section were built on a *Ubuntu 16.04* virtual machine running on a laptop with a processor *Core i7-9750H*. We set the virtual machine system settings as 11GB memory RAM and 6 CPU threads.

4.1.1. Evaluation Topology

The mechanism evaluation was implemented in an emulated 3-tier Fat-Tree topology as shown in Figure 4.1. This topology consists of two pods connected by two core switches. Each pod incorporates two aggregation switches connecting two edge

switches. The links bandwidth was limited as a result of the P4 bmv2 switches' inherent low performance [47] and computational constraints. Consequently, we scaled the bandwidth of the links compared to a production data center. Thus, we set each link connecting a pair of switches' bandwidth as 2,5Mbps.

Due to computational limitations, each edge switch is connected to two servers, and we also set the link bandwidth between every server and its corresponding edge switch as 2.5Mbps. Thus, the network is not oversubscribed as the four servers per pod can together use the 10Mbps bandwidth available across the pod.

It is noteworthy that we used the *P4-Utils* package to build and test our P4 solution. *P4-Utils* is a *Python* package used to provide a testing and prototyping platform based on P4 language [48]. This package allows the virtual network creation and testing using Mininet. Hence, *Mininet* was the emulation tool adopted to emulate the proposed topology. Additionally *P4-Utils* provides bmv2 P4 programmable targets.

We choose the *P4-Utils* since it uses bmv2 targets. TheP4 - Language Consortium recommends these targets as the tool for developing, testing, and debugging P4 data planes [49]. Moreover, bmv2 targets supports the $P4_{16}$ ongoing standard and the *v1model* architecture.

4.1.2. Design Parameters

As we depict in Chapter 3 our proposal relies on two important design parameters. First, the probe frequency must be a few times the RTT to appropriately react to the volatile traffic conditions without overwhelming the network [28, 29]. Thence, we set the probe frequency as 200 ($\sim \mu s$) as the RTT was about 40 ($\sim \mu s$) in the absence of load on the network. Second, the *flowlet* inter-packet gap must be on the RTT order to avoid the packet reordering [28, 29]. Thereby, we also set the the *flowlet* inter-packet gap as 200 ($\sim \mu s$).

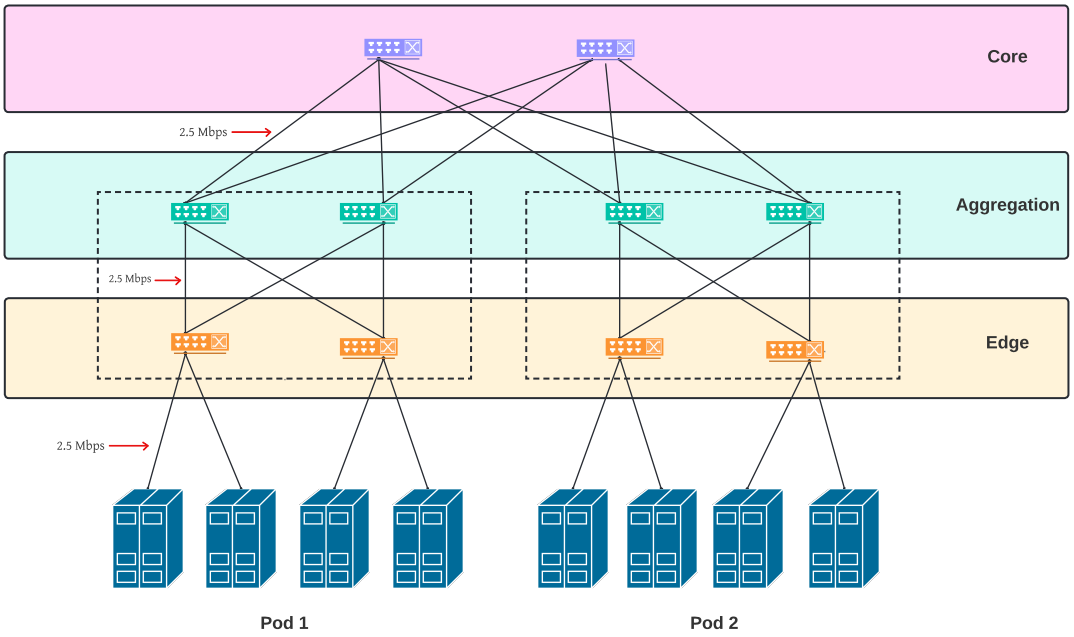

Figure 4.1: Evaluation topology - adapted from [1]

4.1.3. Performance Metrics

Defining the performance evaluation metrics is essential to accomplishing the proposed objectives. Thus, we evaluated the proposed load balancing mechanism regarding the delay, throughput, and packet loss against ECMP in various network scenarios, varying the network's load.

Furthermore, as depicted in section 3.4 we looked to establish the impact of modeling the network congestion using the device-state information in the data-plane-based routing solutions performance. Consequently, we also evaluated our proposed mechanism performance (regarding the delay, throughput, and packet loss) on its two possible configurations.

It is noteworthy that these two configurations simply modify the network congestion modeling. The first configuration models the network congestion by computing the coefficient weights using solely link-state parameters. Furthermore, the second configuration computes the coefficient weights using link-state and device-state pa-

rameters.

Since we use the Distributed Internet Traffic Generator (D-ITG) [50] tool to measure the solution's performance, next, we define each of these metrics as this tool describes them:

- *Delay:* It defines the one-way delay, and D-ITG computes its value as the difference between the transmitted and the received time of each packet. Therefore, increasing the delay leads to higher network congestion.
- *Throughput:* It defines the average communication bit rate measured in $Kbit/sec$. D-ITG computes its value as the ratio between the number of bits received and the sample time taken throughout the evaluation. Therefore, decreasing the throughput indicates higher network congestion.
- *Packet Loss:* It defines the percentage of packets lost on the round-trip path (i.e., only on the one-way path). D-ITG computes its value by comparing the number of packets received at the destination to the packets sent from the source. Therefore, increasing the packet loss reveals higher network congestion.

4.1.4. Traffic Generation

We used the data-mining empirical workload [5] to generate the network traffic, emulating realistic DCN traffic conditions. In particular, this workload is obtained from a production data center composed of a 1,500-node cluster that supports data mining tasks. Figure 4.2 shows the *Cumulative Distribution Function* used to model the network traffic. This figure shows that the data-mining workload is *heavy-tailed* since just a small slice of the whole number of flows (called *large flows*) carries most of the total traffic data.

We replicated a client-server communication model running each server connected to the edge switches as both client and server simultaneously. Next, each client generated traffic (according to the workload in Figure 4.2) to a randomly selected remote server located in a different pod. This ensures that each communication

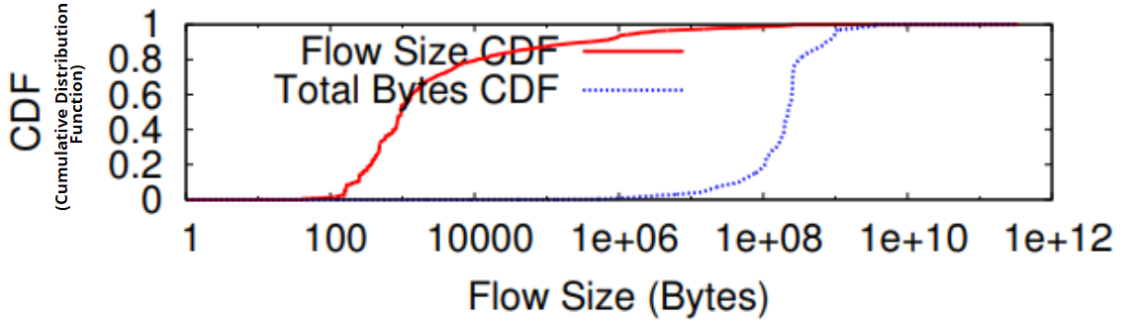

Figure 4.2: Traffic distribution used in the network's traffic generation - source [5].

passes over the core switches, genuinely leveraging the bisection bandwidth offered by the topology. It is noteworthy that we use *iPerf3* [51] to generate the traffic at each client. *iPerf3* is a commonly used network testing tool to create TCP and *User Datagram Protocol* data streams.

The number of persistent TCP connections to a remote server that each client runs depends on the desired network load. Hence, we use the number of persistent connections to model each network load evaluated.

We evaluated the proposed mechanism performance for nine equally-spaced network loads, from 10 % to 90 % of the network's load.

4.1.5. Performance Monitoring

We used the D-ITG [50] to measure the proposed load-balancing mechanism performance for each value of the network's load. D-ITG returns performance parameters from the communication between an D-ITG client and server. The client and server were located at different pods to exploit the network's bisection bandwidth. Moreover, we sent from the client a 64512 Bytes D-ITG package to the server to monitor the transmission performance. At the end of the communication D-ITG generated a file containing the communication packet loss, the average delay, and the average throughput.

We ran this experiment with forty-two random seeds for each possible network load

defined in section 4.1.4. In brief, we measured the delay, throughput, and packet loss forty-two times for each network load to calculate the average of the forty-two runs for each particular load.

Additionally, we repeated this process for each of the three evaluated mechanisms: i.e., ECMP, the proposed load balancing mechanism using solely link-state parameters, and the proposed load balancing mechanism using both link-state and device-state parameters simultaneously.

4.2. Results Analysis

In this section, we present the proposed probe-based mechanism experimental results regarding the performance metrics proposed in the section 4.1.3.

As we have discussed in the section 4.1, we also evaluated the ECMP algorithm to achieve a comparative evaluation of the proposed solution regarding the existing DCN load-balancing schemes. Furthermore, we evaluated our proposed mechanism in two different cases (i.e., using the link utilization solely and using both the link utilization and the queue depth) to establish a comparative evaluation of the device-state impact on the routing performance.

Hence, in the following sections, we will present the experimental results of the following mechanisms:

- *ECMP*.
- *links_state*: Our proposed mechanism using weights coefficients calculated regarding the link utilization solely.
- *links-devices_state*: Our proposed mechanism using weights coefficients calculated regarding the link utilization and the queue depth.

4.2.1. Delay

Figure 4.3 shows the average delay results as the network load increases.

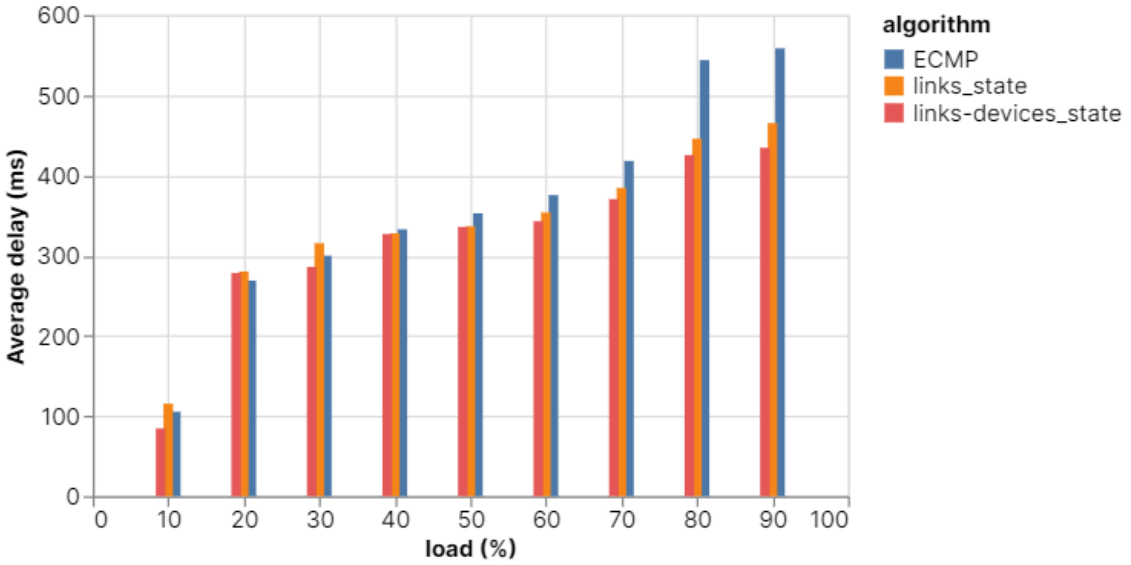

Figure 4.3: Delay evaluation

At small load percentages, the performance of all three different load-balancing mechanisms is almost identical since, in these cases, the network's available bandwidth is high enough to provide a greater tolerance compared to high-traffic load network scenarios. Therefore, the network delay is not significantly affected by the flow routing mechanism chosen as the three mechanisms leverage the network's bisection bandwidth similarly.

Still, at higher load percentages, the overall network bandwidth is reduced, so the flow routing mechanisms need to choose the routing path precisely to leverage the bisection bandwidth properly (*i.e.,* avoiding the forwarding over the most congested paths). Hence, in this scenario ECMP presents the worst performance compared to weighted mechanisms as it performs probabilistic load-balancing disregarding the paths' congestion conditions. Therefore, the weighted schemes reduce the network's delay as they recognize and adopt the network congestion to perform optimal routing.Thus, the *links_ state* mechanism reduces the ECMP delay up to 1.2x, and the *links-devices_ state* mechanism reduces the ECMP delay up to 1.28x.

Moreover, both weighted mechanisms' performance is nearly similar at higher load percentages. Nevertheless, the *links-devices_state* scheme achieved slightly better results for every high load network scenario, as it improves the network congestion characterization using both the link utilization and the queue depth. Thus, the *links-devices_state* scheme reduces the *links_state* delay up to 1.07x.

4.2.2. Throughput

Figure 4.4 shows the average throughput results as the network load increases.

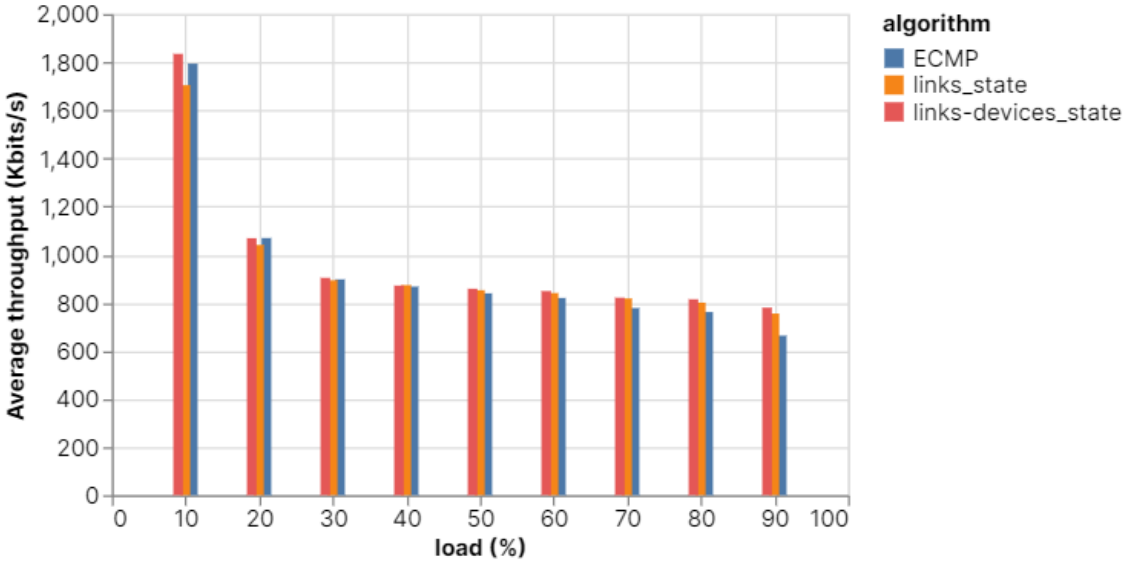

Figure 4.4: Throughput evaluation

The analysis of these results is similar to the one presented in the section 4.2.1 as the evaluation conditions are the same. Therefore, the performance of all three different load-balancing mechanisms was almost identical at small load percentages.

Furthermore, at higher load percentages, ECMP presented the worst performance compared to weighted mechanisms as ECMP still forwarding packets through links presenting higher link utilization indiscriminately. Thus, the *links_state* mechanism increases the ECMP throughput up to 1.14x, and the *links-devices_state* mechanism increases the ECMP throughput up to 1.17x.

Moreover, both weighted mechanisms' performance is nearly similar at higher load percentages. Nevertheless, the *links-devices_state* scheme achieved slightly better results for every high load network scenario. Thus, the *links-devices_state* mechanism increased the *links_state* throughput up to 1.03x.

4.2.3. Packet Loss

Figure 4.5 shows the average packet loss results as the network load increases.

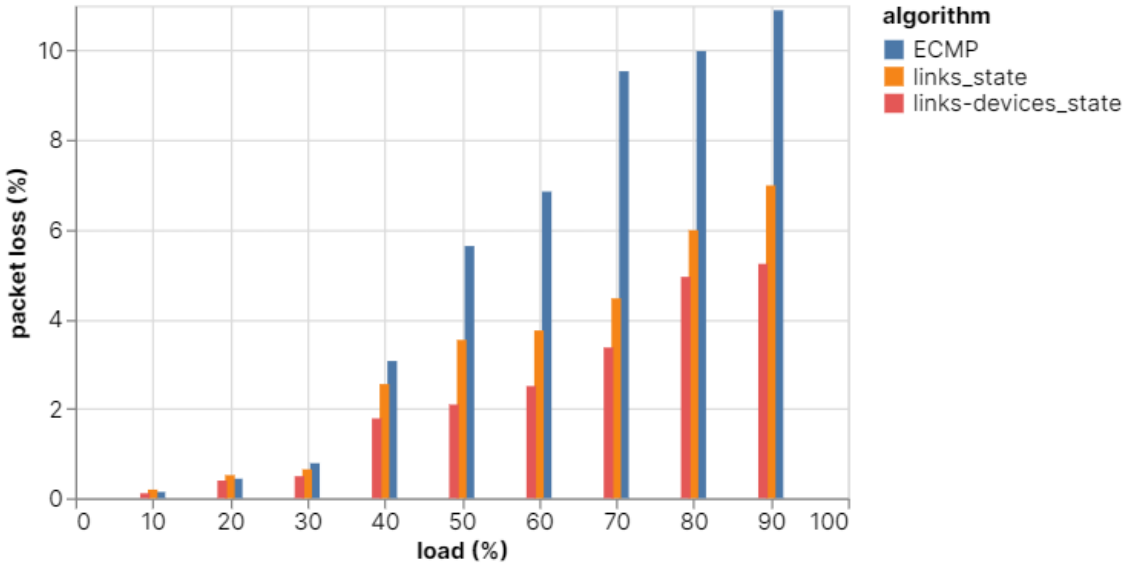

Figure 4.5: Packet Loss evaluation

The analysis of these results is similar to the one presented in the section 4.2.1 as the evaluation conditions are the same. Therefore, the performance of all three different load-balancing mechanisms was almost identical at small load percentages.

Furthermore, at higher load percentages, ECMP presented the worst performance compared to weighted mechanisms as ECMP still forwarding packets through links presenting higher packet-loss indiscriminately. Thus, the *links_state* mechanism reduced the ECMP packet loss up to 1.56x, and the *links-devices_state* mechanism reduced the ECMP packet loss up to 2.08x.

Moreover, at higher load percentages, the *links-devices_ state* scheme achieved better results compared to *links_ state* for every high load network scenario. Thus, the *links-devices_ state* mechanism reduced the *links_ state* packet loss up to 1.33x.

It is noteworthy that among the three performance metrics evaluated, packet loss is the parameter at which the weighted mechanisms present the more significant improvements compared to ECMP. Moreover, the *links-devices_ state* mechanism also presented the higher improvements compared to the *links_ state* mechanism in the packet loss evaluation. This happens because the *links-devices_ state* scheme used the queue depth as a routing parameter, so this algorithm optimized this metric. Besides, optimizing the queue depth leads to forward packets through paths presenting switches with a broad buffer availability. Then, these paths will perform lower packet loss results as they provide less congested buffers. Consequently, the *links-devices_ state* scheme improved the *links_ state* packet loss performance, as the *links_ state* scheme dismissed the switches buffers occupancy in the routing process.

Chapter 5

Conclusions and Future work

5.1. Conclusions

This thesis presents the answer to the question: **How to provide an efficient routing mechanism from a programmable data plane?**.

A data plane weighed load-balancing mechanism using both links' and devices' state information was designed to answer this question. It was evaluated in a 3-tier Fat-Tree topology regarding the delay, throughput, and packet loss. The results show the following:

- The proposed weighted mechanism outperformed the ECMP results regarding the delay, throughput, and packet loss for high network load scenarios.
- As the proposed weighted mechanism runs at line rate leveraging the data plane programmable switches; it adopts features to effectively respond to the volatility of DCN traffic, operating a probe frequency set as 200 ($\sim \mu s$).
- Experimental results showed that using the links' and devices' state information to estimate congestion outperforms the overall network performance compared to solutions solely using the link's state information.

Consequently, we achieved an efficient data-plane weighted load-balancing solution, outperforming the performance of classical solutions such as ECMP and maintaining a low probe frequency to react accurately to the DCN traffic volatility. Moreover, we characterize the impact of modeling the network congestion using device-state information in the data-plane-based routing solutions performance.

5.2. Future work

- Implement a P4-based load-balancing mechanism using regular traffic to collect the network congestion information.
- Characterize the impact of other congestion metrics different from the link utilization and the queue depth in the routing performance.
- Evaluate the P4-based routing mechanisms' effectiveness in network scenarios different from DCN (*e.g.,* large networks).

Bibliografía

[1] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey," *IEEE Communications Surveys and Tutorials*, vol. 15, no. 2, pp. 909–928, 2013.

[2] 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, no. 1, pp. 14–76, jan 2015.

[3] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research," 2021. [Online]. Available: http://arxiv.org/abs/2101.10632

[4] A. P. I. W. Group, "P4Runtime Specification," pp. 1–54, 2018.

[5] A. Greenberg, S. Kandula, D. A. Maltz, J. R. Hamilton, C. Kim, P. Patel, N. Jain, P. Lahiri, and S. Sengupta, "VL2: a scalable and flexible data center network," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 51–62, aug 2009. [Online]. Available: https://dl.acm.org/doi/10.1145/1594977.1592576

[6] H. I. Bahari and S. S. M. Shariff, "Review on data center issues and challenges: Towards the Green Data Center," *Proceedings - 6th IEEE International Conference on Control System, Computing and Engineering, ICCSCE 2016*, pp. 129–134, apr 2017.

[7] M. Chen, H. Jin, Y. Wen, and V. Leung, "Enabling technologies for future data center networking: A primer," *IEEE Network*, vol. 27, no. 4, pp. 8–15, 2013.

[8] M. Alizadeh and T. Edsall, "On the data path performance of leaf-spine datacenter fabrics," *Proceedings - IEEE 21st Annual Symposium on High-Performance Interconnects, HOTI 2013*, pp. 71–74, 2013.

[9] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, aug 2008. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1402946.1402967

[10] M. Alizadeh and T. Edsall, "On the data path performance of leaf-spine datacenter fabrics," *Proceedings - IEEE 21st Annual Symposium on High-Performance Interconnects, HOTI 2013*, pp. 71–74, 2013.

[11] E. Nepolo and G. A. Lusilao Zodi, "A Predictive ECMP Routing Protocol for Fat-Tree Enabled Data Centre Networks," *Proceedings of the 2021 15th International Conference on Ubiquitous Information Management and Communication, IMCOM 2021*, jan 2021.

[12] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks."

[13] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, aug 2013. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2534169.2486019

[14] C.-Y. Hong Srikanth Kandula Ratul Mahajan Ming Zhang Vijay Gill Mohan Nanduri Roger Wattenhofer Microsoft, "Achieving High Utilization with Software-Driven WAN," 2013.

[15] S. A. R. Shah, W. Seok, J. Kim, S. Bae, and S. Y. Noh, "CAMOR: Congestion Aware Multipath Optimal Routing Solution by Using Software-Defined Networking," *2017 International Conference on Platform Technology and Service, PlatCon 2017 - Proceedings*, 2017.

[16] T. T. Huong, N. D. D. Khoa, N. X. Dung, and N. H. Thanh, "A global multipath load-balanced routing algorithm based on Reinforcement Learning in SDN," *ICTC 2019 - 10th International Conference on ICT Convergence: ICT Convergence Leading the Autonomous Future*, pp. 1336–1341, oct 2019.

[17] A. Jerome, M. Yuksel, S. H. Ahmed, and M. Bassiouni, "SDN-based load balancing for multi-path TCP," *INFOCOM 2018 - IEEE Conference on Computer Communications Workshops*, pp. 859–864, jul 2018.

[18] M. Doshi, A. Kamdar, and K. Kansara, "Multi-constraint qos disjoint multipath routing in SDN," *Advances in Intelligent Systems and Computing*, vol. 810, pp. 377–387, 2018.

[19] M. C. Nkosi, A. A. Lysko, and S. Dlamini, "Multi-path load balancing for SDN data plane," *2018 International Conference on Intelligent and Innovative Computing Applications, ICONIC 2018*, pp. 1–6, 2019.

[20] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The Nature of Datacenter Traffic: Measurements Analysis," 2009.

[21] A. O. Basil, "A Consistency Based Research : P4 Versus OpenFlow and the Future of Software Defined Networks," no. March 2016, p. 2019, 2019.

[22] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary, "Offloading real-time DDoS attack detection to programmable data planes," *2019 IFIP/IEEE Symposium on Integrated Network and Service Management, IM 2019*, no. December, pp. 19–27, 2019.

[23] F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, and P. Castoldi, "P4 edge node enabling stateful traffic engineering and cyber security," *Journal of Optical Communications and Networking*, vol. 11, no. 1, pp. A94–A95, 2019.

[24] Z. Zuo, R. He, X. Zhu, and C. Chang, "A novel software-defined network packet security tunnel forwarding mechanism," *Mathematical Biosciences and Engineering*, vol. 16, no. 5, pp. 4359–4381, 2019.

[25] 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, no. 1, pp. 14–76, jan 2015.

[26] 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," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, jul 2014.

[27] M. Pizzutti and A. E. Schaeffer-Filho, "An Efficient Multipath Mechanism Based on the Flowlet Abstraction and P4," in *2018 IEEE Global Communications Conference, GLOBECOM 2018 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 2018.

[28] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "CONGA: Distributed congestion-aware load balancing for datacenters," *Computer Communication Review*, vol. 44, no. 4, pp. 503–514, feb 2015.

[29] 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*, mar 2016. [Online]. Available: http://dx.doi.org/10.1145/2890955.2890968

[30] J. L. Ye, C. Chen, and Y. Huang Chu, "A Weighted ECMP Load Balancing Scheme for Data Centers Using P4 Switches," in *Proceedings of the 2018 IEEE 7th International Conference on Cloud Networking, CloudNet 2018*. Institute of Electrical and Electronics Engineers Inc., nov 2018.

[31] K. F. Hsu, P. Tammana, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Adaptive weighted traffic splitting in programmable data planes," *SOSR 2020 - Proceedings of the 2020 Symposium on SDN Research*, pp. 103–109, 2020.

[32] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker, "Contra: A Programmable System for Performance-aware Routing."

[33] Gupta, "Analysis of an Equal-Cost Multi-Path Algorithm," vol. 6, no. 2, p. 103, 2000.

[34] M. Chiesa, G. Kindler, and M. Schapira, "Traffic Engineering with {ECMP}: An Algorithmic Perspective," *Proc. IEEE INFOCOM*, vol. 25, no. 2, pp. 1590–1598, 2014. [Online]. Available: http://www.dia.uniroma3.it/{~}compunet/www/docs/chiesa/ecmp.pdf

[35] "A survey on the programmable data plane: Abstractions, architectures, and open problems," *IEEE International Conference on High Performance Switching and Routing, HPSR*, vol. 2018-June, 2018.

[36] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using open flow: A survey," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1, pp. 493–512, 2014.

[37] N. Feamster, J. Rexford, and E. Zegura, "Sdnhistory," 2014.

[38] "P4 Language Specification." [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html

[39] 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, no. 3, pp. 87–95, 2014.

[40] H. Stubbe, "P4 Compiler & Interpreter: A Survey," *Network Architectures and Services*, no. May, pp. 47–52, 2017. [Online]. Available: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2017-05-1/NET-2017-05-1{_}06.pdf

[41] B. Arzani, A. Gurney, S. Cheng, R. Guerin, and B. T. Loo, "Impact of path characteristics and scheduling policies on MPTCP performance," *Proceedings - 2014 IEEE 28th International Conference on Advanced Information Networking and Applications Workshops, IEEE WAINA 2014*, pp. 743–748, 2014.

[42] N. Spring, M. Dontcheva, M. Rodrig, and D. Wetherall, "How to resolve IP aliases," *Univ. Michigan, UW CSE Tech. Rep*, pp. 4–5, 2004.

[43] E. Vanini, R. Pan, P. Taheri, T. Edsall, and M. Alizadeh, *Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching*, 2017. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini

[44] "RFC 7637 - NVGRE: Network Virtualization Using Generic Routing Encapsulation." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7637

[45] "RFC 7348 - Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7348

[46] "A Stateless Transport Tunneling Protocol for Network Virtualization (STT)." [Online]. Available: https://tools.ietf.org/id/draft-davie-stt-06.html

[47] H. Tu Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A P4 Language Benchmark Suite." [Online]. Available: http://dx.doi.org/10.1145/3050220.3050231

[48] "Introduction — p4-utils 1.0 documentation." [Online]. Available: https://nsg-ethz.github.io/p4-utils/introduction.html#about-p4-utils

[49] "The bmv2 simple switch target." [Online]. Available: https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md

[50] Alessio Botta, Walter de Donato, Alberto Dainotti, Stefano Avallone and A. Pescapé, "D-ITG 2.8.1 Manual," 2013. [Online]. Available: https://traffic.comics.unina.it/software/ITG/manual/

[51] "iperf - the tcp, udp and sctp network bandwidth measurement tool." [Online]. Available: https://iperf.fr/

Efficient Flow Routing From Programmable Data Plane

ANNEXES

Undergraduate degree

Nicole Venachi Pizo
Jorge Manuel Castillo Camargo

Advisor: PhD. Oscar Mauricio Caicedo Rendón
Co-Advisor: PhD. Cristhian Nicolás Figueroa Martínez

Department of Telematics
Faculty of Electronic and Telecommunications Engineering
Universidad del Cauca
Popayán, August 2022

Annex A

Source Code

Annex A presents the Github repository link containing the entire project source code.

`https://github.com/NicoleVenachi/p4_probebased_loadbalancing`

Annex B

Publications

Annex B presents the scientific paper written during the undergraduate work development:

- **Ludwing Nicole Venachi**, **Jorge Manuel Castillo Camargo**, Oscar Mauricio Caicedo Rendon, Cristhian Nicolás Figueroa Martínez. **Weighted Load-balancing Using Queue Occupancy From Programmable Data Planes** Elsevier - Journal of Network and Computer Applications.
  - Status: Writtend and ready to be sent
  - Classification: A1.
  - Impact Factor: 7.574.
  - H-index: 115 Scimago.