

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Faculty of Engineering

Master of Science in Computer Engineering

Master Thesis

Network Slicing and Quality of Service in Programmable Data Planes

An experimental approach to develop Network Slicing solutions within the 5G Transport layer with P4-enabled devices



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Supervisor:

prof. Paolo Bellavista

Co-supervisor:

prof. Armir Bujari

Candidate:

Lorenzo Riccardi

In collaboration with
UAntwerp & imec (IDLab)
prof. Johann Marquez-Barja

October 2025

To my mother, Nada
To my father, Roberto
To my brother, Cristian

Sometimes it is the people no one imagines anything of
who do the things no one can imagine.
Benedict Cumberbatch as Alan M. Turing in 'The Imitation Game'

Acknowledgements

I would like to express my deepest gratitude to everyone who supported me during the development of this thesis and my entire academic journey. First and foremost, I want to thank my supervisors, Prof. Paolo Bellavista and Prof. Armir Bujari, for their invaluable guidance, expertise, and encouragement throughout this project. Their insights and feedback were instrumental in shaping the direction of my research and helping me overcome challenges along the way. I am also deeply grateful to Prof. Johann Marquez-Barja from UAntwerp and imec (IDLab) for providing me with the opportunity to collaborate with their esteemed institution. Your support and resources were crucial in the successful completion of this thesis.

What has been done in this thesis would not have been possible without the help and support of my colleagues and friends. I want to thank my friends from the University of Bologna for their day to day support, jokes, and experiences made together: Riccardo, Davide, Edoardo, Luca, Luca, Federica, Francesca, Tommaso, Michele, Stefano and Federico. Your presence has helped me through the toughest exams, but also the hardest moments of my life. And, of course, I will never forget the nights spent together in the library, studying until late, with the company of some good, giant, pizzas.

Then, I want to thank the friends I made along the way during my stay in Belgium, who made the experience unforgettable, and whose support has been fundamental for every step of the journey, both within and outside the professional environment: Daniel, Raul, Miguel, Nina, Giselle, Andreas, Julian, Vincent, David, Paola and everyone else from IDLab Antwerp. I hope to see you all again very soon, and in the meanwhile I wish you all the best for your bright future.

Xhulio, you have been a great mentor and friend during my stay in Antwerp, and I want to thank you for your patience, your time, and your support. I learned a lot from you, both professionally and personally, and I hope to keep in touch in the future. For the same reasons, especially for the patience, I want to thank Arno and Alessandro as well, who showed me that true friendship is not about being inseparable, but about being separated without losing touch.

A special thanks goes to my friends in Rimini (and Silvi), especially Marco and Riccardo, who have been a second family for longer than I can remember: every day spent together influenced who I am today, and I will always be grateful for that. Many of you managed to be here even today for me, even if we are all busy with our lives, jobs, and projects. Every

one of you gave me a different insight on life, teaching me something that I will carry every day with me. I am incredibly lucky to have you all in my life and to be part of such a great, big, group of friends.

Also from Scavolino, where I spent my childhood and countless summers, I want to thank Alessandra and Matilde: we grew up together, and I am proud to call you my friends and to see the women you have become. My gratitude goes also to everyone else from the Scavolino family, such as Alessandra, Alessandro, Laura, Matteo and many others, who are always there for me, even if we do not see each other often.

Finally, Davide, for showing me what true dedication and passion is, and for being, more than a roommate, a friend to me.

Last but not least, I want to thank my family for their unconditional love and support throughout my life. To my mother, Nada, and my father, Roberto, for being always there for me, for your endless patience, for giving and encouraging me to take every opportunity to grow and learn, and for giving me a solid foundation to build my dreams upon. To my little brother, Cristian, who is not so little anymore, for being alongside me in every step of this journey, shown in your own, unique, way; and, of course, for having so much patience with me, as I have not always been the best big brother in the world.

I am truly blessed to have such a wonderful family, and I dedicate this thesis to all of you. Thank you for everything.

Summary

5G has capabilities which exceed what has been seen for the previous generations of mobile networks: it can handle very heterogeneous traffic, each one with its own requirements in terms of resources needed for normal operativity. There is the necessity to support Ultra-Reliable Low-Latency Communications (URLLC) along with Enhanced Mobile Broadband (eMBB) and Massive Machine-Type Communications (mMTC) traffic, and every flow is competing for resources with each other.

Therefore, solutions to handle such strict requirements emerged, and got standardized in what is the Fifth Generation (5G) of mobile networks: in particular, Network Slicing enables the scalability needed by modern communications to operate and allows for efficient management of network resources, isolating and distributing them according to various strategies.

Many implementations of it are available for the 5G Core (5GC) and the Radio Access Network (RAN), but there is not a widespread solution for the Transport layer, which is a crucial part of the whole infrastructure. Thus, this thesis proposes an implementation of a Network Slicing solution in the Transport layer with Quality of Service (QoS) delivery, using the Programming Protocol-Independent Packet Processors (P4) language. This latter tool promises very high performances, both in terms of throughput and latency, as it provides the possibility to deploy custom network configurations and behaviors both in software and in hardware, specialized, targets.

The framework is then deployed in a real testbed and evaluated, also against more widespread and properly configured alternatives, to assess the performance of our solution.

An orchestration module for the Control Plane (CP) will be then proposed, to allow for a seamless and unified management of multiple, different, P4-enabled devices, each one with its own capabilities and exposed metadata. This module will lay the foundations for a real-world deployment, as industries will be able to gradually introduce P4-enabled devices in their physical infrastructure.

Contents

1	Introduction	1
2	Background and Related Work	5
2.1	The Internet and its protocols	5
2.2	Queue management	13
2.2.1	Available algorithms and techniques	14
2.3	Towards scalable mobile networks: 5G	18
2.3.1	Reference architecture	18
2.3.2	Network programmability	24
2.3.3	Network Slicing	25
2.4	A Primer on P4	28
2.4.1	Main P4 constructs	29
2.5	Related Work	32
3	Proposed framework	36
3.1	Framework architecture	36
3.2	Control Plane	38
3.2.1	Orchestrator structure	39
3.2.2	Agent structure	41
3.3	The Simple Switch gRPC	44
3.3.1	Architecture	45
3.4	Data Plane implementation	47
3.4.1	Data structures	47
3.4.2	Parser	50
3.4.3	Ingress	52
3.4.4	Traffic Manager	55
3.4.5	Egress	55
3.4.6	Checksum computation	59
3.4.7	Deparser	60
3.5	In-band Network Telemetry	60
4	Results and evaluation	62
4.1	Experimental settings	62
4.1.1	Hardware setup	63
4.1.2	Software setup	64

4.2	Results	67
4.2.1	A comparison with Linux Traffic Control	73
5	Conclusions and Future Work	77
A	The P4 programming language	79
A.1	Language syntax and semantics	79
A.1.1	Literals	81
A.1.2	Program structure	82
A.1.3	Calling convention	83
A.1.4	Data types	83
A.1.5	Expressions	85
A.1.6	Statements	87
A.2	Complete grammar	88
B	CoDel P4 implementation	106
B.1	Original CoDel P4 implementation	106
B.2	Modifications made	109
	References	111

List of Figures

1.1	Deployment scenario of the thesis	2
2.1	Ethernet and VLAN headers structure	7
2.2	IPv4 header structure	9
2.3	TCP header structure	11
2.4	UDP header structure	13
2.5	A generic queue with associated operations	14
2.6	Random Early Detection algorithm	16
2.7	Timeline of mobile networks	19
2.8	5G architecture	19
2.9	NSA and SA deployment of 5G	20
2.10	RAN architecture evolution	22
2.11	O-RAN general architecture	23
2.12	O-RAN typical deployment	23
2.13	Software-Defined Networking architecture	25
2.14	Network Slicing architecture	26
2.15	Network Slicing visualization	27
2.16	P4 deployment process	28
2.17	MAT structure and packet matching	31
3.1	General architecture of the proposed framework	38
3.2	The vlmodel architecture	45
3.3	Parser state diagram	52
3.4	Traffic Manager architecture	56
3.5	Allowed paths of a packet through our P4 pipeline	60
4.1	General overview of the testbed	62
4.2	Network topology of the testbed	64
4.3	First experiment results	68
4.4	Second experiment results	70
4.5	Third experiment results	71
4.6	Fourth experiment results	72
4.7	First TC experiment results	75
4.8	Second TC experiment results	76

List of Tables

2.1	Common Internet protocols and their respective EtherType	7
2.2	Common Transport protocols and their respective IPv4 protocol number	9
2.3	Most important <i>well-known ports</i>	10
2.4	Key Performance Indicators for the different 5G services	20
2.5	Structure of <code>standard_metadata_t</code> in the vlmodel architecture	33
3.1	Orchestrator API endpoints	40
3.2	Mandatory columns for telemetry reporting	41
3.3	Agent API endpoints	42
3.4	Constants, macros and types defined for the Data Plane	47
3.5	Ethernet header fields	48
3.6	IPv4 header fields	48
3.7	INT header fields	49
3.8	VLAN header fields	49
3.9	Custom metadata defined for the Data Plane	49
3.10	Reasons for telemetry reporting	60
4.1	Specifications of the DARPA2 infrastructure	63
4.2	Specifications of the gNodeB machines	63
4.3	Specifications of the UPF machines	63
4.4	Specifications of the P4 router machine	63
4.5	Specifications of the Raspberry Pi 3 Model B+	64
4.6	Software running on the gNodeB machines	65
4.7	Software running on the UPF machines	65
4.8	Software running on the P4 router machine	66
4.9	Software running on the Orchestrator	66
4.10	First experiment flows configuration	69
4.11	Second experiment flows configuration	69
4.12	Third experiment flows configuration	72
4.13	Fourth experiment flows configuration	73
4.14	Parameters used for the Token Bucket Filter qdiscs	74
4.15	First TC experiment flows configuration	74
4.16	Second TC experiment flows configuration	76
A.1	Base types in P4	84
A.2	Default values for P4 data types	85

Listings

3.1	Parser implementation	50
3.2	P4 code for QoS management of Network Slices	53
3.3	P4 code for the Egress block	56
A.1	P4 identifiers	80
A.2	P4 integer literals	81
A.3	P4 program structure	82
A.4	P4 extern declaration	82
A.5	P4 base types grammar	83
A.6	P4 complete grammar	88
B.1	Original CoDel P4 implementation	106
B.2	Latency target per-slice assignment	110

Chapter 1

Introduction

Since the dawn of humanity, the need for fast communication has been a driving force behind technological advancements. From the invention of the wheel to the development of the Internet, each step has brought us closer together, allowing for the exchange of ideas and information at unprecedented speeds. The complexity of modern communication systems demands a strong standardization to ensure interoperability, efficiency and the possibility to continue thriving and innovating in the future, in a world where the only constant is change.

Nowadays, the world is more interconnected than ever before. The Internet has become a global network that connects billions of people, devices, and systems. This interconnect-edness has transformed the way we communicate, work, and live.

During the last decades, the Internet has evolved from a simple network of computers to an extremely complex and heterogeneous ecosystem of devices, applications, and services. The need for management and maintenance of such a complex system has led to the development of various protocols and standards, which have become essential for the functioning of the Internet as we know it today.

Nowadays, the Internet is accessed through mobile nodes more than fixed stations, such as smartphones, tablets, laptops and smartwatches. This trend led to the development of what are known as mobile networks: starting from the first generation (1G/AMPS) many advancements have been of crucial relevance for the development of contemporary Internet; today's 5G (New Radio) promises a vast array of improvements on top of its predecessor (4G/LTE), especially for what concerns scalability and the management of multiple services, thanks to some unique features such as Network Slicing. To support a Network Slicing deployment the environment itself should present some characteristics able to help in providing the needed basis to manage the existing underlying resources and distribute them among the users who request access to the network.

The research here made has been possible thanks to the longstanding collaboration between Alma Mater Studiorum - Università di Bologna and IDLab - imec & University of Antwerp.

In the Network Slicing context, prior research focused on improving the current solutions available in particular within the RAN and the Core Networks, as well as on the integration

of Machine Learning techniques to improve the performance of the algorithms used in the most diverse scenarios. The Transport layer, that is the N3 link in the general architecture proposed for 5G, is often overlooked, as it is considered a mere transport mechanism for the user data packets, and not a critical component of the network. However, as will be shown in Section 2.5, it is indeed a key component and can improve by a large margin the performance of applications built on top of 5G. Moreover, the use of P4 as a tool to implement Network Slicing at the Transport level is still in its infancy, and whilst there are different works that explore the use of this language in this context, they are often more focused on providing Quality-of-Service rather than implementing a real NS solution, thus *isolating* the slices from each other.

Moreover, there are no currently available solutions for managing different and heterogeneous P4 devices within a network from the Control Plane.

Therefore, this thesis aims to fill this gap by proposing a Network Slicing solution for the Transport layer based on P4-enabled devices.

Moreover, it will provide a framework for managing homogeneously different P4 devices from the Control Plane, enabling a simpler end-to-end network slices' management, and thus permitting a wider adoption in the industry: P4 is recognized as an enabler for more scalable and performant solutions in the networking field, but at the moment it is not widely used in production environments and mainly in research for testing purposes, especially in the context of Network Slicing.

To fill the aforementioned gap, this thesis advocates for the use of P4 to develop a NS-capable SDN framework to be deployed on various P4-enabled devices.

The scenario we are considering is the one identified by Figure 1.1. Here, we have different applications, each one with its own requirements in terms of resources (latency, throughput, reliability, etc.), connected to different antennas in the RAN. These antennas connect to different switches in the Transport Network, composed by both P4-enabled and “standard” devices (to achieve a gradual migration towards a fully P4-based network).

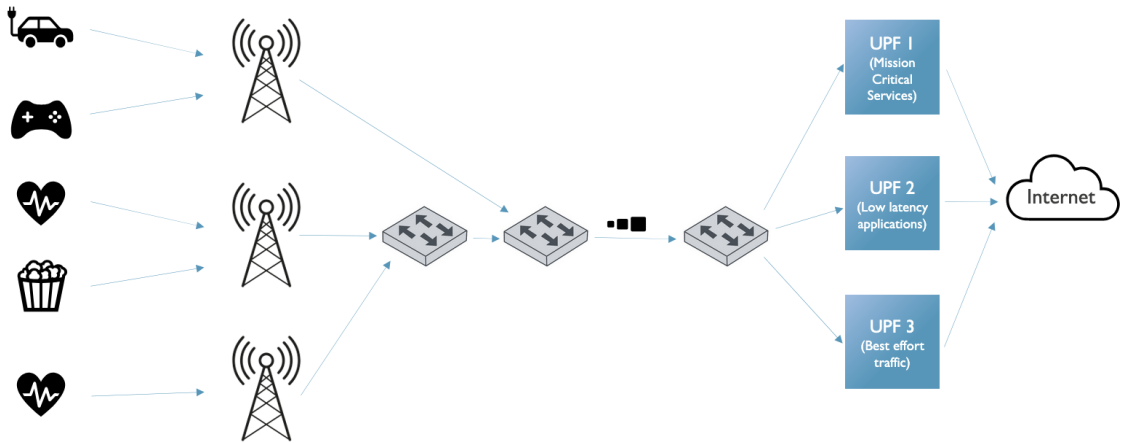


Figure 1.1: Deployment scenario of the thesis

Sometimes, while traversing the network, packets may encounter a bottleneck: this is represented in the figure by the three black rounded squares above the bottlenecked link.

We suppose the existence of a different UPF for each Network Slice, which handles a different kind of traffic: in our scenario, one UPF is for mission-critical applications, one for low-latency applications and the final one for best-effort traffic. These UPFs can be physically located even in geographically distant places, it is not relevant for this thesis to also consider deployment choices for the Core network: this aspect of network management does not influence the design of the solution here proposed, apart from some configurations which will be anyway changeable through the Control Plane.

As the traffic flowing through the N3 link is encapsulated in GTP packets, which operates on top of UDP, we are dealing with UDP traffic. Thanks to P4, it could have effectively been possible to read the inner headers of the packet and base our decisions on that, but because of performance considerations this possibility has been discarded in the end. As will be explored later, UDP determines, in case of missing upper-layers' mechanisms for congestion control, unresponsive flows. Thus, we need to implement a solution that can handle these flows and provide the necessary Quality-of-Service for each Network Slice. This consideration is important, as many congestion control algorithms are designed to work with TCP traffic, which is responsive and can adapt to the network conditions, while UDP traffic does not have this capability by design. This issue will be of a significant relevance in Chapter 3, where we will discuss the design, the implementations, and the possible choices made (or not) in our framework.

The solution needs to be configurable and reconfigurable at runtime in terms of slice creation, deletion and modification, with minimal impact on ongoing traffic through the infrastructure. It must be also independent of the specific P4 targets used. There has to be a way to retrieve the current state of the network in the form of telemetry information, which must be retrievable from the Control Plane independently of the underlying P4 targets. A form of negotiation of the capabilities of the P4 device is advisable, but it is not strictly necessary, and will be left as future work.

A strong choice made is to not suppose a full path of P4-enabled devices, as it is not realistic in a production environment, especially in the first phases of adoption of this technology. Therefore, we had to design a solution with the assumption that some devices could possibly be not aware of custom P4-defined headers, and thus not able to process them: this means that our solution should work as a black-box, where ingress and egress traffic is compliant with existing standards.

To simplify testing and deployment, we will use a single P4-enabled software switch with vast support across almost every feature of the P4 language, rich metadata exposure and a well-defined communication protocol with the Control Plane (*P4Runtime*), known as the **Simple Switch gRPC**.

Other considerations, more technical, will be made in Chapter 3.

Therefore, the main contributions of this thesis are:

- The development of a Network Slicing-capable Data Plane which isolates the different network slices.
- A framework for orchestrating multiple P4 devices in a mixed path of P4 and non-P4 network devices.

We wanted to provide both a general overview on the current state of Internet protocols, network resources management and mobile networks, and an implementation that could fill what already identified as a gap in the literature. Therefore, the structure of the next Chapters in this thesis is:

- Chapter 2 (*Background and related work*) will provide the mandatory knowledge needed to understand the concepts and the technologies that have been dealt with in the course of this thesis, and delve into the current state of the scientific literature for the challenges analyzed.
- Chapter 3 (*Proposed framework*) explains thoroughly the framework proposed delving into the details of both the Data Plane and the Control Plane, their structure and their implementation, along with the considerations, design decisions and compromises made.
- Chapter 4 (*Results and evaluation*) will present the results obtained from the tests performed on the framework, comparing it with other solutions available in the literature and assessing its performance.
- Chapter 5 (*Conclusions*) delivers the conclusions drawn on what has been done and possible future works which could improve the product.

Chapter 2

Background and Related Work

2.1 The Internet and its protocols

Starting from the early days of the Internet, the need for a common language to allow different devices to communicate has been a priority. The organization known as ISO (International Organization for Standardization) proposed a theoretical model, **OSI (Open Systems Interconnection)**, which was established to lay the foundations of a set of standards that would enable interoperability between different systems. However, the OSI model was never fully adopted, and the **Internet Protocol Suite (TCP/IP)** became the de facto standard for communication on the Internet. The Internet Protocol Suite consists in a stack of several layers, similarly to the original OSI model but with some of them restructured and merged for a more practical approach, each with its own set of protocols and functions. The layers it is composed of are:

- **Application Layer:** the uppermost, its responsibility is to provide services to the user, such as web browsing, email, and file transfer. It includes protocols like HTTP, SMTP, and FTP.
- **Transport Layer:** it is its duty to ensure that data is transmitted reliably and efficiently. It includes protocols like TCP and UDP.
- **Internet Layer:** this layer is responsible for routing data across the Internet. It includes protocols like IP (Internet Protocol) and IPv6 (Internet Protocol version 6).
- **Link Layer:** lowest layer, its main job is the physical transmission of data over a network. It includes protocols like Ethernet and Wi-Fi.

In particular, for the interests of this thesis, the latter three layers are of particular importance, as they define how data is transmitted over the Internet.

The Internet Protocol Suite is therefore composed of four layers, designed for efficiency and flexibility. Applications can be built upon them to ensure maximum interoperability, performance and reliability. Each layer has its own set of protocols, each one designed to handle specific types of data and communication (in terms of reliability, speed, etc.).

While we will not delve into the details of the Application Layer protocols, as they are not relevant for the scope of this thesis and we will focus on them by need, it is important

to understand the lower layers, as they provide a foundation for the entire Internet as it is known today.

We will thus analyze these layers in detail, starting from the lowest one, the Link Layer, and moving up to the Transport Layer.

Link Layer

The Link Layer is the lowest layer of the Internet Protocol Suite, responsible for moving packets over two hosts on the same *link*. Conceptually, it corresponds, more or less, to the Layer 2 in the ISO/OSI stack. It also gives for granted the existence of a physical medium, thus not implementing any protocol or mechanism defined for the ISO/OSI Layer 1 (Physical Layer): therefore, the TCP/IP Link Layer is often considered as a combination of both OSI lower layers. It is the only layer that operates within the same local network (in other words, it does not have the concept of *router*), and it is also responsible for the framing of data (the process through which data is divided into smaller frames for transmission) and the addressing of devices on the local network.

It is subdivided into two sub-layers: the **Logical Link Control** (LLC) and the **Media Access Control** (MAC), with the latter being the lower one. The main function of the MAC layer is to provide an abstraction level between the physical layer and the LLC layer, allowing for the implementation and control of different physical media without affecting the LLC layer. At the same time, the LLC sublayer abstracts and multiplexes multiple upper-layer protocols towards its sibling sublayer.

The main addressing scheme here used is the MAC address, which consists in an identifier assigned to each Network Interface Card (NIC) by its manufacturer, thus (almost) guaranteeing the uniqueness of the identifier.

A network device that operates at the Link Layer is usually called a **switch**, which is then responsible for forwarding frames between devices on the same local network.

The two major implementations of the Link Layer are **Ethernet** (IEEE 802.3) and **Wi-Fi** (IEEE 802.11).

In particular, Ethernet is the most widely used protocol for wired networks. Many different version of Ethernet coexist, with different speeds and capabilities, such as Fast Ethernet (100 Mbps), Gigabit Ethernet, and 10 Gigabit Ethernet. Born with a contention algorithm called CSMA/CD (Carrier Sense Multiple Access with Collision Detection), Ethernet has later introduced switches to improve performance and reduce collisions, as the physical medium was not shared anymore between every device in the local network.

The Ethernet header, the outer one in a network packet, is composed of the following fields:

- **Destination MAC address:** the MAC address of the device to which the frame is being sent.
- **Source MAC address:** the MAC address of the device that is sending the frame.
- **EtherType:** a field that indicates the type of upper protocol being used in the payload (e.g. IPv4, IPv6, etc.).
- **Payload:** the actual data being transmitted, which can be of variable length.

- **Frame Check Sequence (FCS):** a checksum used to detect errors in the frame.

Before these Ethernet fields there is also a preamble, which is used to synchronize the sender and receiver before the actual data transmission begins. In Table 2.1 we can see the most common EtherTypes and their respective meanings.

Protocol	EtherType
IPv4 (Internet Protocol version 4)	0x0800
IPv6 (Internet Protocol version 6)	0x86DD
ARP (Address Resolution Protocol)	0x0806
WOL (Wake-On-Lan)	0x0842
RARP (Reverse Address Resolution Protocol)	0x8035
VLAN (Virtual Local Area Network)	0x8100
LLDP (Link Layer Discovery Protocol)	0x88CC

Table 2.1: Common Internet protocols and their respective EtherType

A technology that can be built on top of Ethernet is **VLAN (Virtual Local Area Network)**, which allows for the segmentation of a physical network into multiple logical networks, allowing for better management and security. VLANs are identified by a unique identifier (VLAN ID) and can be used to separate traffic between different departments or groups within an organization. The Ethernet and VLAN headers are shown in Figure 2.1.

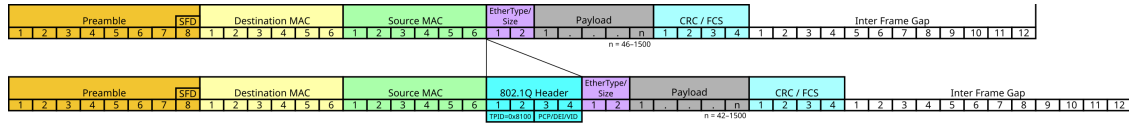


Figure 2.1: Ethernet and VLAN headers structure

Source: Bill Stafford

It is worth to spend a few more words on VLANs, as they are a key technology for our interests. A Virtual LAN is a logical domain (network) built on top of a physical network: a network device (switch) that supports VLANs can isolate traffic between different VLANs by using the VID bits in the 802.1Q header, thus allowing for the creation of multiple logical networks on the same physical infrastructure. This has great benefits in terms of security and performance, as it allows both for *traffic isolation* and QoS delivery within a single *trunk*. It is a technology widely employed in modern networks, as it allows for efficient scaling of the physical infrastructure. Moreover, VLANs can be defined in software, so greatly increasing the flexibility in network design. We will see that isolation and segmentation are key concepts that are largely deployed in the most diverse contexts in networking.

On the other hand, Wi-Fi is the most widely used protocol for wireless networks. As with Ethernet, many different versions of Wi-Fi coexist, with different speeds and capabilities, such as 802.11n (up to 600 Mbps), 802.11ac (up to 3.5 Gbps), and 802.11ax (Wi-Fi 6, up to 9.6 Gbps). Wi-Fi uses radio waves to transmit data, and it is based on the IEEE 802.11

family of standards. The Wi-Fi header is similar to the Ethernet header, but it also includes additional fields for managing the wireless connection: moreover, usually it is encapsulated in an Ethernet frame, as it is often used in conjunction with Ethernet networks.

It can be deployed both in *infrastructure mode*, where a central Access Point (AP) is used to connect devices to the network, and in *ad-hoc mode*, where devices communicate directly with each other without the need for a central AP.

Internet Layer

The Internet Layer is the third layer of the Internet Protocol Suite, responsible for **routing** data across the Internet. Routing is the process of forwarding data packets from one network to another, ensuring that they reach their destination. This layer is also responsible for addressing, which is the process of assigning identifiers to devices on the network: in this layer, this is done through the use of IP addresses (32 bits for IPv4 and 128 bits for IPv6). This is the first layer to introduce the concept of *router*, a device that connects different networks and forwards packets between them, and thus is also the first layer which operates through different networks; at the same time, it is the last layer that is implemented hop-by-hop, meaning that each packet is processed by each router it traverses until it reaches its final destination.

As probably anticipated, the most widely used protocol in this layer is **IP (Internet Protocol)**, an unreliable (in terms of delivery guarantees: it is upper layers' duty to address communication reliability, if needed), connectionless (it does not need to establish a connection before sending data) communication protocol that provides the basic mechanism for addressing and routing packets across the Internet. IP is responsible for encapsulating data from the upper layers into packets, which are then sent down to the Link Layer. The IPv4 header is composed of the following fields (shown in Figure 2.2):

- **Version:** the version of the IP protocol being used (IPv4 or IPv6).
- **Header Length:** the length of the IP header in bytes.
- **Differentiated Services Code Point (DSCP):** a field that indicates the priority and quality of service for the packet.
- **Explicit Congestion Notification (ECN):** a field that indicates ECN-capability and/or congestion in the network.
- **Total Length:** the total length of the packet, including the header and payload.
- **Identification:** a unique identifier for the packet, used for reassembly in case of fragmentation.
- **Flags:** a set of flags that indicate whether the packet can be fragmented or not.
- **Fragment Offset:** the offset of the fragment in the original packet, used for reassembly.
- **Time to Live (TTL):** a field that indicates how many hops the packet can take before it is discarded.

- **Protocol:** a field that indicates the upper-layer protocol being used (e.g. TCP, UDP, etc.).
- **Header Checksum:** a checksum used to detect errors in the header.
- **Source IP Address:** the IP address of the device that is sending the packet.
- **Destination IP Address:** the IP address of the device that is receiving the packet.
- **Options:** optional fields that can be used to provide additional information about the packet.

Together, DSCP and ECN form the so-called **Traffic Class** field.

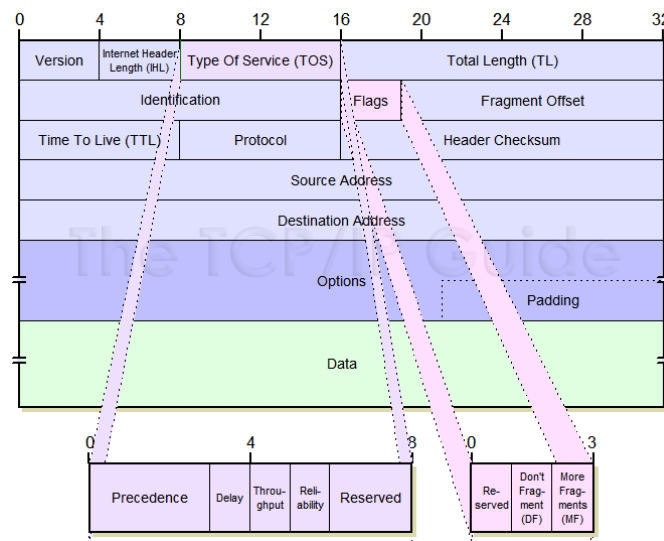


Figure 2.2: IPv4 header structure

Source: *The TCP/IP Guide*

Instead, in Table 2.2 we can see the most common Transport protocols and their respective IPv4 protocol numbers. This knowledge is useful for later stages of this thesis.

Protocol	IPv4 Protocol Number
TCP (Transmission Control Protocol)	0x06
UDP (User Datagram Protocol)	0x11
ICMP (Internet Control Message Protocol)	0x01
HIP (Host Identity Protocol)	0x8B
IP-in-IP (IP encapsulation)	0x04
DSR (Dynamic Source Routing)	0x30

Table 2.2: Common Transport protocols and their respective IPv4 protocol number

Transport Layer

The Transport Layer is the second layer of the Internet Protocol Suite, responsible for providing end-to-end communication between devices (it is the first layer to be implemented only on the end nodes of the communication).

It provides an abstraction, the commonly known **port**, which applications in the upper layers can “reserve” to access the network. A port is a 16-bit number that is used at this layer for multiplexing and demultiplexing data over different applications on the same device. Moreover, a certain range of ports is already reserved for specific applications, named *well-known ports*, that spans from 0 to 1023: Table 2.3 reports the most important ones. Note that a port number can be used simultaneously by different applications, if the underlying Transport protocol is different: for example, it is possible to run an application on port 5437 using TCP and another one on the same port using UDP, as they are different protocols and thus they are not in conflict with each other.

Application	Network port	Protocol
File Transfer Protocol (FTP)	21	TCP
Secure Shell (SSH)	22	TCP
Telnet	23	TCP
Simple Mail Transfer Protocol (SMTP)	25	TCP
Domain Name System (DNS)	53	TCP/UDP
HyperText Transfer Protocol (HTTP)	80	TCP/UDP
HTTP Secure (HTTPS)	443	TCP/UDP
Dynamic Host Configuration Protocol (DHCP)	67/68	UDP
Network Time Protocol (NTP)	123	UDP

Table 2.3: Most important *well-known ports*

The two main protocols used in this layer are **TCP (Transmission Control Protocol)** and **UDP (User Datagram Protocol)**.

TCP is a connection-oriented protocol that provides reliable, ordered, and error-checked delivery of data between applications. It is responsible for establishing a connection between two devices, ensuring that data is transmitted reliably and in the correct order, with a *at-most-once delivery* semantics. UDP, on the other hand, is a connectionless protocol that provides a simpler and faster way to send data between applications, but without the reliability and ordering guarantees of TCP, and operates with *best-effort delivery* similarly to IP.

TCP is widely used for applications that require reliable communication, such as web browsing, email, and file transfer. UDP is used for applications that require low latency and can tolerate some data loss, such as video streaming, online gaming, and voice over IP (VoIP).

The TCP header is composed of the following fields (shown in Figure 2.3):

- **Source Port:** the port number of the application that is sending the data.
- **Destination Port:** the port number of the application that is receiving the data.

- **Sequence Number:** a cumulative unique identifier for (the first byte of) each segment sent in the current communication session, used to ensure that data is received in the correct order.
- **Acknowledgment Number:** a number that indicates the next segment's sequence number that the sender expects to receive.
- **Data Offset:** the length of the TCP header in 4-bytes words.
- **Flags:** a set of flags that indicate the state of the connection (e.g. SYN, ACK, FIN, etc.).
- **Window Size:** a field that indicates the amount of data that the sender of the segment can (or wants) to receive.
- **Checksum:** a checksum used to detect errors in the header and payload.
- **Urgent Pointer:** an offset from the sequence number indicating the last urgent data byte.
- **Options:** optional fields that can be used to provide additional information about the connection.

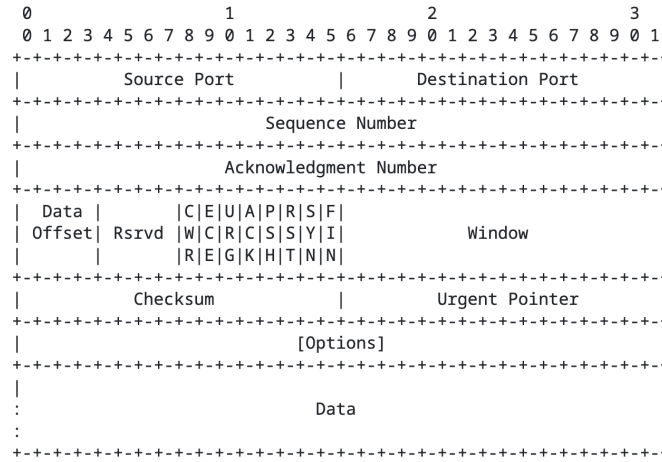


Figure 2.3: TCP header structure

In the context of TCP, the **three-way handshake** is a fundamental process used to establish a connection between two devices. It involves three steps:

1. The client sends a SYN (synchronize) packet to the server, indicating that it wants to establish a connection.
2. The server responds with a SYN-ACK (synchronize-acknowledge) packet, acknowledging the client's request and indicating that it is ready to establish a connection.

3. The client sends an ACK (acknowledge) packet to the server, confirming that the connection has been established.

This process ensures that both devices are ready to communicate and that the connection is established in a reliable manner.

In the same fashion, the **four-way handshake** is used to terminate a TCP connection. It involves four steps:

1. An endpoint sends a FIN (finish) packet to the other, indicating that it wants to terminate the connection.
2. The latter responds with an ACK packet, acknowledging the former's request.
3. The latter then sends a FIN packet to the former, indicating that it is ready to terminate the connection.
4. Finally, the former sends an ACK packet to the latter, confirming that the connection has been terminated.

The TCP header also includes a **window size** field, which is used to control the flow of data between the sender and receiver; it is dynamically adjusted during the data exchange to optimize the quantity of data exchanged and prevent congestion in the network. At the same time, it provides a mechanism for **congestion control**, which is used to prevent network congestion by adjusting the rate at which data is sent based on the current network conditions.

An extension to TCP and IP, thus a cross-layering example, is **ECN** (Explicit Congestion Notification), which allows routers to signal congestion to the sender without dropping packets. This is done by setting two specific bits, while routing across the Internet, in the IP header, which indicate that congestion has been detected and that the sender should reduce its transmission rate. Precisely, they are the two least significant bits of the Traffic Class field in the IPv4 header, which are encoded as follows:

- **00**: not ECN-capable.
- **01**: ECN-capable.
- **10**: ECN-capable.
- **11**: congestion experienced in a ECN-capable scenario.

It is needed to exploit cross-layering in this context, as congestion is handled only by the Transport layer (thus, in the end hosts), while it happens when traversing routers (which do not possess by design the knowledge of the Transport layer). In particular, the ECN mechanism is first established during the three-way handshake in the TCP header, then congestion signaling happens at the IP layer, where the two previously mentioned bits are set accordingly. This mechanism allows for more efficient use of network resources, as it reduces the need for packet retransmission and improves overall network performance. It will be one of the key features of our implementation.

The UDP header is much simpler, as it does not provide any level of reliability nor ordering, opposite to TCP. It is composed of the following fields (shown in Figure 2.4):

- **Source Port:** the port number of the application that is sending the data.
- **Destination Port:** the port number of the application that is receiving the data.
- **Length:** the length of the UDP header and payload in bytes octets.
- **Checksum:** a checksum used to detect errors in the header and payload.

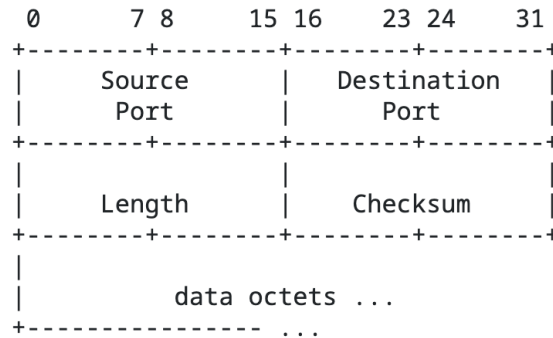


Figure 2.4: UDP header structure

Therefore, UDP is a simpler and faster protocol than TCP, but with a great compromise: it is built with the same delivery semantics as IP, best-effort. While this limitation can be overcome with applicative protocols, or can even be favourable in some cases (e.g. real-time applications or VPN tunneling), it is important to always carefully ponder the choice between TCP and UDP when designing network solutions and applications.

2.2 Queue management

As the Internet has grown in complexity, the need for resource management has become increasingly important. The number of users has vastly increased in the last few decades, and the amount of data being transmitted has grown exponentially. Bandwidth, processing power, memory, all became a scarce resource that needs to be managed efficiently. Resource management is the process of allocating and managing resources in a network to ensure that they are used efficiently and effectively, possibly giving each user the best possible experience according to their needs, and it is a key aspect of network design and operation.

We can define a **resource** as a hardware or software component used by an entity (e.g., a computer or a network) to perform operations. Resources can be limited in quantity, therefore there is the need to manage them efficiently. There are lots of examples of resources, but in the context of this thesis we will focus mainly on bandwidth and latency, as we are dealing first of all with network resources.

Resource management is a complex task, as it involves the coordination of multiple components and protocols, and it requires a deep understanding of the underlying network architecture and the applications running on it.

Our interests lie primarily in *queue management*. A **queue** is a data structure that stores, (usually) processes and finally releases elements (in our case, network packets) in a specific order, and is used to manage the flow of data. They are used in many different contexts, such as in operating systems, databases, and networking. In networking, queues are used to manage the flow of packets in a network device, such as a switch or a router.

There are, in general, two operations associated with such a data structure: **enqueue** and **dequeue**. Enqueue is the operation of adding an element to the queue, while dequeue is the operation of removing an element from the queue.

Usually, different queues in parallel are used to manage different types of traffic, such as latency-critical traffic or bandwidth-intensive flows.

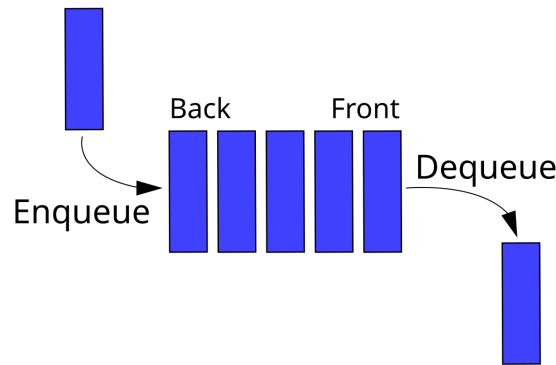


Figure 2.5: A generic queue with associated operations

2.2.1 Available algorithms and techniques

According to queueing theory, there are a variety (here, a non-exhaustive list) of disciplines that can be used for enqueueing and dequeueing:

- **FIFO (First In First Out):** the first element added to the queue is the first one to be removed. This is the most common discipline.
- **LIFO (Last In First Out):** the last element added to the queue is the first one to be removed. With such a dequeueing algorithm, the queue is also known as a stack.
- **Priority:** elements are removed from the queue based on their priority, with higher priority elements being removed before lower priority ones.
- **Equal sharing:** each element in the queue is given a fair/equal amount of capacity.
- **Shortest Job First:** the element with the shortest processing time is elaborated and removed from the queue first.
- **Shortest Remaining Time Job First:** the element with the shortest remaining processing time is elaborated and removed from the queue first.

While ideal queues have unlimited depth, real queues have a *maximum depth*: that is, the maximum number of elements they can store while processing the others. Filling the queue beyond this limit is a situation known as **queue overflow**, which can lead to various negative outcomes, in particular **congestion**.

Congestion is a scenario in which it is experienced a reduced Quality-of-Service due to the increased load of requests of resources: when there are too many competing actors for the same resources and the system cannot handle all of them, this leads, in the case of queues, to delays and loss of requests (in the network context, packets).

In the early days of TCP/IP, congestion control was mainly implemented at the Transport layer through an old TCP protocol version which, given that TCP specification identifies packet loss as a congestion signal, simply retransmitted lost packets: this led to a situation known as **congestion collapse**, where the network was unable to handle the increased load of retransmitted packets, leading to a further increase in congestion.

Therefore, there is the necessity to efficiently handle congestion and prevent it from happening at all: this is done through *congestion control*, and can be achieved in many different ways, apart from the obvious one, which is to increase the resources available in the system, sometimes not feasible due to constraints on costs or the physical environment:

- **Explicit Congestion Notification (ECN)**: as previously discussed, this is a mechanism that allows routers to signal congestion to the sender without dropping packets.
- **Active Queue Management (AQM)**: these are techniques that allow routers to manage queues more efficiently by dropping packets before the queue overflows, thus preventing congestion.
- **TCP built-in solutions**: newer TCP implementations have built-in mechanisms for congestion control, such as *slow start*, which allow it to adapt to changing network conditions.

In particular, AQM techniques are the most important within the context of this thesis, because, as we will see in Chapter 3, we are focusing on management inside the network equipment, thus we will either implement AQM mechanisms or exploit existing, known, protocol features to “signal” congestion to the senders.

A naive approach, not exploiting any of the below techniques, is to simply drop packets when the queue overflows: this is known as **tail drop** and it is the simplest form of AQM, but it is not efficient; whilst it was the main algorithm within early network devices, nowadays is widely discarded. Among the main, modern, AQM techniques there are:

- **Random Early Detection (RED)**: a technique that drops packets randomly before the queue overflows, thus preventing congestion.
- **Weighted Random Early Detection (WRED)**: an extension of RED that allows for different weights to be assigned to different types of traffic, thus allowing for better management of queues.
- **Robust Random Early Detection (RRED)**: yet another extension of RED that uses a more sophisticated algorithm to drop packets, thus preventing congestion.

- **Controlled Delay (CoDel):** a technique that aims to keep the queue length low by dropping packets when the queue length exceeds a certain threshold, thus preventing congestion.
- **Fair Queue Controlled Delay (FQ-CoDel):** an extension of CoDel that allows for fair sharing of resources among different flows, thus preventing congestion.
- **Proportional Integral controller Enhanced (PIE):** a technique that uses a proportional-integral controller to manage queues, thus preventing congestion.

Without any of them, **bufferbloat** can occur, which is the situation where the queue length becomes too long, leading to increased latency, reduced throughput and higher jitter (which is the variation in latency over time).

Let's focus on a couple of them, as they are the most relevant for our work.

Random Early Detection (RED)

The RED algorithm aims to prevent bufferbloat by dropping packets before the queue overflows, and does so by first monitoring the queue's occupancy and then, in case of probable congestion, starts dropping packets by the means of a statistical probability.

The algorithm is also shown in Figure 2.6.

The probability of dropping a packet is based on the average queue length (calculated on a moving average: therefore, this algorithm looks at the “recent” past to determine its parameters): the more it grows, the more likely is for the packet to be dropped before enqueueing. It utilizes, for this purpose, two values: min_{th} and max_{th} , which are respectfully the minimum and the maximum thresholds for the average queue length (avr); when avr is below min_{th} no packet is dropped, while when it is above max_{th} all packets are dropped: everything in the middle is handled with a linear probability function that increases the more avr approaches max_{th} . Values, and relative ranges, can be calculated either in bytes or in packets terms.

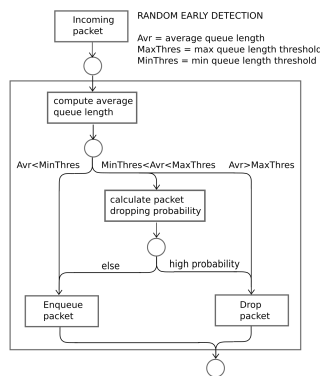


Figure 2.6: Random Early Detection algorithm

Source: helix84 (Wikipedia maintainer)

Because of the moving average discussed before, RED achieves a certain fairness, as it does allow burst traffic to pass through without be subject to dropping (within a certain margin), when compared against a simple tail drop algorithm.

RED, though, it is not exempt from limitations: first of all, it does not provide any Quality of Service guarantees, and then it results in a quite aggressive algorithm especially in only lightly loaded networks or in a quite conservative algorithm under heavy loaded networks.

Therefore, some amendments have been proposed to RED, one of them being **Weighted Random Early Detection (WRED)**: functionally it operates in the same way as RED, but maintains a set of weights, one for each class of traffic, thus allowing for earlier dropping of lower priority flows and greater tolerance for high priority ones.

Another extension is **Robust Random Early Detection (RRED)**, which aims to improve TCP performance when using classical RED, as it is particularly vulnerable to the so-called *Low-rate Denial-of-Service* attacks.

Finally, another major variation is **Adaptive Random Early Detection (ARED)**, which is a more sophisticated algorithm that adapts the dropping probability based on the variations on the moving average, thus allowing for better management of queues and preventing congestion: if the average oscillates around min_{th} then it becomes less aggressive, while if it oscillates around max_{th} then it becomes more aggressive.

Controlled Delay (CoDel)

In 2006 it has been demonstrated how RED uses a metric (average queue length in a sliding window) that poorly represents the actual status of the buffers, and that it cannot distinguish between a *good queue* (one which does not exhibit bufferbloat) and a *bad queue*.

Therefore, a new algorithm has been proposed, **Controlled Delay (CoDel)** ([19]), which aims also to address other critical points of RED. In particular, CoDel is parameterless and does not require any configuration (apart, optionally, from the delay threshold we want to achieve), as every parameter is determined, dynamically, locally in the network equipment itself ignoring statistics that would require a global knowledge of the network (such as round-trip time).

Its main metric is the minimum delay experience by packets in the queue in the time determined by a sliding window. Also, this parameter is, of course, calculable only at dequeueing time (therefore, no need for ingress algorithms which would only add complexity and delays to the system).

At each hop in the network (that adopts CoDel), the algorithm computes the sojourn time of the packet in the queue, and it stores this value only if it is lower than the current minimum for the current interval. At the end of the interval (by default, 100 milliseconds), the minimum sojourn time is compared with the threshold (by default, 5 milliseconds: the latency target we want to achieve), and if it is greater the packet is dropped and the interval is shortened (entering the so-called *drop state*), otherwise the packet is forwarded, the interval is set back to the default value and the loop starts again.

In case of subsequent interval reductions (due to packet not meeting the latency target), the sequence used to determine the new interval value is as follows:

$$T_{sojourn} = \frac{100}{\sqrt{n}}$$

where n is the number of packets that have been dropped since the last time the interval was reset (i.e. the number of times that the minimum sojourn time was greater than the threshold).

Later, **Fair Queue Controlled Delay (FQ-CoDel)** has been proposed, which is an extension of CoDel that allows for *flow queueing* so it can distinguish and manage fairly different flows: moreover, it gives the first packet of a flow priority so that smaller and quicker flows can be processed faster without wasting resources needed by larger flows.

It is worth noting that CoDel is very useful for *responsive* flows, such as TCP, which can adapt to the network conditions and reduce their transmission rate when congestion is detected. On the opposite, it is quite unefficient for *unresponsive* flows, such as UDP without management at the Application layer, which do not adapt to the network conditions and can lead to congestion.

2.3 Towards scalable mobile networks: 5G

What is described in Section 2.2 is particularly relevant in the context of mobility, as in mobile networks the resources are even more limited and the greater dynamicity experienced in this context determines a higher rate of problematic situations.

Nowadays and in the future, as stated before, the target is to connect everything to the Internet: medical appliances, smart cities, sensing, vehicles, and so on. Mobile networks, and in particular the latest generation, 5G, are designed to address these challenges and provide a scalable and efficient solution for connecting a large number of devices and applications.

But how is 5G designed, to be such a scalable solution?

2.3.1 Reference architecture

The 5G architecture is designed to be flexible and scalable, allowing for the integration of a wide range of devices and applications. It is composed of three main components: the **Radio Access Network (RAN)**, the **Core Network (CN)** and the **User Equipment (UE)**. It is shown in Figure 2.8.

It is designed to handle three main types of services, that are **Enhanced Mobile Broadband (eMBB)**, **Massive Machine Type Communications (mMTC)** and **Ultra-Reliable Low Latency Communications (URLLC)**. Table 2.4 shows the Key Performance Indicators (KPIs) for these services ([38]), which state the minimum expected performance for each of them in ideal situations. From this table, we can see that each of the three services have clearly different targets: eMBB aims for very large throughput, mMTC aims for a very large number of devices in a dense area to support massive IoT distributed applications, while URLLC aims for very low latency and high reliability as a basis for critical life-saving applications such as remote surgery or autonomous driving.

Moreover, 5G can be deployed in the most diverse scenarios, from urban to rural areas, accomodating nearly every use case and application.

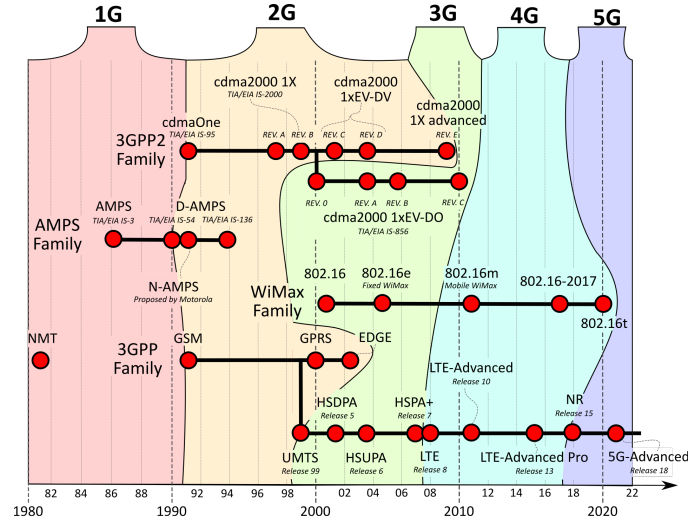


Figure 2.7: Timeline of mobile networks

Source: Michel Bakni

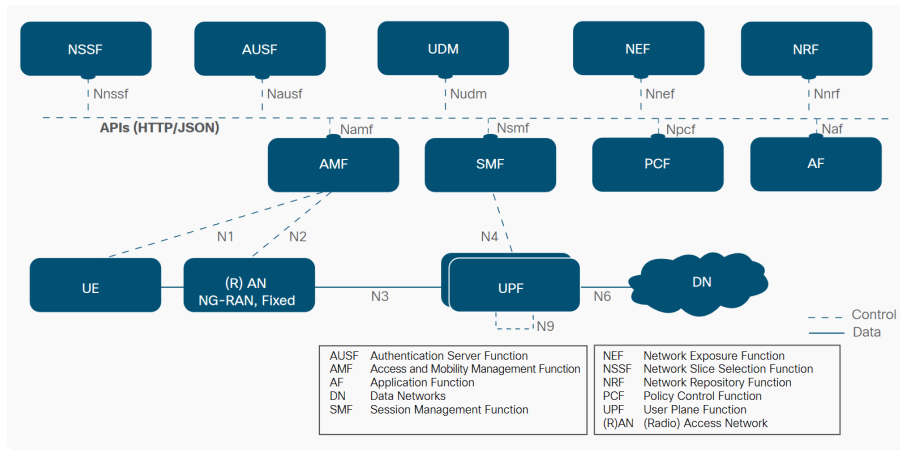


Figure 2.8: 5G architecture

Each physical deployment can be further virtualized and customized for specific needs: physical deployments themselves lend to different scenarios, as the broad range of frequencies reserved for this technology allows for different trade-offs between coverage, capacity and power, as well as different antenna configurations.

5G can also be deployed on top of an existing 4G infrastructure, for a gradual transition towards this new generation: this is called **Non-Standalone (NSA)** deployment, where the 5G RAN is connected to a 4G Core Network, while the **Standalone (SA)** deployment is the one where the whole infrastructure is 5G-ready, thus exploiting its full potential (see Figure 2.9).

Service	KPI	Value
eMBB	Peak Data Rate	20 Gbps (DL); 10 Gbps (UL)
eMBB	User Plane Latency	4 ms (DL, UL)
eMBB	Mobility	500 km/h
URLLC	User Plane Latency	0.5 ms (DL, UL)
URLLC	Reliability	99.999%
mMTC	Device density	1.000.000 devices/km ²
mMTC	Battery Life	> 10 years

Table 2.4: Key Performance Indicators for the different 5G services

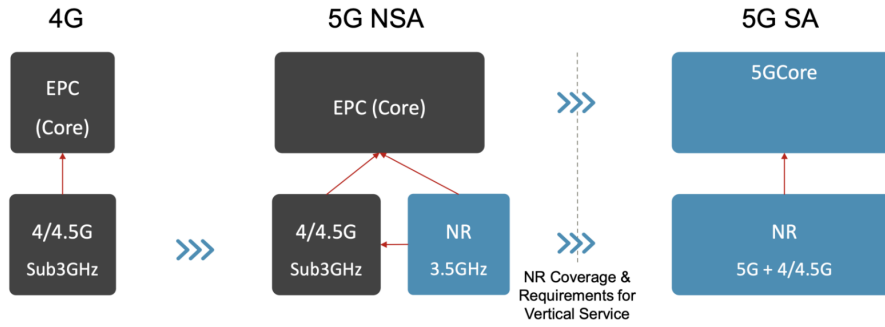


Figure 2.9: NSA and SA deployment of 5G

Let's now delve into the details of each component, to understand how they work together to provide what has been promised.

Core Network (CN)

It is the component that provides the main functionalities of the network and connects the UE (or, more specifically, the RAN) to the Internet. It follows what is called a **Service-Based Architecture (SBA)**, in which every network component, in previous generations known as *Network Entities* and now called **Network Functions (NFs)**, cooperate with each other and integrates with different layers of the architecture (e.g. RAN and UE) to provide the required service. NFs use a common interface to intercommunicate, called **Service-Based Interface (SBI)**, which is based on open RESTful APIs, thus allowing for an interoperable architecture, with both backward and forward compatibility.

It has a strong focus on containerization and virtualization, letting the Core to be deployed in a cloud-native environment. It can be deployed on-premises, in a private cloud or in a public cloud, near or far from the edge, thus giving different trade-offs in terms of latency, bandwidth and reliability: this remains a critical choice for a 5G deployment.

The most important NFs in the 5GC are:

- **Access and Mobility Management Function (AMF)**: it is responsible for managing the access to the network and authenticating UEs. It uses the N1 interface to communicate with the UE and the N2 interface to communicate with the RAN.

- **Session Management Function (SMF):** it is responsible for managing the sessions between the UEs and the Core, including the allocation of IP addresses and the management of Quality of Service (QoS) parameters.
- **User Plane Function (UPF):** it is responsible for routing, inspection, policy-enforcing, reporting and QoS-handling of packets, and it is the only NF that interacts with the Data Plane. It is located between the RAN and the Internet, through interfaces N3 and N6, while N4 connects this NF to the SMF.
- **Policy Control Function (PCF):** it is responsible for managing the policies of the network, including the QoS parameters, while supporting network behaviour control.
- **Network Repository Function (NRF):** it is responsible for providing a registry for currently available NFs, like a DNS.
- **Network Exposure Function (NEF):** it is responsible for exposing the network capabilities to third-party applications, allowing them to interact securely with the network.
- **Unified Data Management (UDM):** it is responsible for managing the user data, including the subscription data and the authentication data.
- **Network Slice Selection Function (NSSF):** it is responsible for selecting the appropriate Network Slice (more on this in 2.3.3) for the UE, based on the service requirements and the available resources.
- **Application Function (AF):** it is responsible for handling service-specific aspects, and application-related policies.
- **Authentication Server Function (AUSF):** it intercepts AMF's authentication requests and, interacting with the UDM, determines if the actual authentication is successful or not.
- **Location Management Function (LMF):** it is responsible for managing the location of the UEs, allowing for location-based services and applications.
- **Network Data Analytics Function (NWDAF):** it is responsible for collecting and analyzing data from the various NFs.

There are different important implementations of the 5G Core, such as **Open5GS** and **Free5GC**, which are open-source projects that provide a complete implementation of the 5G Core Network, including all the NFs and the interfaces between them. They are widely used in academia and industry for testing and research purposes.

Radio Access Network (RAN)

The RAN is the component that connects the UE to the Core Network, and it is responsible for providing the radio interface to the UE; it is deployed within a gNodeB. It is composed of three main parts: the **Baseband Unit (BU)** (responsible for digital signal processing),

the **Radio Unit (RU)** (whose job is to convert digital information into radio signals and viceversa) and an **antenna** (transmitting radio signal into electric waves).

Throughout the generations, the RAN has evolved from a centralized, physical, single unit first into a distributed architecture, then into a virtualized architecture, and finally in what is known as **Disaggregated Open RAN (O-RAN)** (Figure 2.10).

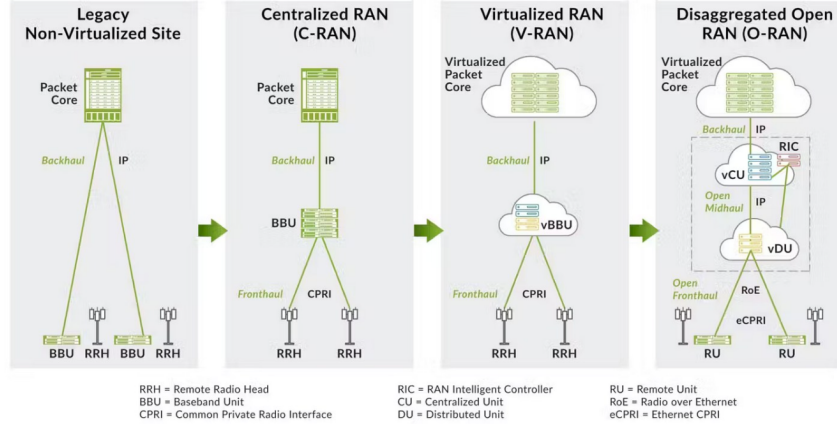


Figure 2.10: RAN architecture evolution

The gNodeB is divided itself into two parts: the **Central Unit (CU)** and the **Distributed Unit (DU)**, connected by an interface called **F1**. The CU is a logical node that controls the different DUs connected to it and manages various different functions that are not already handled by the DUs themselves, such as radio access network sharing, mobility management, and positioning. The position into where split the CU and DU is not fixed and can be chosen based on the deployment scenario: therefore, sometimes the DU handles more functions than the CU and viceversa.

The *O-RAN Alliance* is an industry consortium that aims to promote the adoption of open standards and interfaces in the RAN, allowing for greater flexibility and interoperability between different vendors and technologies: its solution is known as **O-RAN**, and it promotes the idea of a scalable, disaggregated, virtualizable, and open RAN architecture, shown in Figure 2.11. It is not of interest for our thesis, therefore we will not focus on the details. A typical deployment is instead shown in Figure 2.12.

The Core is connected to the RAN and the UE through various interfaces, in particular the **N1** interface (between AMF and UE), the **N2** interface (between AMF and RAN) and the **N4** interface (between SMF and UPF). Interfaces N6 and N9 are used to connect the UPF to the Internet and to other NFs, respectively.

In particular, for the scope of this thesis, we are interested in the **N3** interface, also called **Transport Network (TN)**, which serves to connect the RAN to the UPF, thus allowing the user data packets to be forwarded between the UE and the Internet. Inside of this particular link, a protocol called **GPRS Tunneling Protocol (GTP)** is used to encapsulate the user data packets and forward them between the RAN and the UPF: built on top of UDP and IP protocols, it is needed to provide a tunneling mechanism for General Packet Radio Service (GPRS) within the 5G architecture.

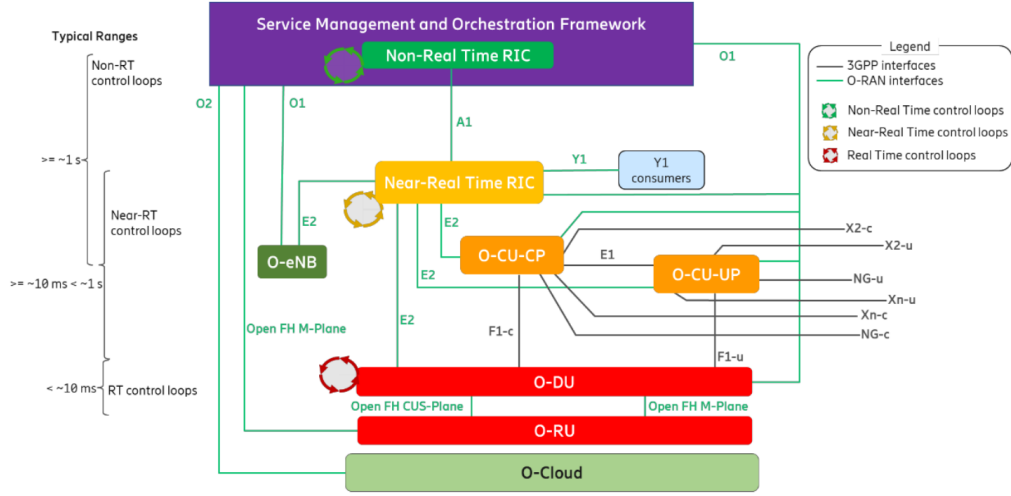


Figure 2.11: O-RAN general architecture

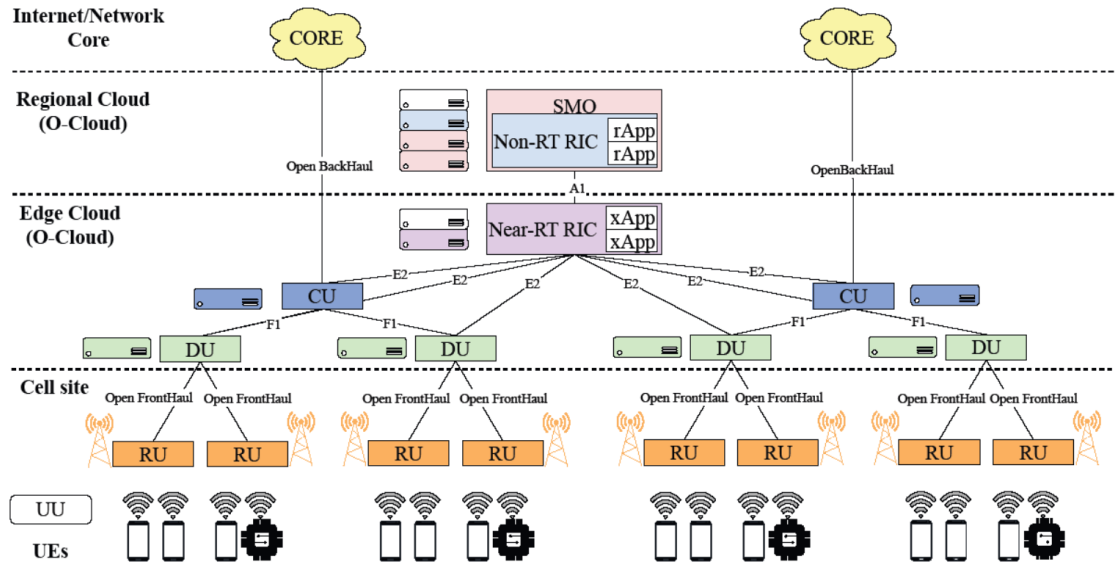


Figure 2.12: O-RAN typical deployment

User Equipment (UE)

The User Equipment (UE) is the device, belonging to the end-user, that connects to the network. It can be a smartphone, a tablet, a laptop, or any other device that is capable of mobile broadband connectivity, ranging from Commercially-available Off-The-Shelf (COTS) devices to professional high-end equipment.

The UE is responsible for finding and attaching to the nearest gNodeB: it corresponds, within the GSM architecture, to the Mobile Station.

It needs to handle different aspects of the mobile connectivity, such as:

- **Mobility management:** tracking where the device itself is, and realizing handoff.
- **Call control:** giving base support for voice calls, as well as tearing them down.
- **Identity management:** keeping the personal identity securely within the device, and giving the appropriate credentials to authenticate with the provider's infrastructure.

2.3.2 Network programmability

The Internet Protocol Suite has evolved over the years, with new protocols being added to address the changing needs of the Internet. As the Internet has grown in complexity, the need for programmability has become increasingly important. This is particularly true in the context of mobile networks, where the number of devices and applications is constantly increasing, and the need for flexibility and scalability is paramount. Network programmability allows for the development of custom protocols and applications that can be tailored to specific needs, but also for adaptation of networks to changing requirements, such as the need for increased bandwidth, lower latency, or improved security.

An approach that has gained popularity in recent years is the use of **Software-Defined Networking (SDN)**, which distinguishes between two orthogonal and complementary planes: the **Control Plane (CP)** and the **Data Plane (DP)**. The Control Plane is responsible for making decisions about how data should be forwarded, while the Data Plane is responsible for actually forwarding the data. This separation of concerns allows for greater flexibility in network design, as the Data Plane can be programmed to implement custom, ad-hoc, protocols, while the Control Plane can be used to rule the network in an always-changing environment, possibly leveraging methods of Machine Learning (ML) and Artificial Intelligence (AI). The Data Plane is therefore composed of network devices, such as switches and routers, that are responsible for forwarding data packets based on the rules defined by the Control Plane.

A general overview of the SDN architecture is shown in Figure 2.13.

Delving into the current panorama of Software-Defined Networking, we can find several tools and technologies that are widely used in the field.

Some of them are protocols, such as **OpenFlow**, which allows the Control Plane to insert rules into an already-defined Data Plane through an open specification adopted by many vendors and open-source projects.

Other tools are software switches, such as **Open vSwitch**, which is a production-quality, multilayer virtual switch designed to enable network automation through programmatic extensions. Integrated in many different platforms, it is also widely used in cloud computing. It supports a large variety of telecommunication protocols, Quality of Service mechanisms, kernel and user-space implementations for packet forwarding, and it is also compatible with OpenFlow. Anyway, it lacks the ability to define custom protocols, as it is not programmable.

Therefore, a language that has been developed to address this limitation is **P4**, which allows for the complete programmability of the Data Plane, enabling the definition of custom protocols and packet processing pipelines. More on this language, which is the main focus of this thesis, will be discussed widely in Section 2.4 and Appendix A.

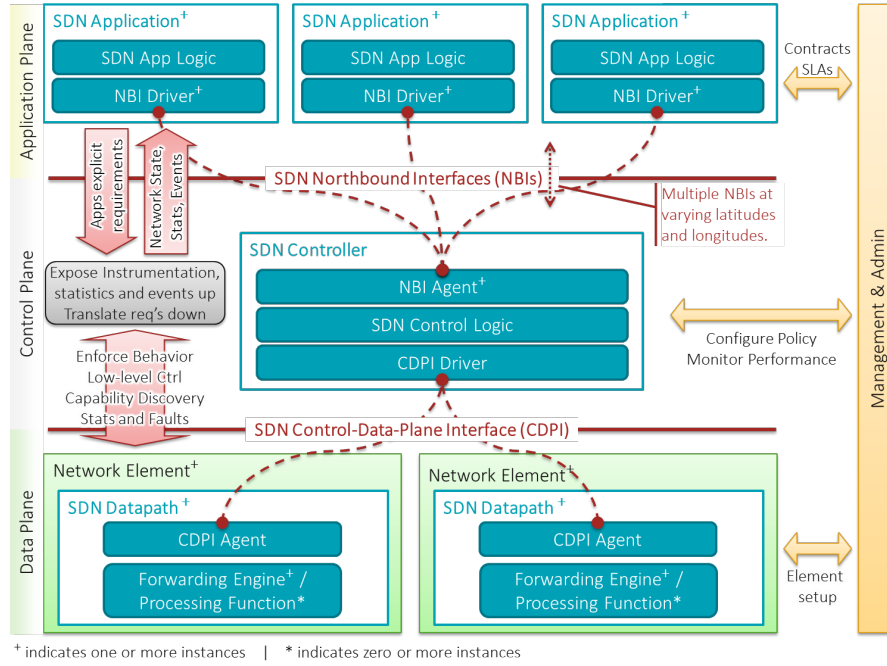


Figure 2.13: Software-Defined Networking architecture

Source: Open Networking Foundation (ONF)

Finally, there are also several controllers that can be used to manage the network, such as **Open Daylight** and **ONOS**, which are open-source projects that provide a platform for building SDN applications and managing the network infrastructure. Both of them excel in different areas, as they are designed for different use cases, but are widely used in academia and industry. They provide a set of APIs through which it is possible to interact with network devices in a protocol-agnostic way, exposing an easier way to interact with multiple and heterogeneous devices in the same network. They both scale well to large networks, but in particular ONOS is designed to be deployed in telecom, carrier-grade, mission-critical environments, with more features for high availability, horizontal scaling, and others.

2.3.3 Network Slicing

Among the most important features of 5G is **Network Slicing (NS)**, which allows for the creation of multiple virtual networks on top of a single physical infrastructure. The idea behind it is to provide a flexible and scalable method to allocate resources for various purposes, thus isolating various services and applications from the others belonging to different slices.

The final product of slicing is a **Network Slice**, an end-to-end logical isolated and virtualized network. It is tightly coupled with Quality-of-Service, as each slice can have its own set of QoS parameters and can deliver an optimal experience for specific kinds of

traffic. Its structure is defined by the **Network Slice Template (NST)** ([30]), which is a blueprint that describes the slice's characteristics, such as the required resources, the supported services, the QoS parameters, and the network functions that are needed to implement it.

To realize this, 5G leverages the concepts of **Network Function Virtualization (NFV)** and **Software-Defined Networking (SDN)**, which have been thoroughly explored in Section 2.3.2.

In the real world, it is often used also by **Mobile Network Operators (MNOs)** within commercial agreements with **Mobile Virtual Network Operators (MVNOs)**, to give them a virtualized and customized access to the underlying physical infrastructure, which can then further tailored by the MVNO itself according to its needs. Of course, the process can then be repeated from MVNOs, and multiple slices can be created and deployed for end customers.

With a strong implementative focus in the Core, recently efforts have been made to extend the Network Slicing concept to the RAN, thus supporting the creation of end-to-end isolated networks across the whole infrastructure.

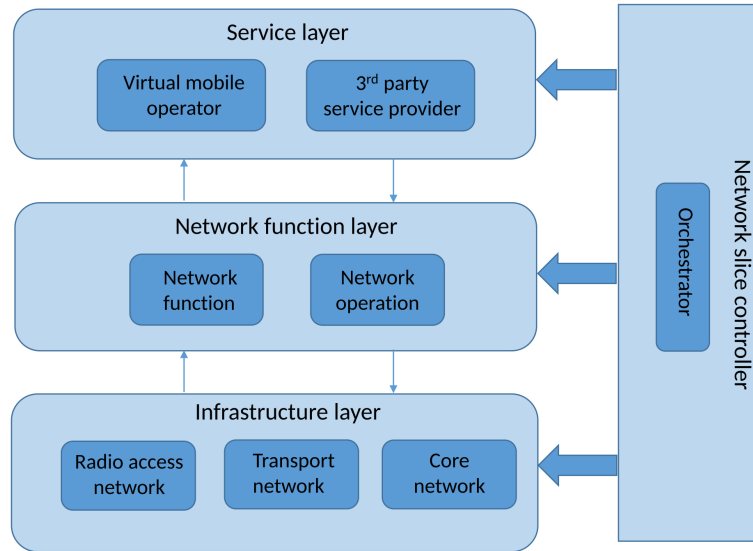


Figure 2.14: Network Slicing architecture

The overall architecture of Network Slicing is shown in Figure 2.14: as already explained, it makes use of the SDN architecture, as it splits the Network Slice Controller (the Control Plane) from the data forwarding implementation. The latter is structured in a *multi-tier architecture* with three layers:

- **Service layer:** it is the interface through which every user of the infrastructure interacts with. Each service is represented as an instance that embodies every requirement in the form of a *Service Level Agreement (SLA)*.
- **Network Function layer:** here, the Network Slices are effectively created, based on

the requirements given by the upper layer. Composed by multiple and well-defined network functions, at slice creation time these functions are chained together to create the behavior desired. The Network Operations serves as elements for configuring the slices.

- **Infrastructure layer:** it is the layer that provides the physical resources needed to implement the Network Slices. It is composed of the physical network devices, such as switches and routers, and the devices giving connectivity, such as base stations.

The Network Slice Controller, instead, is the component that orchestrates the whole lifecycle of the network, overseeing different aspects of the management.

By design Network Slices are isolated from each other (each slice's performance must not affect other slices' performance): this decision comes with huge impacts on the security, as attacks and faults in one slice cannot propagate to the others, thus limiting the attack surface of the infrastructure.

Figure 2.15 shows a visual representation of the Network Slicing concept, where each slice is represented as a separate network with its own set of resources and functions.

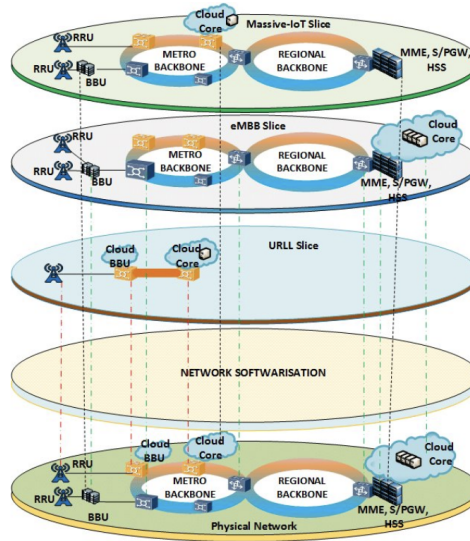


Figure 2.15: Network Slicing visualization

In the Core network, slicing can be implemented in two ways: either all Core functions are instantiated once for each slice, or some are shared among multiple slices, to restrain the resource usage and the complexity of communication (i.e. authentication or mobility). Usually, the UPF (which belongs to the user plane) is private to each slice, while the AMF and the NSSF are shared.

2.4 A Primer on P4

Programming Protocol-Independent Packet Processors (P4) is a high-level, strongly-typed, domain-specific (DSL) programming language designed for programming the Data Plane of network devices.

It enables line-rate packet processing and, most importantly, it can be used on top of different targets, ranging from simple software switches to ad-hoc, specialized, hardware devices (provided, of course, that the target supports the P4 language). Freeing the ties from specific protocols, P4 permits to even define new, custom, network protocols, or completely reinvent the whole networking stack.

As already stated, a P4 deployment is target-specific, meaning that the same P4 program is **not** portable across different targets (i.e. different hardware or software switches): therefore, the P4 program must be tailored to the specific target adopted, even though usually some features are common across them. Figure 2.16 shows the process of writing, compiling and deploying a P4 program onto a P4 target: after having the knowledge about the specific device you are dealing with, it is needed to write a program that exploits the (software or hardware) features available and then, through a vendor-distributed compiler, compile the P4 program into a representation accepted by the final target. The actual deployment of the compiled behaviour inside the device remains target-specific. For example, the Simple Switch (gRPC) accepts a JSON file as input, which describes the target's behaviour, given by the `p4c` compiler used with a P4 program.

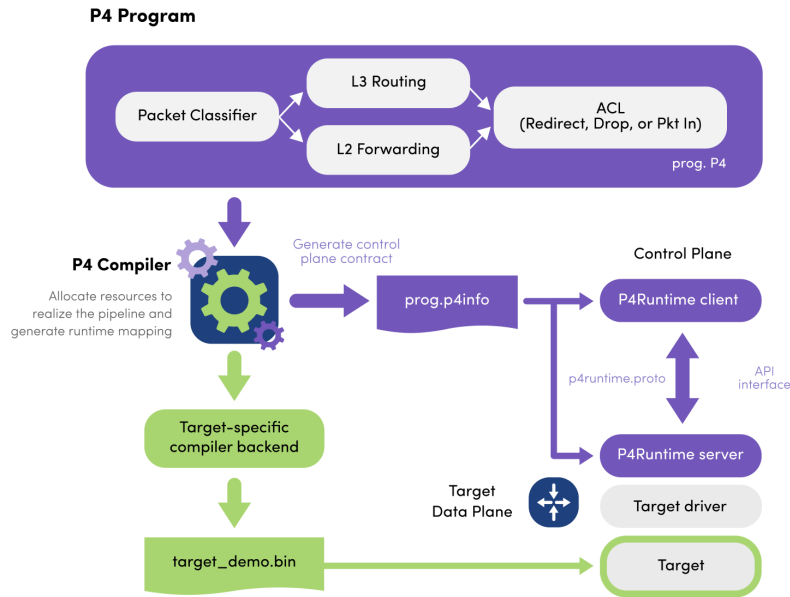


Figure 2.16: P4 deployment process

Source: *P4.org*

A more detailed panorama on the P4 language is available in Appendix A.

2.4.1 Main P4 constructs

According to the P4 specification ([21]), there are a handful of constructs that are used within a P4 program and should be available across every target: they are the main abstractions used to define the behaviour of the data plane, and as of May 2021 they are:

- **Header types:** through these it is possible to define the structure of the headers the packets will be carrying and will be parsed by the parser, in terms of fields it is composed by (each one expressed as a sequence of bits).
- **Parsers:** they define the permitted sequence of headers within an incoming packet, including how to read and extract them.
- **Match-Action Tables (MATs):** these are data structures against which packets are matched through (user-defined and CP-filled) lookup keys, and associated actions are executed (if the rules defined by the CP match, otherwise a default action is chosen).
- **Externs:** special constructs specific to a particular architecture which can be manipulated by the P4 program through a well-defined API. Almost always the internal behaviour is a black box (or, at least, not P4-programmable).
- **User-defined metadata:** previously named also custom metadata, this is a user-defined data structure associated with each packet.
- **Intrinsic metadata:** metadata provided by the architecture associated with each packet.

Parser

A P4 parser can be seen as a **Finite-State Machine (FSM)** that defines the sequence of headers to be parsed from an incoming packet, and how to extract them. It is composed of a set of states, each one representing a specific point in the parsing process, and a set of *transitions* between states, which are triggered by various factors (usually the presence of a specific header or by the absence of it, or by a value inside of the headers themselves).

The first state is the *start* state, and the parsing goes on until the *accept* or *reject* states are reached, which are the last state of this FSM.

Header types have a hidden bit inside of them, which is known as the **validity bit**: it can be set to 0 (not valid) or 1 (valid) by the programmer, but when reading a header through the **extract** operation within the parser, it is automatically set to 1. This bit is also employed by the deparser, which will only emit the headers whose validity bit is set to 1. Other operations also leverage the use of this validity bit, but we will not go into details here as they are not relevant for the purposes of this thesis.

Transitions can be conditional, through the use of **select** statements, or unconditional, through the use of **transition** statements. Conditional transitions are usually used, for example, when distinguishing between different protocols, such as IPv4 or IPv6 in the Ethernet header.

Another operation, useful for error handling, is the **verify** function, which accepts two parameters: the first one is a boolean, which is the condition to be verified, and the second

one is an exception to be raised if the condition is not met.

For extraction, the P4 core library provides a special extern called **packet_in** (completely different from the PacketIn port mentioned before), instantiated automatically by the architecture for each incoming packet. It exposes a set of methods to read the packet's contents:

- **extract**: it is used to read the specified header from the packet and to set its validity bit of the specific header to 1.
- **lookahead**: it provides a way to evaluate the next contents of the packet without extracting them and without advancing the internal pointer of the reader.
- **advance**: it is used to advance the internal pointer of the reader, so that the next read will be from later bits of the packet. Therefore, it allows skipping bits and headers that are not needed.

On the opposite side, the deparser uses another extern, called **packet_out**, through which it is possible to **emit** the headers previously parsed, extracted and processed.

Match-Action Tables

Match-Action Tables (MATs) are the main data structure used to implement the logic of a P4 program, based on a (or multiple) lookup key(s) and an action associated with it. Within a control block it is possible to define a set of MATs, each one with its own set of keys to match against with various criteria. The criterias define how the value inside the packet must match with the one specified by the key (i.e. the Protocol field inside the Ethernet header, or the IPv4 source address, etc.):

- **Exact match**: the values must match exactly with the one defined in the table.
- **Longest Prefix Match (LPM)**: the key value can match with the value having the longest common prefix defined in the table.
- **Ternary match**: the packet's value can match with a specific pattern specified by a value and a mask (in some way, it is a generalization of LPM).
- **Range match**: the value must be within a specific range defined in the table. This one is defined only for the v1model architecture, not by the P4 core library.

Every line inside a MAT is composed of the set of values of the keys to match against, and the action to be executed if the match is successful. In case of miss, there is the possibility to define a default action. The actions can be defined through the Control Plane dynamically or be statically defined inside the P4 program. The structure and the flow of execution of a MAT is shown in Figure 2.17.

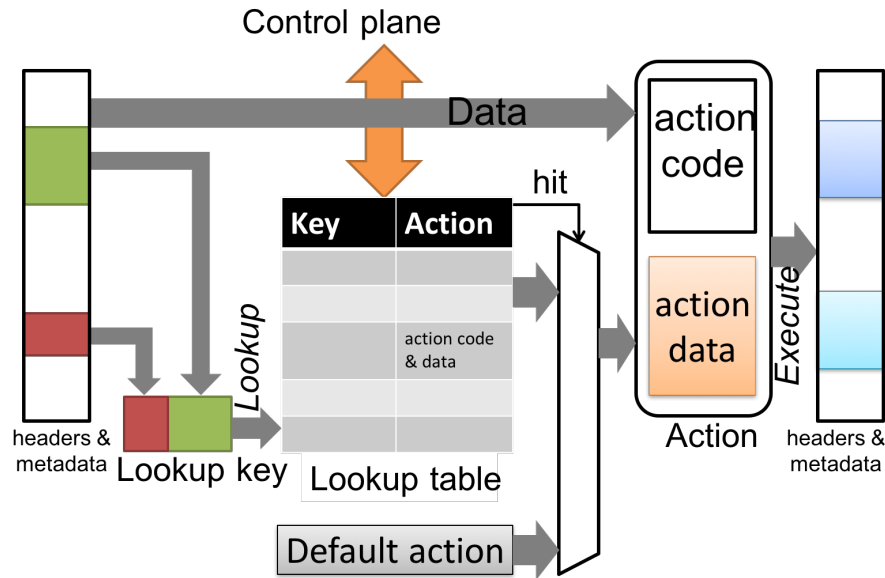


Figure 2.17: MAT structure and packet matching

Source: *P4.org*

Externs

Externs are special constructs that can be used to extend the functionality of a P4 program, providing access to specific hardware features or to external resources. They are not part of the P4 core language, but they are defined by the architecture and can be used to implement custom behaviors that are not directly supported by the P4 language. Externs can be used to implement a wide range of functionalities, such as, in the case of the v1model architecture:

- **Counters:** they are used to count, in the most general term. They can be used to implement traffic monitoring and statistics collection.
- **Direct counters:** quite similar to the previous ones, but automatically count the number of matches for each entry in a table. Therefore, direct counters are associated with a table by design.
- **Meters:** they are used to measure the rate of incoming traffic and to implement rate limiting and traffic shaping. They are based on a token bucket algorithm called **two rate Three Color Marker (trTCM)**, defined in [14].
- **Direct meters:** similar to direct counters, they are associated with a table and automatically measure the “rate of matching” for each entry in the table.
- **Registers:** they are used to store and retrieve data in a way that is not directly supported by the P4 language, allowing for more complex data structures and algorithms but, most importantly, maintain state among different packets.

- **Hash functions:** they are used to compute a hash value for a specific input.
- **Random number generators:** they are used to generate random numbers, which can be used for various purposes, such as load balancing or for implementing custom algorithms.

Other externs provided by the v1model architecture are the operations of cloning, resubmission, and checksum computation/verification, and in general the others implementing the special paths and architecture-specific operations. Other externs (mainly operations) are also exposed by the architecture, but are not of interest for this thesis.

Metadata

Metadata in P4 is a way to associate additional information with packets, which can be used to store temporary data or to pass information from a block to another. There are two types of metadata in P4:

- **User-defined metadata:** these are defined by the programmer and can be used to store any information needed by the specific application needs. They are defined as a data structure, and can be used to store information such as flags, values or any other data that needs to be associated with a packet.
- **Intrinsic metadata:** these are provided by the architecture and are automatically associated with each packet. They can be used to store information such as the input port, the output port, or any other information that is relevant for the packet processing. The values of intrinsic metadata are usually supplied automatically by the architecture as the packet traverses the pipeline. Within the v1model, the data structure is defined in the type `standard_metadata_t` (see Table 2.5).

Within the `standard_metadata_t` structure there are some timestamp fields: these presuppose the existence of a timer or a clock inside the architecture. The Simple Switch (gRPC) has a built-in clock, *relative to the start of the switch*, with the granularity of one microsecond, which is used to fill the timestamp values.

2.5 Related Work

The topics discussed so far are trending strongly in research and are a priority part of it: lots of efforts go in the direction of improving the current solutions available in the fields of congestion control, resource management, programmable networks and Network Slicing.

Congestion control has been, since the early days of the Internet, a hot topic in research, as it is a fundamental aspect of network performance. It allows avoiding congestion collapse and bufferbloat, two phenomena that can severely degrade or void network capabilities. In particular, the focus is on improving the performance of AQM algorithms, such as RED, CoDel, PIE, and respective variants, as well as on developing new algorithms that can better adapt to the changing network conditions and provide better performance in terms

Field name	Type	Description
ingress_port	bit<9>	Source port number
egress_spec	bit<9>	Assigned destination port number
egress_port	bit<9>	Final destination port number
instance_type	bit<32>	> 0 $\iff$ going through special paths
packet_length	bit<32>	Packet length in bytes
enq_timestamp	bit<32>	Timestamp at enqueue in the TM
enq_qdepth	bit<19>	Queue length at enqueue in the TM
deq_timedelta	bit<32>	Time passed inside the TM at dequeue
deq_qdepth	bit<19>	Queue length at dequeue from the TM
ingress_global_timestamp	bit<48>	Timestamp the packet reached Ingress
egress_global_timestamp	bit<48>	Timestamp the packet reached Egress
mcast_grp	bit<16>	Multicast group ID
egress_rid	bit<16>	Replication ID for multicast
checksum_error	bit<1>	Checksum verification errors occurred
parser_error	error	Error raised by the Parser, if any
priority	bit<3>	Priority of the packet

Table 2.5: Structure of `standard_metadata_t` in the v1model architecture

of latency, throughput and reliability, sustaining the increasing demand for bandwidth and low-latency applications and future-proofing in the face of new protocols and technologies.

Many versions of RED exist, such as ARED, WRED, RRED and many others, each one with its strengths and weaknesses. In [42] another variant of RED called Classified Enhanced RED (CERED), designed for Wireless Sensor Networks, is presented.

Machine Learning (ML) techniques are being increasingly used to improve congestion control algorithms, as they can learn from the network conditions and adapt to them in real-time, thus providing better performance and reducing the need for manual configuration and tuning. In [34] a survey of the state of the art of ML-based AQM algorithms is presented, highlighting the potential of these techniques and providing multiple deployment scenarios and methodologies.

Even within 5G networks, congestion control is a hot topic, especially inside the RAN, where the limited resources must be managed and shared among hundreds of users. In [32] and [33], ML techniques are deployed in a disaggregated RAN, where current AQM algorithms are not sufficiently performant.

Software-Defined Networking can be too a valuable tool to improve congestion control, as it allows for flexible management of the network. The feasibility of integration between the SDN architecture and AQM algorithms is explored in [37], where OVS is used to deploy different AQM algorithms dynamically to assist the network in an industrial automation scenario, with different traffic patterns and priorities.

Regarding P4 as a language that enables the development of SDN platforms, there are various works that explore its potential in this context. It can be used for fine-tuning current protocols even in a non-SDN scenario, such as improving TCP fairness [11], or to implement dynamic AQM algorithms: [16] proposes the usage of Strict Priority queueing coupled with RED and CoDel, while [7] advocates the use of Deep Reinforcement Learning and **In-band Network Telemetry (INT)** to maximise the usage of the network with respect to the QoS parameters desired. INT itself is of particular interest in research when it comes to P4, as [41] and [44] demonstrate.

This language is widely employed also for implementing tighter security measures and policies: [26] analyzes what flow table attacks are and how they can be mitigated in the P4 ecosystem. DDoS mitigation is another important threat to nowadays network infrastructures: [47] proposes a solution based on P4 to detect DDoS attacks in real-time.

Instead, [45] explores the use of P4 to implement line-speed Machine Learning algorithms, leveraging the Control Plane for the training phase. Another useful application of P4 can be the integration with NFV, as shown in [25], where the authors propose a solution to dynamically offload complex operations to Virtualized Network Function (VNF) instead of relying on the P4-defined Data Plane. P4 can enable also collaborative large-scale widely geographically-distributed Machine Learning models training, as [40] shows.

We can thus see how P4 is a powerful and extremely versatile tool that is used to implement a variety of heterogeneous solutions, ranging across almost every aspect of networking.

Network Slicing is a very hot topic in research, as it is a key feature of 5G networks and it is expected to be widely adopted in the future. [20], [46], [1], [9], [17] and [43] analyze the impact of Network Slicing within 5G networks, exploring the state of the art and the future trends in this field, while proposing possible solutions for the currently known

challenges. Security is one of the most important aspects in NS, therefore [8] highlights different security issues and addresses them with a set of solutions.

As explained beforehand, P4 is a powerful tool that can be used to implement Network Slicing at the Transport level (N3 link), and we are interested, in this thesis, in exploring possible implementations of it.

In [5], the authors advance a P4-based framework for bandwidth management for different types of traffic, called P4-TINS, then deployed on a software switch. A similar approach is taken by [4]. [39] and [31] exploit rerouting of packets, with a SDN controller on a P4 dataplane and an INT-based protocol to monitor the network performance, thus requiring a full path of P4-enabled switches with similar capabilities. Generative AI has seen an interest surge in the last few years, and it is being employed in the most diverse topics of research: [13] explores the use of Generative AI within the Control Plane of a P4-enabled network to populate the P4 tables. As widely known, P4 supports also hardware-based targets, such as FPGAs or NPUs: [28] and [27] propose a hardware-based solution for QoS delivery in 5G networks. More specifically within the Transport link, [36], [18] and [3] analyze the effects on applications when defining a custom Transport layer suitable for Network Slicing and QoS delivery. Within tactile networks, where latency is a critical issue to deal with, [35] proposes a P4-based framework to tackle this problematic. P4 can then be suitable also for optimizing data-centers' and industrial's networks, as shown by [24], which cover in particular the topic of service chaining, and [10].

General frameworks are also proposed, such as [12], which provides a P4-based framework for implementing *Virtual Queues* to manage more efficiently the resources in a device: this provides a way to implement a software-based queue monitoring solution even when the hardware counterpart does not support it or does not exist.

Finally, other general uses of P4 are covered by [23], which explores the use of P4 in different scenarios, such as network monitoring, traffic engineering, and security, highlighting the versatility and the possible issues that can arise.

The current state of the publications in the scientific literature for this particular topic shows what has been discussed in Chapter 1: the lack of an actual Network Slicing (providing true isolation) P4 solution for the Transport Network which can represent a Proof-of-Concept for a wider-scale industrial adoption, and of a modular Control Plane designed for heterogeneous networks.

Recalling the knowledge gathered by previous Sections and the State of the Art, we are working on the N3 link of the 5G infrastructure, that is the connection designed to handle and carry user data packets forth and back between the RAN (Radio Access Network) and the Core network (in particular, the User Plane Functions). Usually composed by traditional, hardware, switches and routers, our aim is to provide a framework capable of introducing P4 devices inside an already-existing network without requiring immediate and huge financial investments, but gradual and lighter ones.

Chapter 3

Proposed framework

This chapter describes the proposed framework for Network Slicing in programmable data planes. We will start with a general overview of the architecture proposed, highlighting the main components and their roles and some design choices made. Then, we will delve into the details of the Control Plane, explaining the Orchestrator and the Agents, their structure and their API. Finally, we will describe the Data Plane, focusing on the P4 programming language and the Simple Switch gRPC target used in this work.

3.1 Framework architecture

In order to overcome the limitations highlighted in the state-of-the-art (SOTA) analysis, this work introduces a novel framework grounded in the principles of Software-Defined Networking (SDN). The proposed solution is articulated around a dual-plane architecture, consisting of a Data Plane and a Control Plane, thereby enabling a clean functional separation between packet forwarding and network management. This separation is essential to achieve programmability, scalability, and adaptability in heterogeneous networking environments.

The Data Plane (DP) is realized using the P4 programming language, which has emerged as a de facto standard for programmable packet processing. P4 provides the capability to define customized packet parsing, matching, and forwarding behaviors independently of underlying hardware constraints. This programmability enables the network to evolve alongside emerging requirements, avoiding the rigidity inherent in traditional fixed-function devices. Furthermore, the deployment of P4 programs directly on hardware switches ensures performance at line rate, thus reconciling programmability with efficiency. Within the proposed framework, the Simple Switch gRPC target is employed as the DP backend. This component plays a crucial role not only in executing forwarding policies but also in mitigating the isolation problem by integrating traffic management functionalities and having an architecture suited to solving the problem. Such mechanisms guarantee resource separation between network slices, thereby ensuring that the performance of one slice remains unaffected by the behavior of others.

While the DP addresses forwarding and isolation, the challenges of slice management and the coordination of multiple P4-enabled devices necessitate a carefully designed Control

Plane (CP). To this end, the CP is deliberately decoupled from the DP, in accordance with SDN principles, and can be instantiated under different deployment paradigms, such as centralized, distributed or hybrid.

It is organized into two complementary entities:

- The **Orchestrator**, which serves as the global decision-making authority. It is responsible for resource allocation, policy enforcement, and high-level coordination across network slices, ensuring consistency and compliance with system-wide objectives.
- The **Agent(s)**, which operate in close proximity to the P4 devices. They act as intermediaries between the Orchestrator and the Data Plane, translating abstract control policies into device-specific configurations and reporting local state information back to the Orchestrator.

Therefore, the choice made is to have a hybrid architecture, which combines the strengths of both centralized and distributed approaches. The centralization of the Orchestrator ensures the presence of a global and coherent view of the network, providing a single point of control and, also, failure. On the other hand the Agents, being distributed across the network, allow for independent deployment and management of the P4 devices, thus enabling scalability and flexibility in the network management. The deployment of the Agent should be dependent on the Orchestrator, as it is the component that provides the necessary information to the Agent to operate correctly. On the other side, the Orchestrator should not be dependent on the Agents, their deployment and their topology: therefore, the Agents will need to *register* themselves at the Orchestrator, following a *push* communication model. This means that it is the Agent that will send a registration command, while the Orchestrator will **not** wait for the Agents and operate independently from them.

This Orchestrator-Agent model provides a flexible yet robust mechanism for managing the lifecycle of slices, distributing policies, and ensuring system scalability. By leveraging the strengths of both planes, the proposed framework addresses the shortcomings identified in existing SOTA approaches, offering a programmable and manageable networking environment suitable for modern multi-tenant and slice-based infrastructures.

To provide a Network Slicing solution, it is mandatory to define what a slice is and, eventually, how it is composed. While a Network Slice is defined as a set of resources (bandwidth, latency, etc.) that are allocated to a specific application or service, we define a Subslice as a subset of a Network Slice that is allocated to a specific user or group of users. In particular, the set of subslices a slice is composed of should not exceed the resources allocated to the slice itself, even when considering the worst-case scenario (i.e. all subslices are active at the same time). As the enforcement of policies and rules is centralized within the Orchestrator, it is this component that is responsible for checking the validity of the slices and subslices at their creation and their management, and at their deletion as well.

But how is traffic classified into slices and subslices? In our case, we made the following choice:

- **Slice classification:** the slice is assigned based on the VLAN identifier (VID) of the packet, which is extracted by the Parser, through an exact match. If a packet does not belong to a VLAN, it is assigned Slice 0. This choice is made as we suppose,

within our scenario, that every slice is associated with a specific VLAN, and thus the VID is a good classifier for the slice.

- **Subslice classification:** the subslice instead, is based on the destination IP address of the packet (IPv4 header), which is also extracted by the Parser, through a ternary match, and the Slice ID just assigned in an exact match. Recalling our scenario, we suppose that there are different UPFs for different applicative flows, each one with an associated IP address. Therefore, the destination IP address outside the GTP header (i.e. the UPF IP address) is a good classifier for the subslice. The default action is to assign subslice 0.

Each Network Slice has some Quality-of-Service parameters associated with it, which are the maximum and minimum bandwidth and the maximum latency: these are, in concrete terms, the resources allocated to a slice. Subslices associated with a slice can have different parameters, with the rule that the maximum bandwidth and the maximum latency of a subslice cannot exceed the maximum bandwidth and the maximum latency of the slice itself.

In Figure 3.1 the general architecture of the framework is shown.

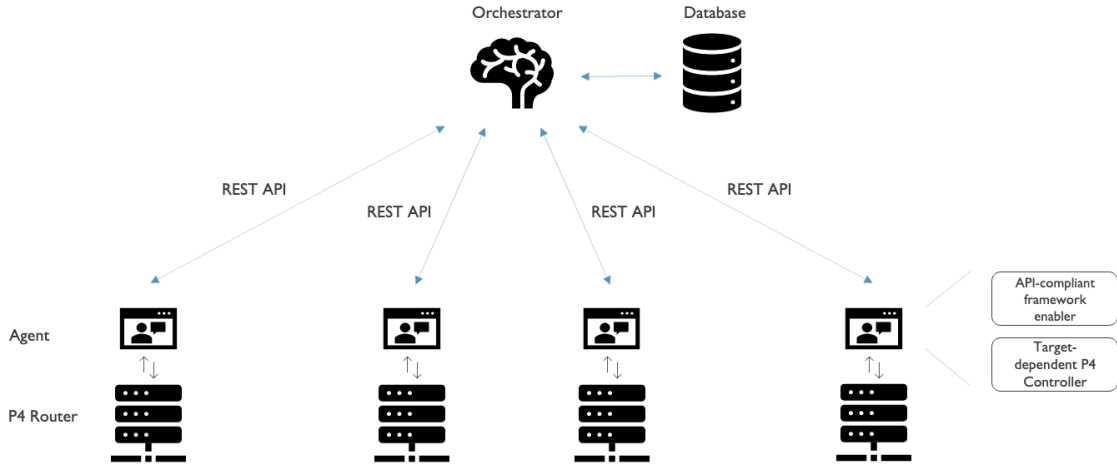


Figure 3.1: General architecture of the proposed framework

3.2 Control Plane

As anticipated, the Control Plane is designed to be able to interact with different and heterogeneous Data Planes, to provide an easy way to extend the framework to different hardware and software solutions and therefore facilitate the gradual adoption of a P4-based infrastructure for Network Slicing. Moreover, another goal of the Control Plane is to provide an unified interface to monitor, control and manage the Data Planes (i.e. the P4 devices) and the Network Slices, thus providing a single point of interaction for the users and the administrators of the network.

Mainly implemented with the Python programming language, the **Orchestrator** is responsible for providing the unified interface and communicating with the various **Agents** deployed on top of or near the P4 devices (depending on their architecture and capabilities). The Agents, on the other hand, are responsible for interfacing with the Orchestrator (through a well-defined REST API) and with the Data Plane and thus dealing with the low-level details of the P4 devices' implementation.

It is important to note that the Orchestrator module is directly connected to a knowledge base (as per Figure 3.1), in our case a Relational Database Management System (RDBMS), which is used to store the Network Slices, their parameters, the Agents and their configuration, along with the telemetry data collected from the various Data Planes. This allows for a persistent storage of the data, which can be later used for analysis, debugging and monitoring purposes.

A second design choice is to decouple the Agent internal structure into two main components: an **API-compliant framework enabler** and a **target-dependent P4 controller**. The former is responsible for providing the API to the Orchestrator, registering to it at startup and receiving the configuration and the commands to execute from it. The latter, instead, converts the API calls into low-level P4 commands, which are then sent to the Data Plane through, in the case of the Simple Switch (gRPC), the P4Runtime API.

Therefore, the only component that has actual knowledge of the Data Plane is the P4 controller, and is the only part that is target-dependent in this architecture: this allows for a complete decoupling of the communication between the Orchestrator and the Agents from the actual implementation of the Data Plane, thus facilitating the deployment of the framework as the API-compliant framework enabler is the same for every Agent, regardless of the underlying Data Plane implementation. On the opposite side, the P4 controller needs to be redeployed and re-implemented for every different target, as there is the need to adapt their low-level protocol and their structure and capabilities.

3.2.1 Orchestrator structure

The Orchestrator is the central component of the Control Plane architecture: it is the main and unique point of entry for the external requests and the “brain” upon which an intelligence or automation component can build. Its main objective is to expose an API for the management of the network, independent from the underlying topology and technologies used. Moreover, it has to interface with the deployed known Agents, from whom it receives *registration requests* (the Orchestrator, therefore, keeps in its local memory a table with currently registered Agents and their IP). The interface between the Orchestrator and the Agents' API-compliant framework enabler is defined by another REST API. The implementation of this component has been made through the Python **Flask** framework, through which it is possible to rapidly implement REST webservers.

Whenever the Orchestrator goes down (exploiting the capabilities of the `atexit` Python package), the slices are deleted from every registered Agent as there is no way to further manage them, and thus the network is restored to a pristine state.

In Table 3.1 the API endpoints for the Orchestrator are shown.

Endpoint	Method	Source	Response codes
/agents/register	PUT	Agents	200, 204
/agents/deregister	DELETE	Agents	200, 404
/agents/telemetry	POST	Agents	200
/slices	GET	External	200
/slices/<int>	PUT	External	200, 400
/slices/<int>	DELETE	External	200, 400, 404
/subslices/<int>/<int>	PUT	External	200, 400, 404
/subslices/<int>/<int>	DELETE	External	200, 400, 404
/slices/client/<int>	PUT	External	200, 204, 400, 404
/slices/client/<int>	DELETE	External	200, 404
/subslices/client	PUT	External	200, 400
/subslices/client	DELETE	External	200, 400, 404

Table 3.1: Orchestrator API endpoints

In particular, there are three endpoints used by the Agents, which are used respectfully for self-registration and self-deregistration at the Orchestrator and sending telemetry information. The telemetry information has to respect a predefined common structure, as we will see briefly. The body of the HTTP request, for Agent (de)registration, must be, JSON-encoded, `{'port': int}`.

The response codes in Table 3.1 agree with the HTTP standard, thus it is possible to refer to that to understand the meaning of the responses given by the API endpoints. The same will stay for the Agent.

The others endpoints are designed to be exposed towards the external network, thus accepting requests in terms of slice retrieval (`/slices`, empty HTTP body), creation (`PUT /slices/<int>`, where `<int>` is (and will be) the slice number desired) and deletion (`DELETE /slices/<int>`). Respectively, the same logic applies for the subslices' endpoints. The HTTP body for (sub)slice creation and deletion shall be `{'id': int, 'max_bandwidth': int, 'min_bandwidth': int, 'latency': int}`; when designing the API for clients and subclients (that are, respectfully, the keys in the MATs for slice and subslice assignment, i.e. VLAN IDs and IPv4 addresses), for practical reasons, has been slightly differentiated: in particular, while in the case of VLANs assignment the ID is embedded into the URL, in the case of IPv4 addresses the IP address itself is sent inside the HTTP body. Therefore:

- In the case of VLAN assignments, the body shall be `{'slice_id': int}` for assignment while empty for deassignment.
- Otherwise, `{'slice_id': int, 'subslice_id': int, 'dstAddr': String}` for assignment, and is the same for deassignment.

This ends the overview on the HTTP REST API exposed by the Orchestrator, and now the only missing point is the management of persistence. First of all: why is it needed? The need for persistence arises as it is (extremely) helpful to save, for later analysis, benchmarking or troubleshooting, telemetry reports sent by the various Agents deployed on the Transport network.

When a telemetry packet is received at the Orchestrator level it is marked with its source IP address (remember that the telemetry packet is sent through the HTTP REST API just explained): this is sufficient in the majority of the cases, as it is improbable a deployment where the Orchestrator is (geographically or topologically) far from the Agents, but problems *may* arise under Network Address Translation (NAT). After having been marked the packet is sent to the Database Management System (DBMS) of choice, which is in our case PostgreSQL. Every field of the JSON object, representing the telemetry information within the HTTP request, is copied into the homonymous column of a dedicated DB table. This table is structured with many different fields, some mandatory independently from the underlying P4 device (described in Table 3.2) and some instead device-specific therefore optional (`NULLABLE` columns). For optimization reasons, every time the Orchestrator goes down the table is truncated, as it is assumed that the telemetry data is not needed anymore, and thus it is not persisted. This is done to avoid having a too large table, which would make the queries slower and more expensive in terms of resources.

Field	Type	Description
protocol	int	Telemetry report reason (as of Table 3.10)
payload	int	Bandwidth used (in KB)
slice_id	int	Slice ID (as per the P4 program)
subslice_id	int	Subslice ID (as per the P4 program)
inserted_at	timestamp with TZ	Timestamp of insertion

Table 3.2: Mandatory columns for telemetry reporting

In the same database, but in different tables, the Network Slices and their parameters are stored, and also the Agents and their configuration.

The persistence management is done through the `psycpg2` Python package, and the credentials are stored and read thanks to the `dotenv` package, which allows to read environment variables from a file (of course, as it is holding sensitive credentials, it should be properly secured; more secure ways to hold credentials do exist, but for the sake of easiness and fast prototyping this choice has been made).

3.2.2 Agent structure

Within the Control Plane architecture proposed, the Agent is the component that interfaces with the Data Plane and provides the low-level control and management of the P4 devices. It is responsible for receiving the commands from the Orchestrator, translating them into P4Runtime (or other protocols accepted by the specific device) commands and sending them to the Data Plane. Moreover, it is responsible for receiving telemetry reports from the Data Plane and sending them to the Orchestrator, as well as for registering and deregistering itself at the Orchestrator. The Agent is implemented in Python, and in particular with the `Flask` framework, to provide a REST API for the communication with the Orchestrator.

It is divided into two main components, an API-compliant **framework enabler** and a target-dependent **P4 controller**. The framework enabler is responsible for managing the API and for interfacing with the Orchestrator, while the P4 controller is responsible for

managing the Data Plane and for sending the commands to the P4 device it is attached to.

Table 3.3 shows the API endpoints exposed by the Agent’s framework enabler: in this case, every endpoint is designed to be used only by the Orchestrator, and thus they should not be exposed to the external network.

Endpoint	Method	Response codes
/slices/<int>/<int>	PUT	200, 400
/slices/<int>/<int>	DELETE	200, 400
/slices/client	PUT	200, 204, 400, 404
/slices/client	DELETE	200, 404
/subslices/client	PUT	200, 400
/subslices/client	DELETE	200, 400, 404

Table 3.3: Agent API endpoints

The first glaring difference with the Orchestrator API is that the Agent does not expose an endpoint for subslice creation, as it is not needed: the same API endpoint is used for both slice and subslice creation, as at the Agent level a subslice is simply considered a slice with a subslice ID different from 0; the same applies for deletion. The HTTP body for these requests is the same as the one for the Orchestrator, which means (still using JSON encoding) `{'id': int, 'max_bandwidth': int, 'min_bandwidth': int, 'max_latency': int}`.

Instead, the endpoints for client and subclient management are *not* the same as the Orchestrator’s ones, as they are respectively:

- For slice assignment (PUT /slices/client), `{'slice_id': int, 'vlan_id': int}`, and the same for deassignment.
- For subslice assignment (PUT /subslices/client), `{'slice_id': int, 'subslice_id': int, 'dstAddr': String}`, and the same for deassignment.

A particular design choice is that the Agent, at startup, does not know the Network Slices and their parameters, and thus it has to request them to the Orchestrator through the GET /slices endpoint: this is done automatically at registration time.

There is no discovery process associated with the Agents, as they are supplied with the Orchestrator’s IP address at startup, through a command-line argument.

Still at startup, a daemon thread is started, which is responsible for gathering telemetry reports from the lower layer of the Agent and sending them to the Orchestrator through the previously-defined API endpoint. Every 5 seconds, the thread tries to read at most 50 packets from the device buffer: these values are hardcoded and can be subject to change in future work, though they should not impact the performance of the system as they are.

Delving now inside the P4 controller, it should be recalled that this is the component that interfaces with the Data Plane, translating the API calls from the framework enabler into the specific protocol commands. The explanation that will follow, of course, is specific for the Simple Switch (gRPC) target, as it is the one used in this work. However, the P4

controller is designed to be an interface through which the framework enabler can interact with the Data Plane, and thus it exposes some operations that must be implemented for every P4 device:

- `P4Controller(switch_address: String, port: int)`: this is the constructor of the P4 controller, which takes as arguments the address of the P4 device and the port to use for the connection. It is responsible for initializing the connection to the P4 device and for setting up the necessary data structures to manage the P4Runtime (or specific protocol) API calls.
- `add_table_rule(table_name: String, keys: Dict, action: String, params: Dict, priority: int=0)`: this method is responsible for adding a rule to a table in the P4 device. It takes as arguments the name of the table, the keys to match on, the action to execute and its parameters, and an optional priority value (in case of ternary or LPM matches). If the priority is not specified, it defaults to 0 (which is, for non-LPM-nor-ternary matches, the only admissible value).
- `delete_table_rule(table_name: String, keys: Dict, action: String, params: Dict, priority: int=0)`: as before, this method has the same parameters but is used to delete a rule from a table.
- `update_table_rule(table_name: String, keys: Dict, action: String, params: Dict, priority: int=0)`: as before, this method has the same parameters but is used to update a rule in a table.
- `set_clone_session(session_id: int, replicas: Tuple|List<Tuple>)`: this method is responsible for setting up a clone session with the specified ID and the replicas that should be created for it. The replicas are specified as a list of tuples, where each tuple contains the egress port and a Replica ID.
- `install_pipeline(p4info: FilePath, binary: FilePath)`: this method is responsible for installing the P4 pipeline on the P4 device. It takes as arguments the path to the P4Info file and the path to the binary file generated by the P4 compiler. The first one is used to understand the structure of the P4 program and the format of the messages that can be exchanged, while the second one contains the actual implementation of the program. In the case of the Simple Switch (gRPC), the binary is a JSON file.
- `manage_slice(slice_id: int, subslice_id: int, max_bandwidth: int, min_bandwidth: int, max_latency: int, delete_mode: Boolean=False)`: this method is responsible for managing a Network Slice, either creating or deleting it. If `delete_mode` is set to `True`, the slice is deleted, otherwise it is created with the specified parameters. The parameters are the same as those used in the Orchestrator API.
- `manage_client(slice_id: int, vlan_id: int, delete_mode: Boolean=False)`: this method is responsible for managing a client, either creating or deleting it. If `delete_mode` is set to `True`, the client is deleted, otherwise it is created with the specified parameters. The parameters are the same as those used in the Orchestrator API.

- `manage_subclient(slice_id: int, subslice_id: int, dstAddr: String, delete_mode: Boolean=False)`: this method is responsible for managing a subclient, either creating or deleting it. If `delete_mode` is set to `True`, the subclient is deleted, otherwise it is created with the specified parameters. The parameters are the same as those used in the Orchestrator API.
- `get_packets(number: int, p4info_metadata: FilePath)`: this method is responsible for getting a number of packets from the P4 device. It takes as arguments the number of packets to get and the path to the P4Info file, which is used also to understand the structure of the packets. The packets are returned as a list of `Packet` objects, where each `Packet` contains the fields of the packet within a dictionary.

The latency specified through the API call, in the case of the Simple Switch (gRPC) target, is converted into a priority number to be used within the P4 program (in particular, to set the `standard_metadata.priority` field), as the Simple Switch does not support latency-based scheduling. The priority is calculated on a range-basis, checking the value of the latency and the occupancy of the queues inside the Traffic Manager, and reserving the most appropriate free queue.

3.3 The Simple Switch gRPC

To facilitate development and testing, the prototype presented in this Chapter will be based on a P4-enabled *software* switch called **Simple Switch gRPC**, which is the reference implementation of the P4 language. It is based on what is known as the **BMv2** framework, which lets end developers to implement their own P4-based targets (known examples are the Simple Switch, the Simple Switch gRPC, and the PSA Switch). These switches, when started, attach to the interfaces specified and assigns a port number to each of them, which are the ones used to refer to the interfaces within the P4 program: moreover, two more virtual ports can be defined, that are the **PacketIn** and the **PacketOut** ports, which are used to respectfully send and receive packets to and from the Control Plane (in other words, the names are made with respect to the CP).

The Simple Switch is designed to be fairly easy to use and deploy, and it is widely used in the P4 community for testing and development purposes. It exposes a set of two APIs through which it is possible to interact with the switch:

- **P4Runtime**: this is the main API used to interact with the switch, allowing to program the data plane and to retrieve information about the switch's state. It is a gRPC-based API that provides a way to control the switch's behavior at runtime. It is not available in the Simple Switch (but only in this gRPC version), even though it is the flagship reference API for P4 devices.
- **Simple Switch API**: based on the Thrift protocol, it is used to retrieve particular statistics and to configure some components. This same API is also exposed by the Simple Switch.

3.3.1 Architecture

This software switch is based on the **v1model** architecture, it presents six programmable blocks, plus a non-P4-programmable **Traffic Manager** (configurable only through the Thrift-based API). Figure 3.2 shows the v1model as it is, with each programmable block. We will now delve into the details of each block, highlighting their functionalities: the effective implementation of the proposed framework will be further explored in Section 3.4.

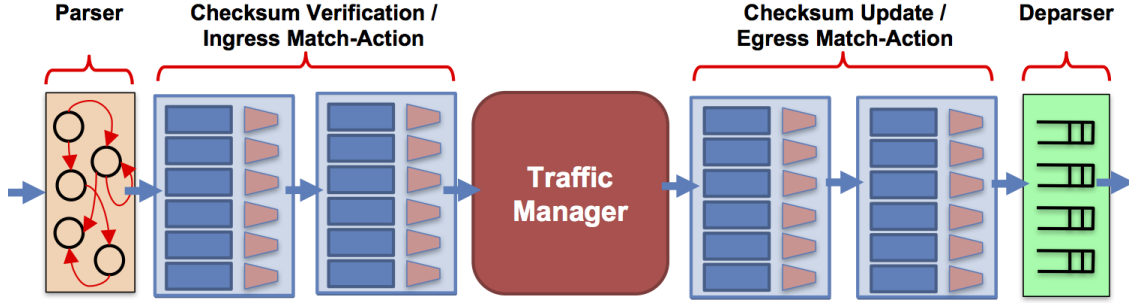


Figure 3.2: The v1model architecture

Architectures describe the *packet processing pipeline* of the switch, therefore the way in which packets are processed as they traverse the switch: the order in which they are processed is not defineable by the user. In particular, within the v1model we have (in order):

- **Parser:** the first programmable block is responsible for parsing the incoming packets, extracting the relevant headers according to the logic here defined. There can be conditional jumps between headers, and it can also handle malformed packets (which can be then dropped, or sent to a specific port, or whatever is preferable). The headers which remains unparsed are ignored and not processed further.
- **Checksum verification:** this block aims to verify the checksums of the headers parsed by the previous block, to ensure packet's validity. If a checksum is not valid, the packet is usually dropped.
- **Ingress:** here usually happens the main logic of the switch, where the packet is thoroughly processed, altered, marked, or anything else is needed. The routing decisions must be made here by choosing the output port for the packet. Moreover, replication is also possible in this block, allowing the packet to be duplicated and sent to multiple ports or handled differently.
- **Traffic Manager:** not a programmable block, is composed of a single set of FIFO queues per output port, which are used to store packets before they are sent for egress processing. The Traffic Manager can be configured to have multiple queues per port, and being them managed with a Strict Priority algorithm, its purpose can be also that of implementing packet scheduling and prioritization.
- **Egress:** another computation-heavy block, after routing has been set it is possible to further process the packet, and replicate it if needed. The packet can be dropped here, too.

- **Checksum computation:** here, after all the processing is done, the checksum of each header is recomputed to ensure validity after the modifications made in the previous blocks.
- **Deparser:** in the last programmable block, the headers previously parsed are re-emitted towards the destination interface (output port), in a custom-defined order.

Note that the Ingress and the Egress blocks are what is usually referred to as *control* blocks (i.e. blocks where the main logic and computation of the program happens).

The one defined conceptually above is the normal path packets go through, but there are also some special paths (some mentioned) that can be exploited to handle specific needs. Some of them happen after ingress processing, and others after egress processing. The former ones are:

- **Ingress-to-Egress (I2E) Cloning:** if the packet is marked for cloning, at the end of ingress processing the **Packet Replication Engine (PRE)** is cloned; the original packet continues its path through the switch as nothing happened, whilst the cloned packet is sent back to the Parser and then directly sent to the Traffic Manager. Custom P4 metadata associated with the packet may or may not be conserved through the cloning operation: it is up to the developer to decide the semantics associated.
- **Resubmission:** the packet, exactly as it is (including its own set of P4 metadata), is sent back to the start of the Ingress pipeline for another round of processing.
- **Multicast:** if the multicast group set to the packet has a match inside the rules defined by the Control Plane, the packet is copied according to them (i.e. if the group is set with two copies, one for port 1 and one for port 2, the packet is copied once and every copy is independently sent to its port). Every packet here created continues its path through the switch, with its specific egress port set. P4 custom metadata is also copied in each packet through this operation.
- **Drop:** already mentioned, if the packet is marked for dropping, it is simply discarded at the end of ingress processing and thus does not reach the Traffic Manager.

Instead, the latter ones (after egress processing) are:

- **Egress-to-Egress (E2E) Cloning:** quite similar to the previous I2E cloning, but it happens after egress processing. The main difference is that here the Parser and Deparser are skipped.
- **Recirculation:** a packet marked for recirculation, at the end of Deparser execution, is sent back to the Parser as it was a new packet. Its P4 custom metadata is preserved through the operation if specified by the developer.
- **Drop:** the packet is dropped at the end of egress processing, and thus does not reach the Deparser.

3.4 Data Plane implementation

Now we will delve inside the implementation of the proposed P4 Data Plane. First of all, we will describe the data structures (constants, headers, metadata, etc.) used within our prototype: then, we will explore the various P4-programmable blocks and how they are used to provide a Network Slicing solution in the Transport layer with QoS delivery, and how the Control Plane interacts with the P4 Data Plane in terms of In-band Network Telemetry (INT) gathering.

Note, again, that this implementation is based on the Simple Switch gRPC target, and thus it is not a general-purpose implementation: it is designed to be used with the Simple Switch gRPC, and it is not guaranteed to work with other P4 targets. However, the structure of the P4 program is designed to be easily adaptable to other targets, as the main logic should be implemented in a way that is independent from the target itself, even though there are no guarantees at all, as the P4 language is not designed for these purposes.

3.4.1 Data structures

This paragraph describes the data structures (headers, metadata, registers, etc.), the constants and the macros used inside the P4 program. It is of capital interest to understand these before of the rest, because they are the building blocks of the proposed Data Plane.

Table 3.4 enumerates the constants, the macros and the type definitions used inside the P4 Data Plane, their respective values and the description of their meaning.

Symbol name	Type	Description	Value
MAX_REG_SIZE	Macro	Maximum size of a register	511
NUM_PRIORITY_QUEUES	Macro	Number of priority queues in the TM	8
PRESERV_FIELDS_ID	Macro	Metadata preservation ID	1
TYPE_IPV4	Constant	IPv4 EtherType	0x0800
TYPE_VLAN	Constant	VLAN EtherType	0x8100
CPU_PORT	Constant	CPU port number	142
CPU_MIRROR	Constant	Mirror ID for the CPU port	100
MAX_QUEUE_SIZE	Constant	Maximum length of queues of the TM	255
egressSpec_t	Type	Type of the egress port fields	bit<9>
macAddr_t	Type	Type of MAC addresses	bit<48>
ip4Addr_t	Type	Type of IPv4 addresses	bit<32>
time_t	Type	Type of timestamps	bit<48>

Table 3.4: Constants, macros and types defined for the Data Plane

The CPU port is a special, abstract, port used to send packets to the Control Plane: when packets are sent to this port, they are locally sent to the process (in the case of the Simple Switch gRPC, to the P4Runtime server) listening to a (transport, i.e. UDP or TCP) port, allowing the Control Plane to inspect and process them.

Instead, *mirroring*, used in CPU_MIRROR, is a simpler, alternative, mechanism offered only by the Simple Switch (gRPC) which provides a way to multicast packets to various

ports without the need of setting up a multicast group: when a packet is marked for mirroring, and that mirror ID is set to one of the specific values previously set by the CP (in our case, 100), the packet is cloned and sent to the specified port, which is the CPU port in this case. This mechanism is implemented through the use of the Thrift-based API, and is not part of the P4Runtime API specification.

The next step is to define the headers used inside the Data Plane: they are defined in Table 3.5 (Ethernet, `ethernet_t`), 3.6 (IPv4, `ipv4_t`), 3.7 (INT, `int_t`) and 3.8 (VLAN, `vlan_t`).

Field name	Type	Description
<code>dstAddr</code>	<code>macAddr_t</code>	Destination MAC address
<code>srcAddr</code>	<code>macAddr_t</code>	Source MAC address
<code>etherType</code>	<code>bit<16></code>	EtherType

Table 3.5: Ethernet header fields

Field name	Type	Description
<code>version</code>	<code>bit<4></code>	IP version used
<code>ihl</code>	<code>bit<4></code>	Internet Header Length
<code>dscp</code>	<code>bit<6></code>	Differentiated Services Code Point
<code>ecn</code>	<code>bit<2></code>	Explicit Congestion Notification
<code>totalLen</code>	<code>bit<16></code>	Packet total length
<code>identification</code>	<code>bit<16></code>	ID for fragments in a datagram
<code>flags</code>	<code>bit<3></code>	The three flags R, DF, MF
<code>fragOffset</code>	<code>bit<13></code>	Offset of this fragment in the datagram
<code>ttl</code>	<code>bit<8></code>	Time-To-Live
<code>protocol</code>	<code>bit<8></code>	Transport protocol used
<code>hdrChecksum</code>	<code>bit<16></code>	Header checksum
<code>srcAddr</code>	<code>ip4Addr_t</code>	Source IPv4 address
<code>dstAddr</code>	<code>ip4Addr_t</code>	Destination IPv4 address

Table 3.6: IPv4 header fields

Finally, our custom metadata is showed in Table 3.9: the last column tells whether the specific field is marked for preservation during the cloning operation (i.e. it is marked with the `PRESERV_FIELDS_ID` tag). Note that the `to_report` field corresponds to the `protocol` field in the INT header: it is simply copied over when assembling the packet in order to report telemetry. The same is valid for `payload` (INT header) and `bandwidth` (metadata). Moreover, every other field marked for preservation has the same destiny: they are copied over to the INT header when the packet is cloned, thus the similarity between the two headers.

The structure `code1_t` is defined as according to [15]: the implementation there used has been the basis for our modifications, made according to the need of providing a Network Slicing mechanism and accomodating the CoDel algorithm on a multi-queue basis instead of a single queue. More details on this in Section 3.4.5.

Field name	Type	Description
protocol	bit<16>	Reason of telemetry report
ingress_global_timestamp	time_t	From <code>standard_metadata_t</code>
egress_global_timestamp	time_t	From <code>standard_metadata_t</code>
enq_timestamp	bit<32>	From <code>standard_metadata_t</code>
enq_qdepth	bit<24>	From <code>standard_metadata_t</code>
deq_timedelta	bit<32>	From <code>standard_metadata_t</code>
deq_qdepth	bit<24>	From <code>standard_metadata_t</code>
payload	bit<40>	Telemetry payload
slice_id	bit<8>	Slice ID
subslice_id	bit<8>	Subslice ID
priority	bit<3>	Priority of the packet

Table 3.7: INT header fields

Field name	Type	Description
tpid	bit<16>	Tag protocol identifier
pcp	bit<3>	Priority Code Point
dei	bit<1>	Drop Eligible Indicator
vid	bit<12>	VLAN Identifier

Table 3.8: VLAN header fields

Field	Type	Description	Preserved
to_report	bit<3>	Reason for telemetry report	Yes
ingress_global_timestamp	time_t	From <code>standard_metadata_t</code>	Yes
egress_global_timestamp	time_t	From <code>standard_metadata_t</code>	Yes
enq_timestamp	bit<32>	From <code>standard_metadata_t</code>	Yes
enq_qdepth	bit<19>	From <code>standard_metadata_t</code>	Yes
deq_timedelta	bit<32>	From <code>standard_metadata_t</code>	Yes
deq_qdepth	bit<24>	From <code>standard_metadata_t</code>	Yes
bandwidth	bit<32>	Bandwidth used by the slice	Yes
slice_id	bit<4>	Slice ID	Yes
subslice_id	bit<4>	Subslice ID	Yes
priority	bit<3>	Priority of the packet	Yes
slice_hash	bit<32>	Hash of slice and subslice IDs	No
is_elephant	bit<1>	Packet in an elephant flow	No
dropped	bit<1>	Packet scheduled for dropping	No
code1	code1_t	CoDel parameters. Unused	No

Table 3.9: Custom metadata defined for the Data Plane

In the end, it is worth noting that a global register has been defined in the Data Plane, which is used to store the length of a Virtual Queue (as described in [12]): in particular, it is a 10-bit register of size 8, where every entry is the length of the Virtual Queue associated with the (same indexed) priority queue of the egress port in the TM. This register is called `vqueues_reg`.

3.4.2 Parser

The Parser is the first programmable block of the v1model architecture and, as already known, it is responsible for parsing the incoming packets, extracting the relevant headers according to the logic defined by the P4 program. The Parser is implemented as a Finite-State Machine (FSM), where each state represents a specific point in the parsing process, and transitions between states are triggered by the presence or absence of specific headers or by the values inside the headers themselves.

The main design choice made in the Parser is to keep it simple and light as possible, as to not burden the switch with unnecessary processing. Therefore, the Parser is designed to parse only the Ethernet, IPv4 and VLAN headers, and to skip any other header that is not relevant for the purposes of this thesis. In fact, as the traffic going through the switch is the one going through the Transport layer (N3 link), the packets here are encapsulated inside the GTP tunnel, built on top of the UDP protocol: the outer UDP layer does not have relevant informations for the purposes of this thesis, and thus it is not parsed by the Parser. A possible choice could have been to parse the inner layers (i.e. the one inside the GTP tunnel), but this would have required a more complex Parser, which would have heavily impacted the performance of the switch.

In Figure 3.3 the state diagram of the Parser is shown: it starts from the *start* state, then it transitions towards the Ethernet parsing state, where the Ethernet header is extracted. Now, depending on the EtherType field, it can either parse the IPv4 header (if the EtherType is 0x0800) or the VLAN header (if the EtherType is 0x8100). In the latter case, the VLAN header is extracted, the previous bits of the EtherType are shifted as to not lose them and if, following a verification on the composition of the current header, there is a failure an error is thrown. After this, the Parser transitions to the IPv4 parsing state, where the IPv4 header is finally extracted. In the former case, instead, the transition is direct to the IPv4 parsing state. The packet, now, is finally accepted and sent to the Checksum Verification block. Listing 3.1 shows the code of the Parser.

```
1 parser MyParser(packet_in packet,
2                 out headers hdr,
3                 inout metadata meta,
4                 inout standard_metadata_t standard_metadata) {
5
6     state start {
7         transition select (standard_metadata.ingress_port) {
8             default: parse_ethernet;
9         }
10    }
11
12    state parse_ethernet {
```

```
13     packet.extract(hdr.ethernet);
14     transition select (hdr.ethernet.etherType) {
15         TYPE_IPV4: parse_ipv4;
16         TYPE_VLAN: parse_vlan;
17         default: accept;
18     }
19 }
20
21 state parse_vlan {
22     packet.extract(hdr.vlan);
23     hdr.ethernet.etherType = hdr.vlan.pcp ++ hdr.vlan.dei ++
        hdr.vlan.vid;
24     bit<16> tmp = hdr.vlan.tpid;
25     hdr.vlan.tpid = TYPE_VLAN;
26     hdr.vlan.pcp = tmp[2:0];
27     hdr.vlan.dei = tmp[3:3];
28     hdr.vlan.vid = tmp[15:4];
29     verify(hdr.vlan.vid != (bit<12>) 0xFFFF,
        error.InvalidHeaderField);
30     transition select (hdr.ethernet.etherType) {
31         TYPE_IPV4: parse_ipv4;
32         default: accept;
33     }
34 }
35
36 state parse_ipv4 {
37     packet.extract(hdr.ipv4);
38     transition select (hdr.ipv4.protocol) {
39         default: accept;
40     }
41 }
42 }
```

Listing 3.1: Parser implementation

If the Parser encounters an error, or the headers effectively present during transitions are not the ones expected as described above, packets are still accepted and sent to the Checksum Verification block but, as we will see later, these will have little impact on the computation as the mechanisms used to provide our service skip packets that do not provide the expected headers. A better solution would have been to drop the packet, but the Simple Switch (gRPC) does not provide a way to transition to the **reject** state within the Parser.

Checksum verification

Even though the Checksum Verification is not properly a part of the Parser, but a separate block, it can be considered an extension to it. Within our framework, the checksum verification is skipped, as it is not needed for the purposes of this thesis. Therefore, every packet successfully parsed is not checked for checksum validity, thus lightening the processing load

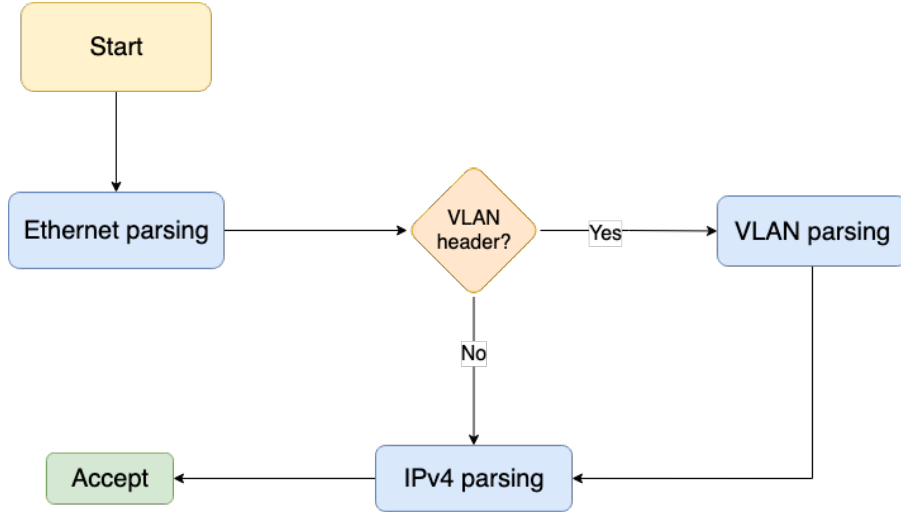


Figure 3.3: Parser state diagram

on the switch and improving its overall throughput and latency. Packets therefore just pass through this step and are directly sent to the Ingress.

3.4.3 Ingress

The Ingress block is the first control block, of the two of the v1model architecture, and thus it is responsible for processing and routing the packets after they have been parsed and their checksum has been verified.

Before delving into the explanation of how our Ingress is implemented, it is important to make two premises: first of all, all the traffic is assigned temporarily the lowest priority (this will be managed later, in the case the packet belongs to a slice with a different priority). Then, if the packet does not have the IPv4 header, or it is invalid, it is not further processed and directly sent to the Traffic Manager (thus with the lowest priority).

The first action taken by the Ingress is then to route the packet: this is done through a Match-Action Table that associates each destination IP address (i.e. the UPF IP address) to a specific egress port, changes the destination MAC address of the packet to the one associated with that UPF (we assume that the UPF is directly connected to the switch, thus reachable through a single hop: this is not a limitation of the framework, as the rules can be easily changed through the Control Plane and therefore accomodate a more complex topology), and finally decreases the TTL by one.

The key of this MAT is the 32-bit destination IP address of the packet, and the match operates with the *ternary* semantics. In the event of a table miss, the default action to be taken, as by our solution, is to drop the packet. This is done as the prototype here proposed is thought as a Proof-of-Concept instead of a fully fleshed-out industrial-grade software: in the latter case further inspection and actions on the packet may be necessary.

To provide a Network Slicing solution, once the routing has been made, we need to define the slices and, eventually, the subslices: this is done through two cascading MATs, the first one to assign a slice ID and the second one its subslice ID.

The Slice ID and the Subslice ID together provide a unique identifier for the Network Slice, and are stored in the user-defined metadata of the packet (inside `slice_id` and `subslice_id` fields, respectively).

Once the traffic gets classified into Network Slices, we need to assign Quality-of-Service parameters to the Network Slices: this is still done through a (fourth) MAT, which has as keys the Slice ID and the Subslice ID (both exact matches), and as (unique) action one that sets the maximum bandwidth, the desired maximum latency (in terms of the Simple Switch, the priority queue of the Traffic Manager) and the minimum bandwidth for the Network Slice. While the minimum bandwidth is managed by the Control Plane, the other two parameters are managed directly by the Data Plane, at line-rate. This algorithm, along with the MAT, is shown in Listing 3.2.

```
1 table slice_mgmt_exact {
2     key = {
3         meta.slice_id: exact;
4         meta.subslice_id: exact;
5     }
6     actions = {
7         manage_slice;
8         NoAction;
9     }
10    default_action = NoAction;
11 }
12
13 action manage_slice(bit<32> max_bandwidth, bit<3> priority,
14    bit<32> min_bandwidth) {
15     bit<32> byte_cnt;
16     time_t last_time;
17     time_t cur_time = standard_metadata.ingress_global_timestamp;
18     byte_cnt_reg.read(byte_cnt, meta.slice_hash);
19     last_time_reg.read(last_time, meta.slice_hash);
20
21     if ((byte_cnt >> 10) >= max_bandwidth && cur_time - last_time
22         < 1000000) {
23         drop();
24     } else if ((byte_cnt >> 10) >= (max_bandwidth >> 1) &&
25         hdr.ipv4.ecn != 0) {
26         hdr.ipv4.ecn = 3;
27     }
28
29     if (cur_time - last_time >= 1000000) {
30         if (byte_cnt >> 10 < min_bandwidth)
31             meta.to_report = 4;
32         else
33             meta.to_report = 2;
34         meta.bandwidth = byte_cnt >> 10;
35         byte_cnt = 0;
```

```
32     last_time_reg.write(meta.slice_hash, cur_time);
33 }
34 if (meta.dropped == 0) {
35     byte_cnt = byte_cnt + standard_metadata.packet_length;
36     byte_cnt_reg.write(meta.slice_hash, byte_cnt);
37
38     bit<10> queue_len;
39     standard_metadata.priority = priority;
40     vqueues_reg.read(queue_len, (bit<32>)
41         standard_metadata.priority);
42     vqueues_reg.write((bit<32>)
43         standard_metadata.priority, queue_len + 1);
44 }
```

Listing 3.2: P4 code for QoS management of Network Slices

The code above supposes the existence of two registers, `byte_cnt_reg` and `last_time_reg`, which are used to store the number of bytes sent by the Network Slice and the last time the slice was checked, respectively. These two are defined at a block-level, thus not visible by the entire program but only by the Ingress block. In these, the `slice_hash` field of the metadata is used as the key, which is a 32-bit hash computed from the Slice ID and the Subslice ID.

At line 20, if the used bandwidth is above the maximum allowed and the time elapsed since the last check is less than one second, the packet is dropped: the `drop()` function is a special action that marks the packet for dropping at the end of the current block. The time check is mandatory because it solves one particular corner case: that is, if the time elapsed is more or equal to one second then the packet is possibly sent for telemetry to the Control Plane, and thus it should not be dropped. Otherwise, in the absence of this check, the telemetry packet would be marked for dropping and not able to reach later stages and thus the Control Plane.

Instead, if the bandwidth used is above half of the maximum allowed and the transport network is ECN-capable, then the ECN field of the IPv4 header is set to 3, which means, recalling the previous Chapter, *Congestion Experienced*.

Successively, if more than one second has passed since the last reset of the counters, the bandwidth used is saved into the metadata of the packet, and if the bandwidth used is below the minimum allowed, then the telemetry report reason is set to 4, which means *underutilization*, otherwise it is set to 2, which means *normal bandwidth report*. Then the counters are reset: the bytes one with the size of the current packet, and the last time one with the current timestamp.

Otherwise, if not enough time has elapsed and the packet is not marked for dropping, the bytes counter is incremented by its size and the priority value is set. In the end the length of the Virtual Queue associated with that priority queue is incremented by one, as the packet is now being enqueued inside the Traffic Manager. Details on how the priority value is determined for the particular Network Slice are given later, when the Control Plane and its interaction with the Data Plane are explained.

3.4.4 Traffic Manager

The Traffic Manager is not a P4-programmable block, but it is anyway a crucial part of our framework: it is responsible for holding packets in queues, and for dequeuing them according to their priority. The algorithm this component adopts is the *Strict Priority* one, which means that packets are dequeued from the highest priority queue first, and only when that queue is empty the next one is considered.

Without further customization, the Traffic Manager contains only one queue per egress port: when starting the Simple Switch (gRPC) it is possible to specify the number of queues per port, and through the Thrift-based API it is allowed to specify the size and the dequeuing rate of a specific (set of) queues.

As the priority value inside `standard_metadata_t` is set and its type is `bit<3>`, the maximum number of queues is 8. Therefore, our Traffic Manager will have 8 priority queues per egress port. We are mainly interested, as per our scenario (see Chapter 1), in the egress link towards the UPFs. The least priority queue is the one with index 0, while the highest priority one is the one with index 7. Inside, the queues are all implemented with a FIFO algorithm, therefore the only way to reorder packets is leveraging the multi-queue nature of the TM.

Figure 3.4 shows the architecture of the Traffic Manager, with the 8 priority queues and the egress port, even though only the queues towards a single port are shown (with multiple egress interfaces, the queues are the same and replicated together, separated from the other sets).

Leveraging this element is therefore mission-critical for the implementation of the proposed framework, as it allows to provide Quality-of-Service delivery for the Network Slices even when using unresponsive flows, such as in our use case.

Once descheduled, packets are sent to the Egress computational block, where they are further processed.

3.4.5 Egress

The Egress block is the second control block of the v1model architecture, and it is responsible for processing packets after they have been dequeued from the Traffic Manager. The most important difference among the Ingress and this block is that the Egress block is invoked only *after* routing has been done (in the Ingress), and thus it is not responsible for routing packets, but only for further processing: therefore, here the value of `egress_port` inside `standard_metadata_t` is already set, and it is not modified anymore.

Within our prototype, the Egress block is responsible for two high-level main tasks:

- **Setting the source MAC address:** the source MAC address of the packet is set to the one associated with the egress port, as defined by the Ingress block.
- **Cloning packets for telemetry purposes:** if the packet is marked for telemetry, it is cloned and sent to the Control Plane, otherwise it is not cloned and sent directly to the Deparser.

In Listing 3.3 the Egress implementation is shown.

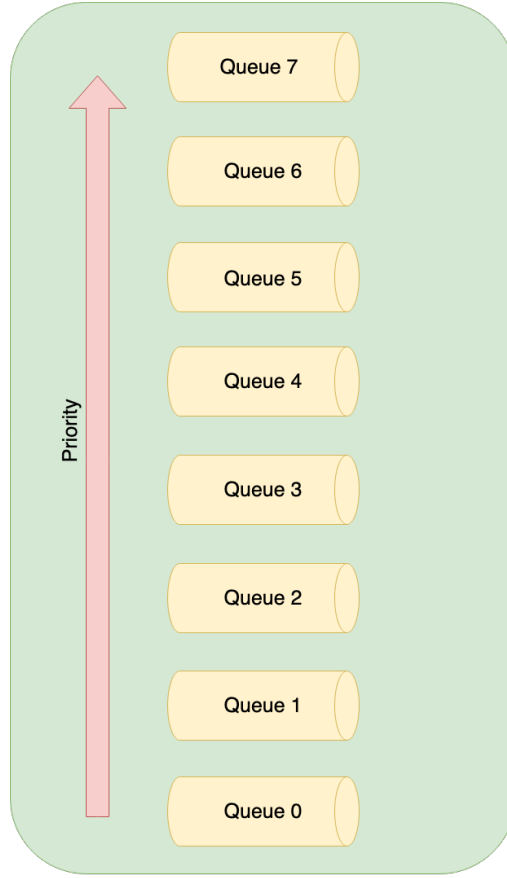


Figure 3.4: Traffic Manager architecture

```
1 control Egress(inout headers hdr,
2               inout metadata meta,
3               inout standard_metadata_t standard_metadata) {
4
5     register<time_t>(MAX_REG_SIZE) last_priority_report_reg;
6     c_code1() c_code1_0;
7
8     action clone_packet() {
9         meta.ingress_global_timestamp =
10             standard_metadata.ingress_global_timestamp;
11         meta.egress_global_timestamp =
12             standard_metadata.egress_global_timestamp;
13         meta.enq_timestamp = standard_metadata.enq_timestamp;
14         meta.enq_qdepth = standard_metadata.enq_qdepth;
15         meta.deq_timedelta = standard_metadata.deq_timedelta;
16         meta.deq_qdepth = standard_metadata.deq_qdepth;
17         meta.priority = standard_metadata.priority;
```

```
16         clone_preserving_field_list(CloneType.E2E, CPU_MIRROR,
17             PRESERV_FIELDS_ID);
18     }
19     action rewrite_smac(macAddr_t macAddr) {
20         hdr.ethernet.srcAddr = macAddr;
21     }
22
23     table smac_exact {
24         key = {
25             standard_metadata.egress_port: exact;
26         }
27         actions = {
28             rewrite_smac;
29             NoAction;
30         }
31         default_action = NoAction();
32         const entries = {
33             1: rewrite_smac(48w0xaaaf3591688);
34             2: rewrite_smac(48w0xda81d8cc552a);
35             3: rewrite_smac(48w0xbaf370fad58f);
36         }
37     }
38
39     apply {
40         if (standard_metadata.instance_type != 0) {
41             hdr.telemetry.setValid();
42
43             hdr.telemetry.protocol = (bit<16>) meta.to_report;
44             hdr.telemetry.ingress_global_timestamp =
45                 meta.ingress_global_timestamp;
46             hdr.telemetry.egress_global_timestamp =
47                 meta.egress_global_timestamp;
48             hdr.telemetry.enq_timestamp = meta.enq_timestamp;
49             hdr.telemetry.enq_qdepth = (bit<24>) meta.enq_qdepth;
50             hdr.telemetry.deq_timedelta = meta.deq_timedelta;
51             hdr.telemetry.deq_qdepth = (bit<24>) meta.deq_qdepth;
52             hdr.telemetry.payload = (bit<40>) meta.bandwidth;
53             hdr.telemetry.priority = (bit<8>) meta.priority;
54             hdr.telemetry.slice_id = (bit<8>) meta.slice_id;
55             hdr.telemetry.subslice_id = (bit<8>) meta.subslice_id;
56
57             hdr.vlan.setInvalid();
58             hdr.ethernet.setInvalid();
59             hdr.ipv4.setInvalid();
60         } else {
61             smac_exact.apply();
62         }
63     }
```

```

61         if (meta.to_report >= 4 &&
            standard_metadata.enq_qdepth < (bit<19>) (3 *
            MAX_QUEUE_SIZE) >> 2 &&
            standard_metadata.deq_qdepth < (bit<19>)
            MAX_QUEUE_SIZE >> 2) {
62             meta.to_report = 2;
63         }
64
65         meta.codel.queue_id = (bit<9>) meta.slice_hash;
66         /* c_codel_0.apply(hdr, meta, standard_metadata); */
67
68         bit<10> queue_len;
69         @atomic {
70             vqueues_reg.read(queue_len, (bit<32>)
                standard_metadata.priority);
71             vqueues_reg.write((bit<32>)
                standard_metadata.priority, queue_len - 1);
72         }
73
74         if (meta.codel.state_dropping == 1w1) {
75             time_t last_time;
76             last_priority_report_reg.read(last_time,
                meta.slice_hash);
77             if (standard_metadata.egress_global_timestamp -
                last_time >= 1000000) {
78                 meta.to_report = meta.to_report + 1;
79                 last_priority_report_reg.write(meta.slice_hash,
                    standard_metadata.egress_global_timestamp);
80             }
81         }
82
83         if (meta.to_report > 0)
84             clone_packet();
85     }
86 }
87 }

```

Listing 3.3: P4 code for the Egress block

The `apply` block, conceptually the same as the `main` method in a traditional programming language, distinguishes between two cases: packets going through the normal path and packets already cloned. Let’s examine both possibilities:

- **Normal path:** when the packet is not a cloned one, its source MAC address is set to the one associated with the egress interface; this is done through the `smac_exact` MAT. Then, the status of the queue the packet was dequeued from is checked through the `standard_metadata` object: if the queue is not congested (we took as an indicator of decongestion an utilization of less than $\frac{3}{4}$ of the total depth of the queue at enqueue time and less than $\frac{1}{4}$ utilization of the same queue at dequeue time) and the packet was marked for telemetry reporting with reason 4 (i.e. underutilization), then the reason

is set to 2, which means normal bandwidth report. This is done to avoid sending telemetry reports for underutilization when the queue is not congested, as it would not provide any useful information to the Control Plane and would only increase the overhead of the telemetry system. The Virtual Queues register is then updated, decrementing the length of the queue by one unit. Then, if the packet is marked for cloning (i.e. `meta.to_report` is greater than 0), the `clone_packet()` action is invoked, which saves the relevant `standard_metadata` fields into the user metadata and clones the packet, preserving the fields defined with the `PRESERV_FIELDS_ID` tag. The cloning operation is done through the `clone_preserving_field_list()` function, which leverages the Packet Replication Engine (PRE) previously discussed. The choice of not cloning the packet in the Ingress has considerable advantages, as it allows to avoid the overhead of cloning packets that are not needed, and thus to reduce the load on the Control and on the Data Planes. Packets cloned, as known, are sent before all to the Traffic Manager and marked automatically with an `instance_type` value greater than 0, which is used to distinguish them from the normal packets; then they will be processed through the Egress block, but in a different way, through the second case of the `apply` block.

- **Cloned packets:** when the packet is a cloned one, the telemetry header is set to valid, and the relevant metadata fields are copied over to the telemetry header. Finally, the Ethernet, VLAN and IPv4 headers are set to invalid, as they are not needed anymore: the telemetry header is the only one that will be emitted in the Deparser, as it is the only one that contains the telemetry data. This is done to avoid sending unnecessary headers to the Control Plane, as it is programmed to only parse the telemetry information and not the standard TCP/IP headers.

Note that the `c_code1` block is provided by the P4 CoDel implementation ([15]), adapted to our multi-queue needs. More details on the implementation of it can be seen in Appendix B. In this prototype we do not use it (commented out in the code), but it is still present in the codebase as a reference for future work (lines 65–66 and 74–81): in particular, it is worth noting how it can be used to send latency reports (identifying latency issues with the *dropping phase* of the CoDel algorithm); the register here used serves to keep latency reports limited in quantity, as it caps reports at a rate of one per second.

In Table 3.10 the reasons for telemetry reporting (values of `meta.to_report` and its dual, within the INT header, `hdr.telemetry.protocol`) are shown, along with their associated values. A value of 0 means that the packet is not to be cloned and thus no report is to be sent to the Control Plane.

3.4.6 Checksum computation

Inside this block, the checksum of the IPv4 header is re-computed and put inside the header, as it is mandatory because of the modifications made by the Ingress block (i.e. the decrement of the TTL and, possibly, the setting of the ECN field).

Once done, the packet is sent to the Deparser for the final reassembly of the packet and its transmission towards the output port.

Reason	Value
Latency issue (LI)	1
Normal bandwidth report (NBR)	2
LI & NBR	3
Underutilization (UU) & NBR	4
LI & UU	5

Table 3.10: Reasons for telemetry reporting

3.4.7 Deparser

The Deparser is the last programmable block of the v1model architecture, and it is responsible for reassembling the packet after the processing done by the Ingress, Checksum Computation and Egress blocks. It emits the headers in instruction-order, but skipping the ones that have their *validity bit* set to 0. This is useful for the INT header, which is only emitted if the packet is cloned for telemetry purposes, and thus it is usually 0.

The order of emission of the headers is the following: first, the INT header, if needed, as this is the only way the Control Plane can read the telemetry data; then, in the order, the Ethernet header, the VLAN header (if present) and the IPv4 header. Remember that, in the Egress stage, the Ethernet, VLAN and IPv4 headers are set to invalid if the packet is cloned for telemetry purposes, thus they will never be emitted along with the INT header.

To emit a packet, as already stated, a Deparser within the v1model architecture uses the `emit()` function available through the architecture-injected `packet_out` object.

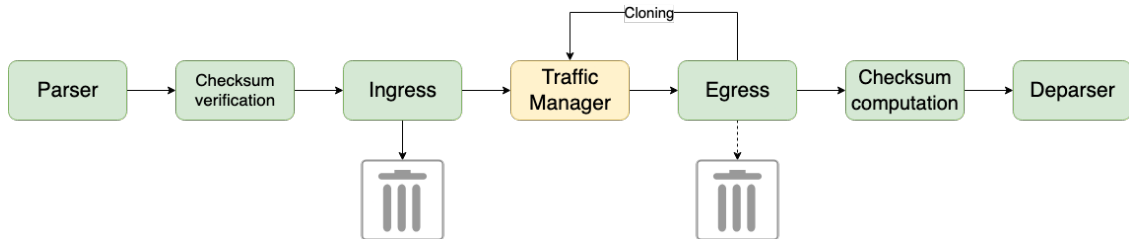


Figure 3.5: Allowed paths of a packet through our P4 pipeline

3.5 In-band Network Telemetry

In this Section we will explore the path telemetry packets take, step by step, to better understand the decisions and the peculiarities of the choices made.

To reach the Orchestrator, a telemetry packet is first sent from the Data Plane to the egress port allocated for the Control Plane (i.e. the CPU port for the Simple Switch targets), then the gRPC server (listening on port 9559, reserved by the IANA for P4Runtime) receives the packet and the Agent’s framework enabler, according to its own strategy, asks the P4 controller to read a certain number of packets from the device buffer (the gRPC server itself). The P4 controller, in turn, uses the P4Runtime API to read the packets from the device buffer, to process them converting into an intermediate representation (Packet

objects), and then it returns them to the framework enabler, which then sends them to the Orchestrator through the `POST /agent/telemetry` endpoint. The Orchestrator, finally, receives the telemetry packets and stores them in the database.

Between the P4 device and the gRPC server there needs to be a pre-negotiation on the format and contents of the telemetry packets, as otherwise the latter would not be able to read and process them. This is done through the P4Info file, which is one of the two outputs of the P4 compiler (p4c): it contains the structure of the P4 program, including the headers, the metadata and the actions used in the program, but most importantly it contains the structure of the telemetry header (marked with the annotation `@controller_header("packet_in")` within the P4 program) and therefore its fields, the same that will be received from the Data Plane in the same order specified by the P4Info file itself.

The `/agents/telemetry` Orchestrator API endpoint expects a JSON-encoded body, which should contain the telemetry data in an array of objects, each one representing a telemetry packet. These objects must have the mandatory fields defined in Table 3.2 and, in addition, can have some optional fields, depending on the P4 device used. A comprehensive list of these is both not possible to provide and not useful, as it would heavily depend on the specific P4 target used and on its capabilities.

The P4Runtime specification ([22]) defines precisely the runtime control through P4Runtime messages used to interact with P4 devices (in particular, the ones following the v1model or PSA architectures: efforts are ongoing to standardize this specification independently from the underlying architecture).

Therefore, also the telemetry packets are sent with a *push* communication model between the Agent and the Orchestrator, through which it is possible to effectively decouple these two components and to allow the Agent to send telemetry packets whenever it is needed, without waiting for a request from the Orchestrator. This is useful to avoid delays in the telemetry reporting, as the Agent can send telemetry packets as soon as they are available.

Chapter 4

Results and evaluation

4.1 Experimental settings

In order to validate the proposed framework, a testbed has been set up.

It consists of a set of 6 (six) virtualized machines and 1 (one) physical device. The virtualized machines are running on top of DARPA2, the infrastructure provided by IDLab University of Antwerp & imec, a research group focused on the design and development of innovative network architectures and protocols along with Machine Learning and Artificial Intelligence solutions ([2]), whose aid and contributions have been extremely valuable for the success of this thesis. The virtualization platform used to provision the various machines is Proxmox Virtual Environment (VE), a powerful and flexible open-source virtualization platform based on KVM and LXC containers. The physical device, instead, is a Raspberry Pi 3 Model B+.

In Figure 4.1 a general overview of the testbed is shown.

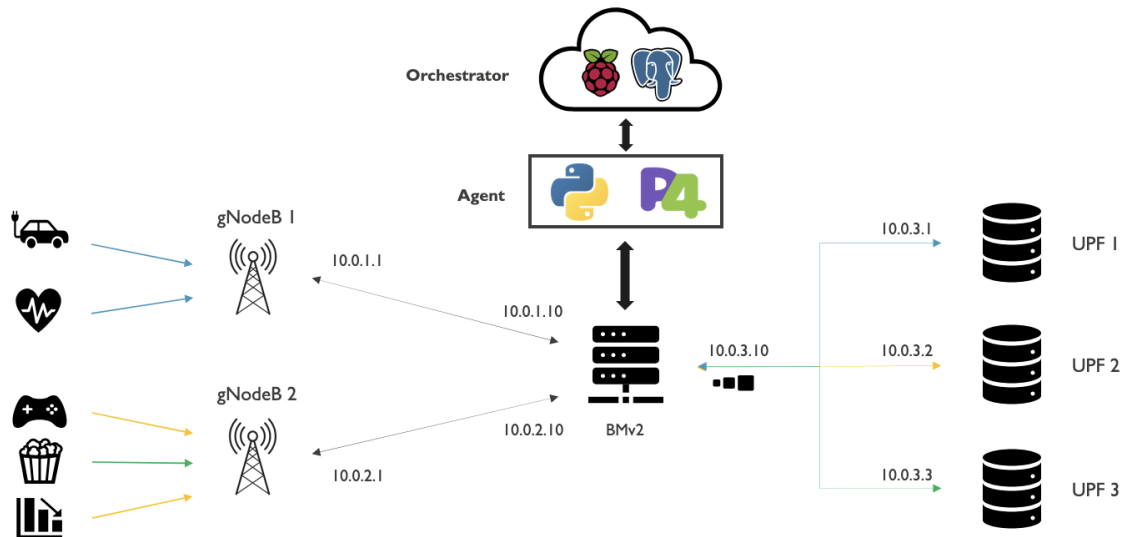


Figure 4.1: General overview of the testbed

4.1.1 Hardware setup

As anticipated, the testbed is based on the DARPA2 infrastructure (whose hardware specifications are shown in Table 4.1), which is used to run the virtualized machines.

Specification	Value	Quantity
CPU	Intel Xeon E5-2620 v4 @ 2.10 GHz	2 (8 cores, 2 threads each)
RAM	160 GB DIMM DDR4	4 x 8 GB, 4 x 32 GB
HDD	10 TB	2 x 4 TB, 2 x 480 GB, 1 x 1 TB

Table 4.1: Specifications of the DARPA2 infrastructure

In Table 4.2 the specifications of the gNodeB machines are summarized: these latter two are the ones acting as “clients” of the framework towards the UPFs, which are in turn the ones acting as “servers” of the framework and are hosted on other three machines (whose hardware specifications are in Table 4.3), each one independent from the others. The gNodeBs are connected to the UPFs through the last remaining Virtual Machine, which acts as a router among the three subnets this testbed is divided into and runs the actual implementation of the P4 device, the Simple Switch (gRPC) target, while also hosting the Simple Switch (gRPC)’s Agent: its specification are in Table 4.4. The hardware shown represents the resources allocated to each virtualized machine, which are taken from the total resources of the DARPA2 infrastructure.

Machine	Cores	Threads/Core	Clock	RAM
gNodeB 1	2	1	2 GHz	9000 MB
gNodeB 2	2	1	2 GHz	9000 MB

Table 4.2: Specifications of the gNodeB machines

Machine	Cores	Threads/Core	Clock	RAM
UPF 1	4	1	2 GHz	15 GB
UPF 2	4	1	2 GHz	15 GB
UPF 3	4	1	2 GHz	15 GB

Table 4.3: Specifications of the UPF machines

Specification	Value
Cores	4
Threads/Core	1
Clock	2 GHz
RAM	16 GB

Table 4.4: Specifications of the P4 router machine

These virtualized machines are connected, through the Proxmox VE infrastructure and the physical network provided by IDLab, first to a switch (not the one running the P4

program, but a simple Ethernet switch) and then to the Raspberry Pi 3 Model B+, which runs the Orchestrator (the software details of this deployment will be explained later). The hardware specifications of this device are shown in Table 4.5.

Specification	Value
CPU	Cortex-A53 (ARMv8) 64-bit SoC @ 1.4 GHz
RAM	1 GB LPDDR2 SDRAM
Connectivity	Gigabit Ethernet over USB 2.0 (max 300 Mbps)
PoE	Supported
Memory	16GB microSD card

Table 4.5: Specifications of the Raspberry Pi 3 Model B+

The links between the various machines are made either through the Proxmox VE infrastructure or through Ethernet cables (in the case of the Raspberry Pi 3 Model B+), and the final network topology is shown in Figure 4.2. Note that the link between the Raspberry Pi and the Proxmox VE infrastructure is only theoretically a Gigabit Ethernet link, as the Raspberry Pi 3 Model B+ has a Gigabit Ethernet port but it is connected through a USB 2.0 interface, which limits the maximum throughput to 300 Mbps. The Proxmox VE infrastructure, instead, is connected to the IDLab’s network, which is a high-speed network with a maximum throughput of more than 15 Gbps. The Proxmox VE infrastructure is also connected to the Internet, allowing the virtualized machines to access external resources and to be accessed from the outside world, if needed.

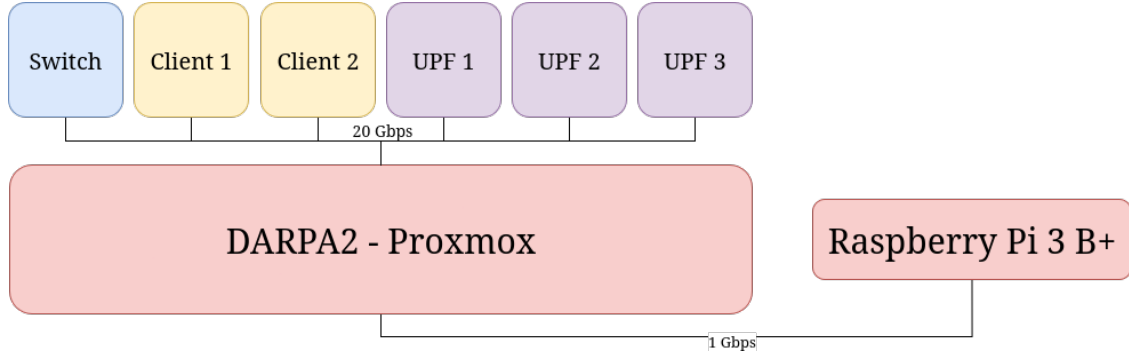


Figure 4.2: Network topology of the testbed

4.1.2 Software setup

The testbed is equipped with a set of state-of-the-art software components, which are used to run the framework and to perform the experiments. These are listed and explained in Tables 4.6 (for gNodeB machines), 4.7 (for UPF machines), 4.8 (for the P4 router machine) and 4.9 (for the Orchestrator).

Software	Version
Operative System	Ubuntu 22.04.5 LTS
Kernel	5.15.0-0133-generic
iperf	3.9
MGen	5.1.1

Table 4.6: Software running on the gNodeB machines

Software	Version
Operative System	Ubuntu 22.04.5 LTS
Kernel	5.15.0-0134-generic
iperf	3.9
MGen	5.1.1

Table 4.7: Software running on the UPF machines

MGen (Multi-Generator) is a traffic generator and measurement tool, developed by the U.S. Naval Research Laboratory, which is used to generate traffic flows and to measure the performance of the network. It is used to generate traffic flows between the gNodeBs and the UPFs, and to measure the performance of the network in terms of throughput, latency and packet loss. It is also capable of generating flows with specific and heterogeneous patterns, as it allows the definition of the traffic through configuration files (with the `.mgen` extension).

Throughout our experiments, it is used alongside `iperf3`, which is a well-known tool for measuring the maximum achievable bandwidth on a network path, and it is used to measure the performance of the network in terms of throughput and packet loss (but not latency, as it is not designed for that). Both are particularly relevant for the experiments conducted.

The choice of Ubuntu 22.04.5 LTS as the operative system within gNodeB and UPF machines is due to its stability and long-term support, but more importantly to the fact that it is the recommended operative system for the P4 development environment, as the tools used to compile, deploy and run P4-related software is tested on this version of Ubuntu.

Note that the P4 program is compiled with the `p4c` compiler, which is the official P4 compiler developed by the P4 Language Consortium, and then deployed on the P4 router machine, by being fed to the Simple Switch (gRPC) target (which accepts the binary file in JSON format). In this case, the Simple Switch (gRPC) software has been recompiled, according to [6], to drastically improve its performance, as the original implementation struggled to handle more than 30 Mbps of traffic, while the recompiled version is able to handle more than 200 Mbps, which is more than enough for the testbed’s needs, but still lower than the promised 1 Gbps by the implementation. This is probably due to the fact that we are running on top of a virtualized infrastructure, which adds some overhead to the performance of the devices; moreover, possible misconfigurations cannot be ruled out.

Mininet is a network emulator that allows to create a virtual version of almost every

Software	Version
Operative System	Ubuntu 22.04.5 LTS
Kernel	5.15.0-0133-generic
p4c	1.2.5.4
Simple Switch (gRPC)	1.15.0-d12eefc7
Python	3.10.12
pip	22.0.2
Protobuf	3.20.3
p4runtime-shell	0.0.3.post21+g7374d7c.d20250304
Mininet	2.3.1b4
grpcio	1.51.3
Flask	3.1.0
Thrift	0.16.0
Behavioral Model	1.15.0

Table 4.8: Software running on the P4 router machine

network topology and to run different virtual machines on top of it, simulating the behavior of a real network. Although it has been explored as a possible solution to run the prototype, it has not been used in the end, as its performance would severely hit and degrade the effectiveness of the framework.

Instead, **Protobuf** is a language-agnostic serialization format developed by Google, which is used, within our framework, to define the messages exchanged between the P4 controller and the P4 device and their format.

Software	Version
Operative System	Raspberry Pi OS (Debian 12 - Bookworm)
Kernel	6.12.20+rpt-rpi-v8
Python	3.11.2
pip	23.0.1
Flask	2.2.2
Docker	28.1.1
Grafana (Docker image)	11.6.1
PostgreSQL	15.12
psycpg2	2.9.5

Table 4.9: Software running on the Orchestrator

Grafana is a powerful open-source visualization and analytics platform, which is used to visualize the telemetry data collected by the Orchestrator and stored in the PostgreSQL database. It is used to create dashboards and graphs to visualize the performance of the network and to monitor the status of the Network Slices and their parameters. The choice of using Docker to run Grafana is due to the fact that it allows to easily deploy and manage the software, as well as to isolate it from the rest of the system, making it easier to manage

and maintain. The Docker image used is the official one provided by Grafana, which is regularly updated and maintained by the Grafana team.

Moreover, it allows for the integration with an extremely wide range of data sources apart from PostgreSQL, such as Prometheus, InfluxDB, and many others, which can be used to collect and visualize telemetry data from different sources homogeneously, from a single dashboard in a source-independent way.

4.2 Results

In this Section we will show the results obtained from the experiments conducted on the testbed described in Section 4.1. Each experiment is based on multiple runs, in order to obtain a statistically significant result, and the results are averaged over the runs.

Results are taken in two different ways: the first one is through the `iPerf3` and `ping` tools, which are used to measure the throughput and the latency (Round-Trip Time) of the network, respectively; the second one is through the `MGen` tool, which is used to generate traffic flows and to measure the performance of the network in terms of throughput, latency and packet loss. The results may be slightly different depending on the tool used, as they use different methods to measure the performance of the network, but they should be consistent with each other. Moreover, it is worth noting that, in the case of `iPerf3`, the latency measurements are taken with `ping`, therefore there may be some discrepancies as the tools generate different packets which may be treated differently by the network devices, thus discarded. In the end, `ping` measures the Round-Trip Time (RTT) of the packets sent, and not the effective end-to-end latency, which is the time it takes for a packet to travel from the source to the destination and the actual parameter we are interested in. However, the RTT is a good approximation of the end-to-end latency, as it includes the time it takes for the packet to travel from the source to the destination and back, and it is a widely used metric to measure the performance of the network.

Independently from the underlying tools used, every run is conducted for a duration of 80 seconds, splitted into multiple phases:

1. First of all, previous configuration is cleared, if any; this means that the Network Slices previously set for the network are removed.
2. At the beginning, two small flows are generated, from gNodeB 1 to UPF 1, simulating mission-critical traffic.
3. After 20 seconds, other two, larger flows are started, from gNodeB 2 to UPF 2, simulating URLLC traffic.
4. At second 40, background traffic, which is not mission-critical and is not expected to be prioritized, is generated towards UPF 3. This consists in a large volume of data able to congest the network without proper management.
5. After another 20 seconds, the Network Slices are configured and set in the network, giving appropriate resources to each flow, according to the type of traffic they are carrying (i.e., recalling our scenario, their VLAN and destination address, that is the specific UPF).

The particular choice of the flows' size will be shown case by case, when discussing the results of each experiment.

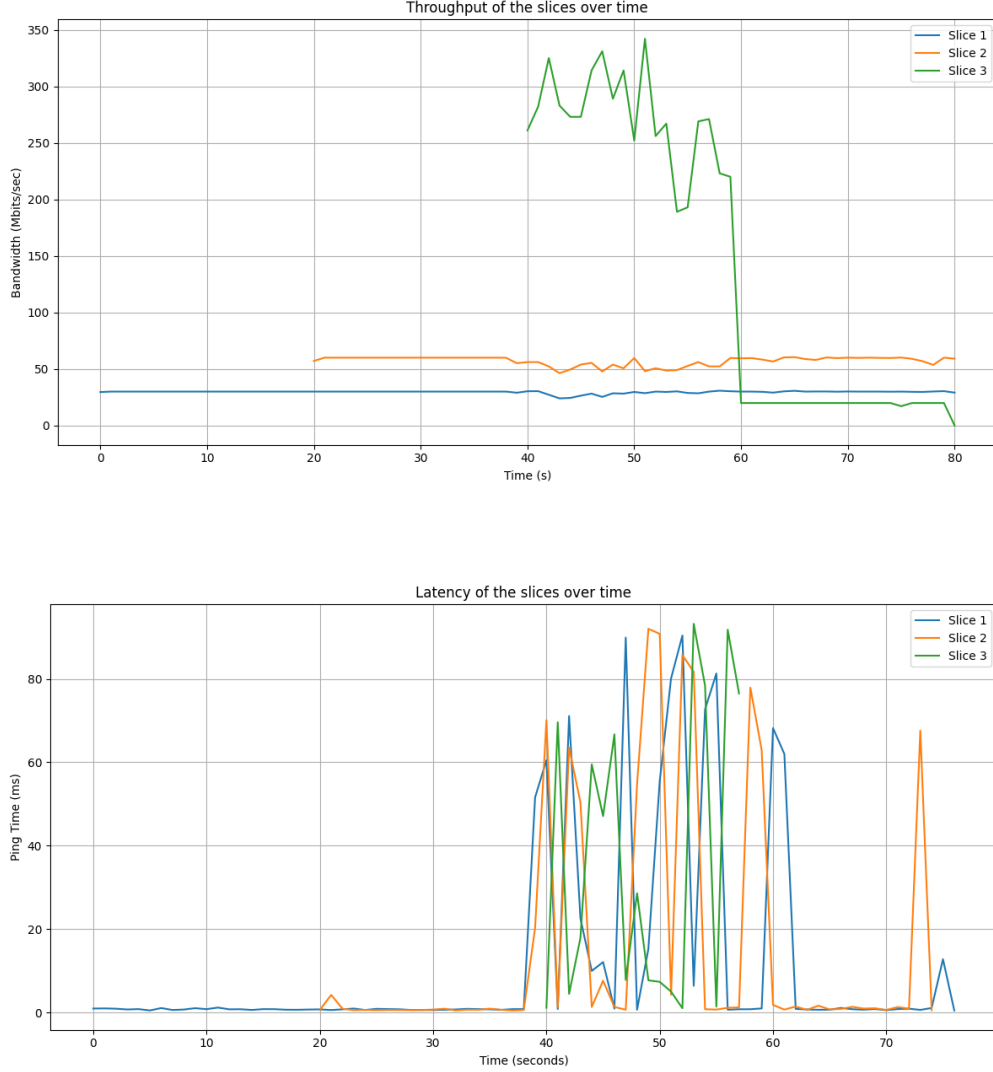


Figure 4.3: First experiment results

In the first experiment (results shown in Figure 4.3) the flows generated are reassumed in Table 4.10.

We can see that the volume of data needed by the background traffic is enough to saturate the link and provoke congestion, thus there is the need for a Slicing mechanism which could restore the normal operativity of the network, allocating the due resources to each flow. At second 60 the Network Slices are configured and set in the network, and we can see that the throughput of the mission-critical and URLLC flows is restored to the expected values, while the background traffic is heavily limited, as it is not mission-critical

Flow	Volume
Mission-critical (1)	15 Mbps
Mission-critical (2)	15 Mbps
URLLC (1)	30 Mbps
URLLC (2)	30 Mbps
Background traffic	300 Mbps

Table 4.10: First experiment flows configuration

and does not need to be prioritized. The latency, which was heavily affected by the congestion, is also restored to the expected values, with the mission-critical flows being the ones with the lowest latency, followed by the URLLC flows and finally the background traffic, which is the one with the highest latency, as expected. The two final spikes in the latency graphs are probably outliers, as they are not consistent with the rest of the data and are probably due to some external factor, such as a temporary issue with the P4 device.

The second experiment conducted is quite similar to the first one, but with a larger volume of data for the background traffic, as to test the infrastructure’s limits under heavy congestion. The flows generated are reassumed in Table 4.11, while the actual results are shown in Figure 4.4.

This time, the background traffic is thus set to 330 Mbps, which is enough to completely saturate the link and provoke severe congestion. Again, at second 60 the Network Slices are configured and set in the network, and we can see that the throughput of the mission-critical and URLLC flows is restored to the expected values and remains stable at that value, while the background traffic is heavily limited. The fact that the mission-critical flows are almost not affected by the congestion is probably due to the fact that they are small flows, and thus they are able to be transmitted quickly, before the congestion can affect them. The latency, which was heavily affected, is also quickly restored to normal values, and the background traffic is clearly affected by the mechanism, as the ICMP request packets, used for RTT measurement, do not even reach the UPF: remember, in fact, that the traffic going back to the gNodeBs is not managed by our Network Slicing mechanism, as the slices are defined only by destination IP address, therefore not setting any rule for the return path results in the packets not being treated by our mechanism and simply forwarded by the P4 device.

Flow	Volume
Mission-critical (1)	15 Mbps
Mission-critical (2)	15 Mbps
URLLC (1)	30 Mbps
URLLC (2)	30 Mbps
Background traffic	330 Mbps

Table 4.11: Second experiment flows configuration

It is worth noting, once again, that the final spike in the mission-critical latency graph is probably an outlier and not consistent with the results obtained.

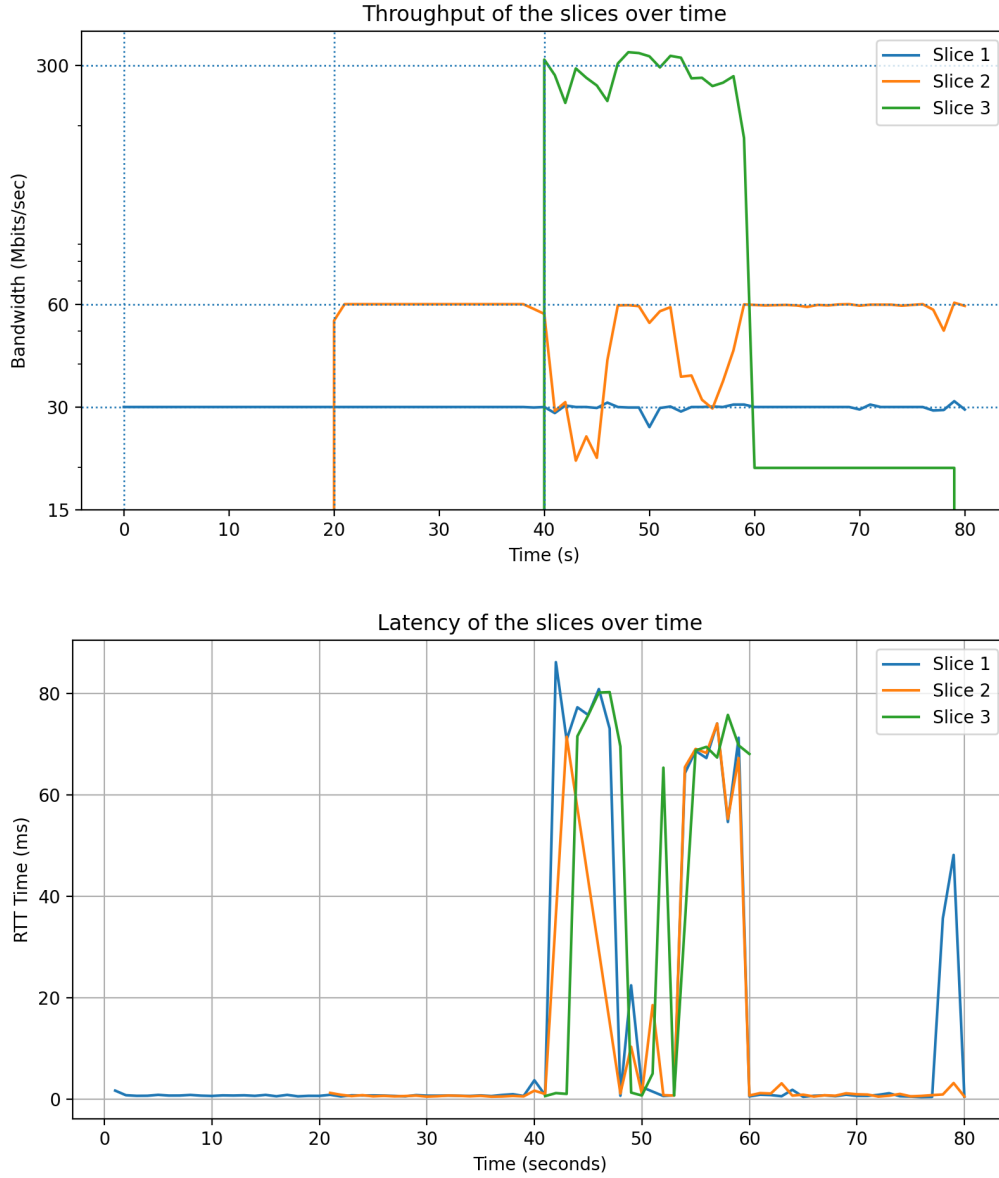


Figure 4.4: Second experiment results

We will dive now into some experiments conducted with **MGen**, which allows to generate more complex traffic patterns, as it is possible to define the traffic through configuration files. The first experiment conducted with this tool is shown in Figure 4.5, while the flows generated are reassumed in Table 4.12.

It is possible to note that while there are similarities with the previous experiments, there are also some discrepancies, especially for what regards the latency of the flows. These differences are probably due to the fact that **MGen** may generate traffic in a different way than **iPerf3**, and thus the packets may be treated differently by the network devices, resulting in different performance metrics. However, the overall behavior of **MGen** has been

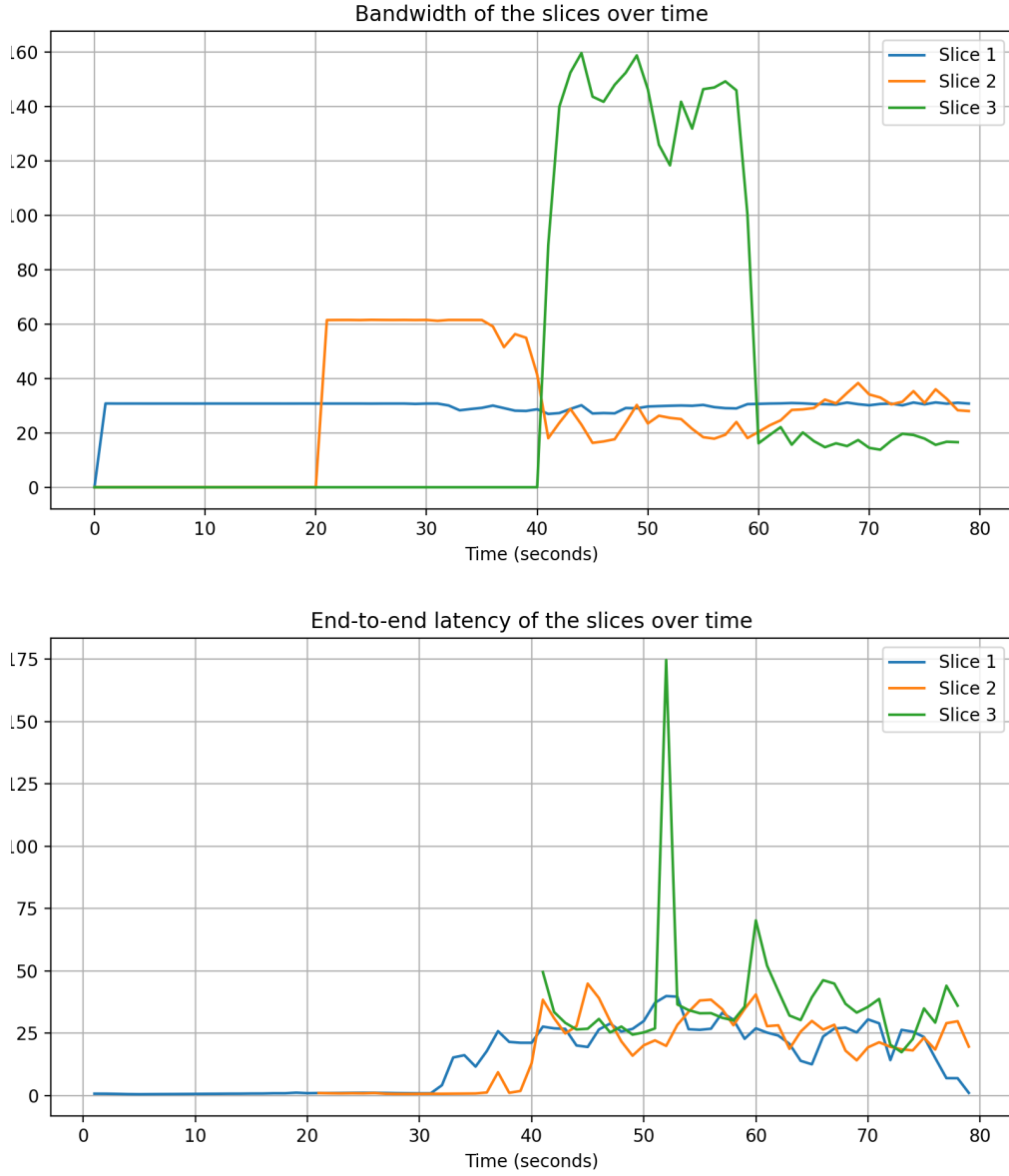


Figure 4.5: Third experiment results

not consistent throughout the experiments, and may point to some misconfiguration, incompatibility or general development issues with the tool.

This is confirmed, and even more evident, in the last experiment conducted, whose results are shown in Figure 4.6, while the flows generated are reassumed in Table 4.13. In this case, the results obtained are not consistent at all with every single experiment conducted so far, and thus they are probably due to some issue with the tool itself.

As it was with the third experiment, the volume of the flows is lower than the one used in the first two experiments (the iPerf3 ones), as MGen struggled to generate higher

Flow	Volume
Mission-critical (1)	16 Mbps
Mission-critical (2)	16 Mbps
URLLC (1)	32 Mbps
URLLC (2)	32 Mbps
Background traffic	256 Mbps

Table 4.12: Third experiment flows configuration

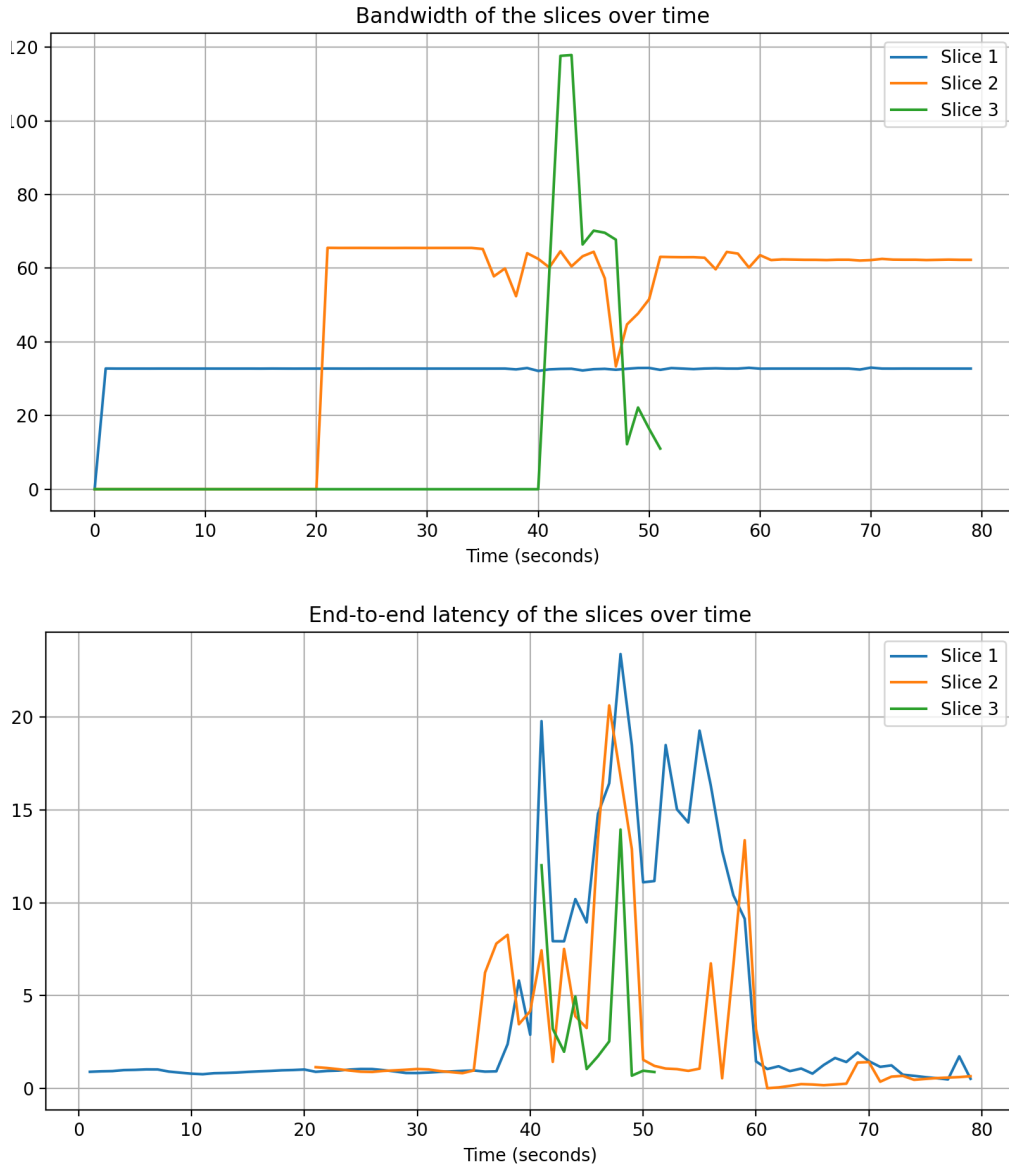


Figure 4.6: Fourth experiment results

Flow	Volume
Mission-critical (1)	16 Mbps
Mission-critical (2)	16 Mbps
URLLC (1)	32 Mbps
URLLC (2)	32 Mbps
Background traffic	256 Mbps

Table 4.13: Fourth experiment flows configuration

volumes of traffic.

In particular, in this case, we failed to gather more results from the background traffic later than second 50, and even before that the throughput of this flow itself was not stable at all and dropping almost immediately. Latency, instead, is more or less as expected, at least from the background traffic, again, as after 50 seconds the measurements were not available anymore.

In general, we found the use of MGen to be quite problematic and not very reliable.

4.2.1 A comparison with Linux Traffic Control

To propose a comparison against a more widespread, and more efficient, alternative, we decided to test the performance of a similarly-configured network using Linux **Traffic Control** (TC), which is a tool used to configure the **kernel** packet scheduler. In fact, while our P4-based solution is based, at least in the current state, on a software, **userspace**, switch, TC is a part of the Linux kernel, therefore should provide better performance, as it is more optimized and has direct access to the kernel’s networking stack. We will thus try to replicate the same scenario and configuration as the one used in the experiments shown so far, and with the same methodology, to provide a fair comparison.

What is going to be described is the configuration used on the egress Linux interface of the router machine, which is the one connected to the UPF machines in our testbed. The ingress interface, instead, is left unconfigured, as it is not needed for our purposes (remember that the P4 prototype proposed does not manage traffic when going back towards the gNodeBs). The configuration used for TC is based on the **prio** queuing discipline (qdisc), which is a simple and effective way to implement priority-based scheduling. It is based on the concept of **bands**, which are queues with different priorities, and it allows to classify packets into different bands based on their priority. In our case, we will define eight bands (but use only three of them) to be as similar as possible to the P4 implementation, where the Traffic Manager did have eight queues available (but with only three of them used, too). To each used band a **tbf** (Token Bucket Filter) qdisc is attached, which implements rate limiting and latency management: the effective parameters used to configure these latter qdiscs are shown in Table 4.14. Again, to the same egress interface a **flower** filter (classifier) is used to classify packets based on their destination IP address (not the VLAN ID, for a faster implementation) and to assign them to the appropriate queue (one of the bands of the **prio** qdisc).

To summarize, this setup should provide a similar behavior as the P4 implementation, with up to eight queues available, but with the lack of VLAN-based classification, which is supported by TC but would have required a more complex setup, and thus more time to be implemented and tested.

Queue	Max Rate	Burst Size	Latency
Mission-critical	150 Mbit	2 Mbit	2 ms
URLLC	3 Gbit	5 Mbit	50 ms
Background traffic	80 Mbit	2 Mbit	200 ms

Table 4.14: Parameters used for the Token Bucket Filter qdiscs

We will provide the results of two experiments conducted. The first one is shown in Figure 4.7, while the flows generated are reassumed in Table 4.15.

Flow	Volume
Mission-critical (1)	250 Mbps
Mission-critical (2)	250 Mbps
URLLC (1)	400 Mbps
URLLC (2)	400 Mbps
Background traffic	1 Gbps

Table 4.15: First TC experiment flows configuration

The tool used to generate the traffic in this case is **MGen**. It is possible to observe that the results obtained here are quite different from the ones obtained with the P4 implementation: first of all, from a latency point of view, after the activation of the Network Slicing mechanisms, the latency of the various flows only degrade, without any improvement; while the background traffic is heavily affected by the mechanism, as expected, the mission-critical and URLLC flows do not benefit from this configuration and show a latency that is higher than the one obtained without any mechanism at all (before second 60).

Moreover, the throughput of the slices after the activation show the desired effect, but not the desired values: the background traffic still uses a lot of bandwidth, more than the one allocated to it; anyway, the mission-critical and URLLC flows seem to be able to recover from the temporary congestion, at least by a certain extent and with a limited stability.

The second experiment conducted is shown in Figure 4.8, while the flows generated are reassumed in Table 4.16. This time, the traffic is generated with **iPerf3**, which has proved to be more reliable than MGen.

This time, the background traffic is set to 2 Gbps, to see if the TC mechanism is able to handle a higher volume of data and to provide the expected performance. Even though there is almost no congestion, the activation of the mechanism provoke a degradation of both latency and throughput for every flow, and the results obtained are not consistent

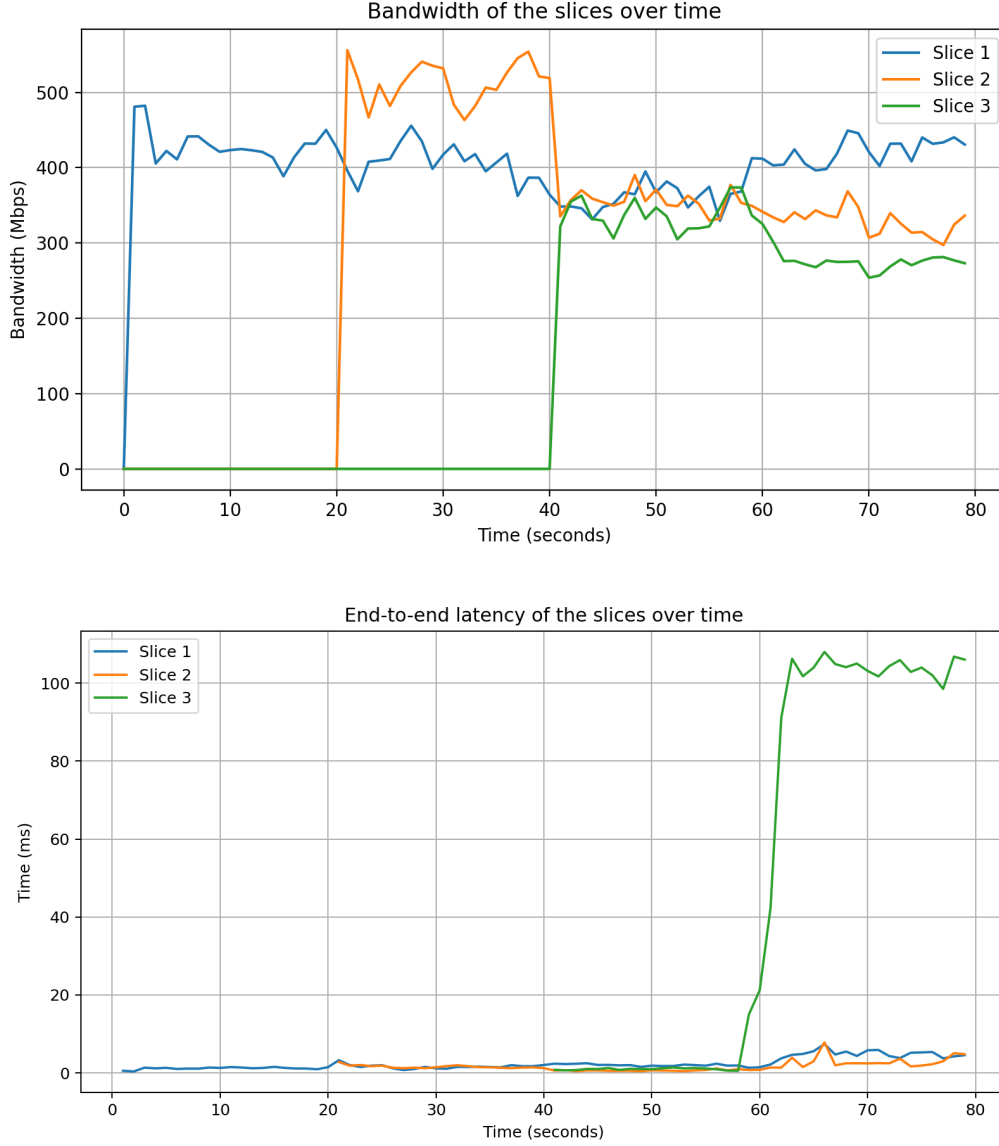


Figure 4.7: First TC experiment results

with the ones obtained in the previous experiments, and thus they are probably due to some misconfiguration in the TC setup. However, the lower throughput of the slices is expected: every slice is assigned lower resources than the one requested, therefore their throughput is going to be limited.

Flow	Volume
Mission-critical (1)	150 Mbps
Mission-critical (2)	150 Mbps
URLLC (1)	300 Mbps
URLLC (2)	300 Mbps
Background traffic	2 Gbps

Table 4.16: Second TC experiment flows configuration

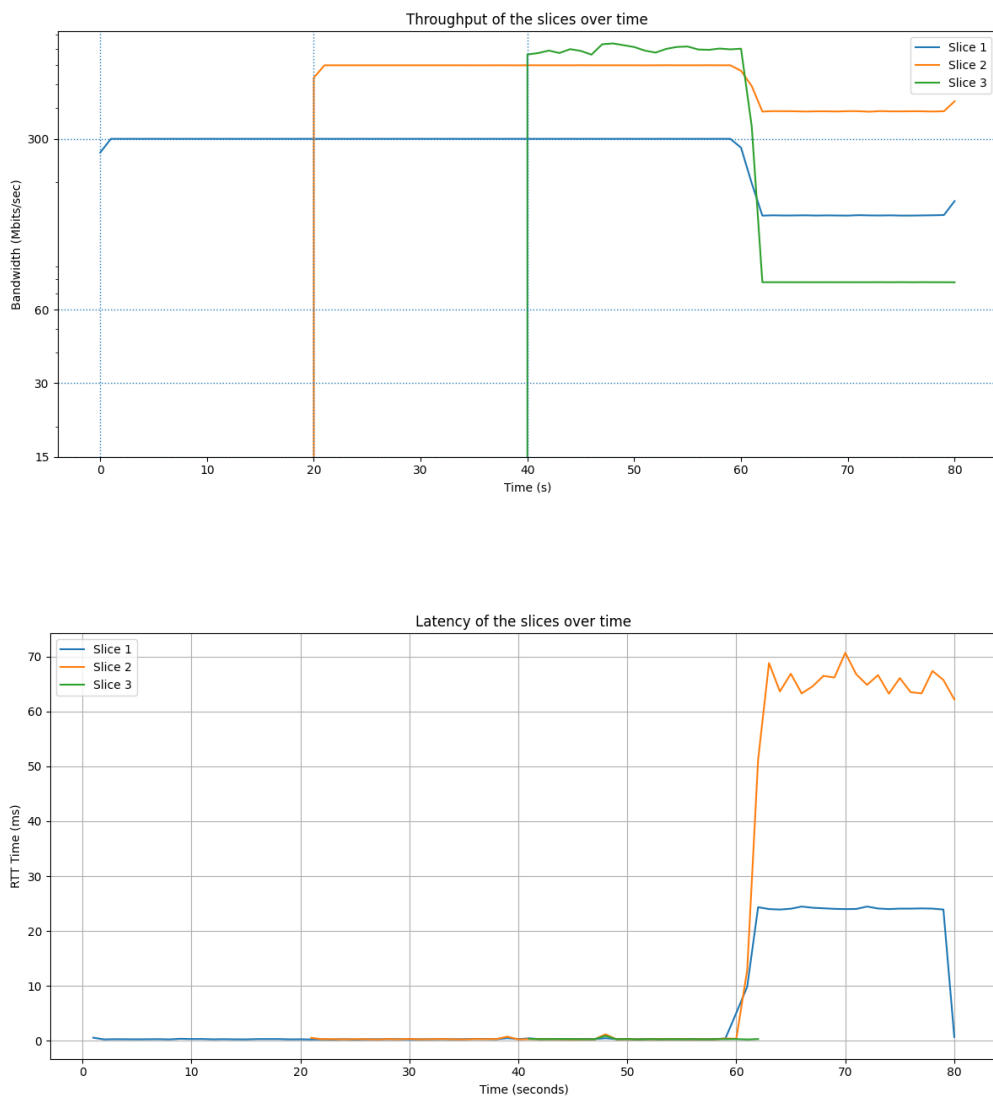


Figure 4.8: Second TC experiment results

Chapter 5

Conclusions and Future Work

Network Slicing is one of the key features of 5G networks and is expected to improve the performance and the flexibility of the modern infrastructures and technologies. In this thesis, we have explored the implementation of Network Slicing mechanisms in a P4-based environment, showing how it is possible to leverage the programmability of the Data Plane to achieve the desired results. The implementation has been tested in a real testbed, showing promising results in terms of latency and throughput for the various slices, even in presence of congestion. Moreover, we have compared our solution with a more traditional one, based on Linux Traffic Control, showing how the P4-based approach is able to provide better performance, at least in relative terms, and more customization possibilities. However, the introduction of P4 devices in an industrial context is still limited by various factors, such as the cost of the hardware and the lack of maturity of the software ecosystem. While there are some P4 hardware switches available on the market, they are still quite expensive and not widely adopted, thus limiting their use in production environments. Moreover, the software ecosystem is still evolving, and there are not many tools and libraries available for P4 development and deployment. This means that, while P4 provides a lot of flexibility and programmability, it also requires a significant investment in terms of time and resources to develop and maintain the necessary software components. Overall, while P4-based solutions show a lot of promise, there are still some challenges to overcome before they can be widely adopted in industrial contexts.

More specifically to our framework, the main limitation is represented by the use of a software, userspace, switch, which is not able to provide the performance required by modern networks. Moreover, more complex and advanced, maybe real-world, scenarios could be tested to further validate the proposed solution and its scalability.

The implementation of a complete and efficient Network Slicing framework within the N3 link of the 5G architecture could represent the first, big, step towards an end-to-end solution from the Radio Access Network until the Core Network, therefore succeeding in virtualizing multiple networks on top of a single physical infrastructure along the whole path: multiple applications could live on top of 5G networks, and this would represent a major milestone for the development of scalable and interconnected services, improving the lives of millions of people.

Future works could explore the integration of the proposed solution with another P4

software switch, known as NIKKS (Native In-Kernel Software Switch), which is a P4 implementation that runs directly in the Linux kernel, thus providing better performance than BMv2. Of course, hardware targets could be explored as well, to fully exploit the potential of P4 and to provide a more realistic, real-world, evaluation.

Moreover, a mechanism for the management of the minimum bandwidth guaranteed to each slice could be implemented, but it has been chosen to leave it out of the scope of this thesis, so that it could be implemented in an external component (with respect to the Orchestrator) to provide more flexibility.

Finally, a capabilities negotiation submodule within the Control Plane can represent an improvement for scalability at deployment time, allowing for the presence of heterogeneous P4 devices with different structure and capacities. This novelty should exploit the already existing API, extending it, respecting backwards compatibility.

Concluding, our framework represents a first step towards the implementation of a complete Network Slicing mechanism in the Transport layer, and it shows how P4 can be used to provide the desired performance and flexibility. However, there are still some challenges to overcome before it can be widely adopted in industrial contexts, and further works are needed to fully explore its potential.

Appendix A

The P4 programming language

P4 is a domain-specific programming language designed to program network devices, such as switches and routers, at the data plane level. It allows developers to define how packets are processed, enabling the creation of custom protocols and behaviors in network devices. P4 is protocol-independent, meaning it can be used to implement various protocols and functionalities without being tied to a specific protocol stack. This flexibility makes P4 suitable for a wide range of networking applications, including network monitoring, traffic engineering, and custom packet processing.

P4 programs are typically compiled into low-level representations that can be executed on programmable network devices, such as those based on the Tofino architecture or other P4-compatible hardware. The language provides constructs for defining packet headers, parsing logic, match-action tables, and control flow, allowing developers to specify how packets should be processed at each stage of the pipeline. P4 has gained popularity in the networking community due to its ability to enable rapid prototyping and deployment of new networking features, as well as its potential to improve network performance and flexibility.

In this Appendix, we will explore the P4 programming language in detail, focusing on its syntax and semantics.

A.1 Language syntax and semantics

Preprocessing directives in P4 are used to include files, define macros, and control the compilation process. The syntax for preprocessing directives in P4 is similar to that of C/C++ languages: even further, a P4 compiler should be able to parse and interpret C preprocessing directives, at least the ones from the set that will be provided below. The directives are typically placed at the beginning of the P4 source file and are processed before the actual compilation of the P4 program begins. The following are some common preprocessing directives used in P4:

- `#include <filename>` - This directive is used to include the contents of another file into the current P4 source file.
- `#define MACRO_NAME value` - This directive defines a macro with the specified name

and value. The macro can be used later in the P4 source code to represent the defined value.

- **#undef** *MACRO_NAME* - This directive undefines a previously defined macro, removing its definition from the P4 source code.
- **#if** *condition* - This directive checks if the specified condition is true. If it is true, the code following this directive will be included in the compilation process.
- **#else** - This directive is used in conjunction with **#if** to specify an alternative code block that will be included in the compilation process if the condition in the **#if** directive is false.
- **#elif** *condition* - This directive is used to specify an additional condition to check if the previous **#if** or **#else** condition was false. If the condition is true, the code following this directive will be included in the compilation process.
- **#ifdef** *MACRO_NAME* - This directive checks if the specified macro is defined. If it is defined, the code following this directive will be included in the compilation process.
- **#ifndef** *MACRO_NAME* - This directive checks if the specified macro is not defined. If it is not defined, the code following this directive will be included in the compilation process.
- **#endif** - This directive marks the end of a conditional compilation block that started with **#ifdef** or **#ifndef**.

P4 keywords use only ASCII characters.

The lexer recognizes the following terminal symbols:

- **IDENTIFIER** - A sequence of letters, digits, and underscores that starts with a letter or underscore. Identifiers are used to name variables, functions, and other entities in P4 programs.
- **TYPE_IDENTIFIER** - identifier that denotes a type name.
- **INTEGER** - an integer literal.
- **DONTCARE** - a single underscore character (`_`), used to match any value in a match-action table or elsewhere.
- **KEYWORD** - keywords in P4, such as **RETURN**.

The listings that will be presented in this Appendix will use the BNF notation, as they are grammar productions (unless otherwise specified in the text).

```
1 nonTypeName
2   : IDENTIFIER
3   | APPLY
4   | KEY
5   | ACTIONS
6   | STATE
```

```
7      | ENTRIES
8      | TYPE
9      ;
10
11 name
12     : nonTypeName
13     | TYPE_IDENTIFIER
14     ;
```

Listing A.1: P4 identifiers

Comments in P4 can be single-line or multi-line, similar to C/C++ languages. Single-line comments start with `//` and continue until the end of the line, while multi-line comments are enclosed between `/*` and `*/`. The lexer recognizes these comments and ignores them during the compilation process, and nested comments are not supported. What is supported, though, is the use of Javadoc-style comments.

A.1.1 Literals

Literals in P4 can be of different types, such as integer literals, string literals, and boolean literals. The lexer recognizes these literals and categorizes them accordingly.

Booleans are the constants `true` or `false`.

Integers, instead, by default are expressed in decimal notation, but they can also be expressed in other notations (binary, octal or hexadecimal) by using the according prefix (respectively, `0b`, `0o` and `0x`). Moreover, to express the size in bits of the integer, a suffix can be used composed by an unsigned number followed by a signedness indicator (`w` for unsigned, `s` for signed). Therefore, some examples of integer literals are:

```
1 32w255          // a 32-bit unsigned number with value 255
2 32w0d255        // same value as above
3 32w0xFF          // same value as above
4 32s0xFF          // a 32-bit signed number with value 255
5 8w0b10101010    // an 8-bit unsigned number with value 0xAA
6 8w0b_1010_1010  // same value as above
7 8w170           // same value as above
8 8s0b1010_1010   // an 8-bit signed number with value -86
9 16w0377         // 16-bit unsigned number with value 377 (not 255!)
10 16w0o377       // 16-bit unsigned number with value 255 (base 8)
```

Listing A.2: P4 integer literals

Strings in P4 are enclosed in double quotes (`"`), and can contain any sequence of characters, including escape sequences. They are terminated by the first unescaped double quote. No operations nor processing nor verification are performed on the string and are passed as-they-are to the backend compiler.

A.1.2 Program structure

A P4 program consists of a sequence of declarations, which can include type definitions, control blocks, tables, and other constructs. The program structure is defined by the following grammar:

```
1 p4program
2   : /* empty */
3   | p4program declaration
4   | p4program ';' /* empty declaration */
5   ;
6
7 declaration
8   : constantDeclaration
9   | externDeclaration
10  | actionDeclaration
11  | parserDeclaration
12  | typeDeclaration
13  | controlDeclaration
14  | instantiation
15  | errorDeclaration
16  | matchKindDeclaration
17  | functionDeclaration
18  ;
```

Listing A.3: P4 program structure

Scopes are used to define the visibility of variables and functions in P4 programs. The scope of a variable or function is determined by its declaration and the context in which it is used. The following constructs provide scopes in P4, apart from the global one:

- Derived type declarations (struct, header, header_union, enum).
- parser, table, control and action blocks.
- Declarations with type variables, such as in Listing A.4.
- Block statements.

```
1 extern E<H>(/* parameters omitted */) { /* body omitted */ } //
   scope of H ends here.
```

Listing A.4: P4 extern declaration

All stateful elements (e.g. `table` and `extern` objects) must be allocated through instantiation statements. Moreover, as parsers, controls and packages may contain stateful elements, they must be instantiated as well (and are thus treated as stateful elements).

A.1.3 Calling convention

Within the P4 language, the calling convention (that is, the invocation method) adopted is the **call by copy-in/copy-out**.

In the signature of a function each parameter can be tagged with a **in**, **out** or **inout** keyword, which determines how the parameter is passed to the function. The **in** keyword indicates that the parameter is read-only, the **out** keyword indicates that the parameter is write-only (therefore, similar to a l-value and it is uninitialized at function invocation), and the **inout** keyword indicates that the parameter can be both read and written to.

When a function is invoked, the parameters' value is copied into the function's **in** variables. At the end of the function call, the values of the **out** and **inout** parameters are copied back to the original variables.

As **extern** objects can be arbitrarily extended, invasive and powerful, this semantics avoids a situation in where an extern object has references to a P4 program object.

To provide a clear and standardized way to manage corner cases when passing parameters, P4 has made the following choices:

1. Arguments are evaluated from left to right as they appear in the function call expression.
2. If a parameter has a default value and no corresponding argument is supplied, the default value is used as an argument.
3. For each **out** and **inout** argument the corresponding l-value is saved (so it cannot be changed by the evaluation of the following arguments). This is important if the argument contains indexing operations into a header stack.
4. The value of each argument is saved into a temporary.
5. The function is invoked with the temporaries as arguments. We are guaranteed that the temporaries that are passed as arguments are never aliased to each other, so this “generated” function call can be implemented using call-by-reference if supported by the architecture.
6. On function return, the temporaries that correspond to **out** or **inout** arguments are copied in order from left to right into the l-values saved in step 2.

A.1.4 Data types

P4₁₆ is a statically typed language, meaning that the type of each variable must be known at compile time.

The base types supported in P4 are the ones shown in Table A.1; the grammar used to define them is shown in Listing A.5.

```
1 baseType
2   : BOOL
3   | ERROR
4   | BIT
```

```

5      | INT
6      | STRING
7      | BIT '<' INTEGER '>'
8      | INT '<' INTEGER '>'
9      | VARBIT '<' INTEGER '>'
10     | BIT '<' '(' expression ')' '>'
11     | INT '<' '(' expression ')' '>'
12     | VARBIT '<' '(' expression ')' '>'
13     ;

```

Listing A.5: P4 base types grammar

Type	Description
<code>void</code>	Represents the absence of a value. Limited in use cases.
<code>error</code>	Represents error in a target-independent way.
<code>string</code>	A compile-time string constant.
<code>match_kind</code>	Describes the implementation of a table lookup.
<code>bool</code>	A boolean value (true or false).
<code>int</code>	A variable-length constant integer.
<code>bit<></code>	Bit-strings of fixed width.
<code>int<></code>	Fixed-width signed integers (two's complement).
<code>varbit<></code>	Variable-length bit-strings with a maximum length.

Table A.1: Base types in P4

The P4 core library contains a set of predefined `match_kind` types, although it is possible to define custom ones. The predefined match kinds are:

- `exact` - Exact match on a field.
- `ternary` - Ternary match on a field, allowing for wildcard bits.
- `lpm` - Longest prefix match on a field.

The choice of implementing the `varbit` type is motivated by the need to handle variable-length fields, such as those found in the options of the IPv4 header.

Given that a P4 program is to be deployed on a specific target, this latter means that the target must support the specific data types used in the program: this is not always the case, as some targets may not support all the data types defined in P4 or present some limitations on their usage (e.g. `bit<4342354>` may not be supported by some targets, as it is too large for them).

P4 allows for the definition of custom data types, which can be used to create more complex data structures (**derived data types**). These custom types can be defined using different constructors:

- `enum`

- header
- header stacks
- struct
- header unions
- tuple
- extern
- parser
- control
- package

Moreover, the **typedef** instruction can be used to create an alias for an existing type, allowing for more readable code.

P4 presents a default value for many data types, and they are reported in Table A.2.

Type	Default Value
int , bit<> , int<>	0
bool	false
error	error.NoError
string	“” (empty string)
varbit<>	String of zero bits
enum	First enumerator
header	invalid
Header stacks	Every header invalid and nextIndex set to 0
header_union	All union elements are invalid
struct	Every field set to its default value
tuple	Same as struct

Table A.2: Default values for P4 data types

A.1.5 Expressions

Bit strings

For bit strings, the following operators are defined:

- Equality (==) and inequality (!=) operators, which compare two same-length bit strings for equality or inequality. The result is a Boolean.
- Unsigned comparisons (<, <=, >, >=), which compare two same-length bit strings as unsigned integers. The result is a Boolean.

- Negation ($-$), obtained by subtracting the bit string from 2^W , where W is the width of the bit string. The result is an unsigned bit string of the same width.
- Unary plus ($+$), which results in a no-op.
- Addition ($+$), which adds two same-length bit strings as unsigned integers. The result is a same-length bit string, truncated if necessary. Associative.
- Subtraction ($-$), which subtracts two same-length bit strings as unsigned integers. The result is a same-length bit string obtained by the sum of the first operand and the negation of the second operand.
- Multiplication ($*$), which multiplies two same-length bit strings as unsigned integers. The result is a same-length bit string, truncated if necessary. Some architectures may impose limitations, such as multiplication by only powers of two (e.g. Simple Switch (gRPC)).
- Bitwise AND ($\&$), which performs a bitwise AND operation on two same-length bit strings. The result is a same-length bit string.
- Bitwise OR ($|$), which performs a bitwise OR operation on two same-length bit strings. The result is a same-length bit string.
- Bitwise XOR ($\wedge$), which performs a bitwise XOR operation on two same-length bit strings. The result is a same-length bit string.
- Bitwise complement ($\sim$), which performs a bitwise NOT operation on a bit string. The result is a same-length bit string.
- Extraction of a set of contiguous bits ($[m:l]$), which extracts a set of contiguous bits from a bit string, where m and l are positive and $m \geq l$. The result is a bit string of length $m - l + 1$. Also, $0 \leq l < m < W$ and $l \leq m < W$ must be both true. A slice can also be used as a l-value.
- Logical shift left and right ($\ll, \gg$), which shifts a bit string left or right by a specified number of bits. The result is a same-length bit string, with the bits shifted in from the left or right, respectively. The shift amount must be a non-negative integer.
- Concatenation ($++$), which concatenates two bit strings of arbitrary length. The result is a bit string of length equal to the sum of the lengths of the two operands, where the most relevant bits are taken from the left operand.

Fixed-width integers

For integers (`int<>`) the same operators for bit strings are defined, with the addition of the following:

- Arithmetic shift left and right ($\ll, \gg$), which shifts a signed integer left or right by a specified number of bits. The result is a same-length integer, with the bits shifted in from the left or right, respectively. The shift amount must be a non-negative integer.

- Logical shift is *not* defined.
- Concatenation (`++`), in case of signed integers, takes the sign of the result from the left operand.

Arbitrary-precision integers

For arbitrary-precision integers (`int`), the same operators for fixed-width integers are defined, with the addition of the following:

- Truncating integer division (`/`), which divides two positive integers and returns the quotient as an integer.
- Modulo (`%`), which divides two positive integers and returns the remainder as an integer.
- Bitwise operations are not supported.

Variable-length bit strings

For variable-length bit strings (`varbit<>`), only the two following operations are defined:

- Assignment to another varbit variable, given that they have the same static width.
- Comparison with another varbit variable, given that they have the same static width. The result is a Boolean, **true** *iff* they have the same dynamic width and all the bits up to the dynamic width are equal.

A.1.6 Statements

The P4 language allows for a variety of statements within its programs, which can be used to control the flow of execution and manipulate data. The following are the key types of statements in P4:

- **Assignment statements** - Used to assign values to variables or fields. They can be simple assignments or compound assignments, such as incrementing a variable.
- **Empty statements** - Denoted by `;` (semicolon), these statements do nothing and are often used as placeholders or to indicate the end of a block. They therefore behave as a no-op.
- **Block statements** - Enclosed by curly braces (`{ }`), these statements group multiple statements together, allowing for structured control flow. They can contain any number of statements, including other block statements.
- **Return statements** - It terminates immediately the execution of the current function and returns control to the caller. It must return a value, which must match the function's return type: therefore, it cannot be used in parsers or in other functions that do not return a value.

- **Exit statements** - It terminates immediately the execution of all the blocks currently executing. Not allowed in parsers nor functions.
- **Conditional statements** - Used to execute a block of code based on a condition.
- **Switch statements** - There are two kinds of switch statements: the first one is used to select among a set of possible outcomes after an `apply()` operation on a table, while the second is the classic switch value-based statement.

There are **no loops** in P4, as the language is designed to be deterministic and predictable, which is crucial for networking applications.

A.2 Complete grammar

The complete grammar of the P4 language is defined in Listing A.6, using the Yacc/Bison notation. Note that different features of the P4 language were not provided in Section A.1, as they are not relevant to the purposes of this thesis.

```
1 p4program
2   : /* empty */
3   | p4program declaration
4   | p4program ';' /* empty declaration */
5   ;
6
7 declaration
8   : constantDeclaration
9   | externDeclaration
10  | actionDeclaration
11  | parserDeclaration
12  | typeDeclaration
13  | controlDeclaration
14  | instantiation
15  | errorDeclaration
16  | matchKindDeclaration
17  | functionDeclaration
18  ;
19
20 nonTypeName
21   : IDENTIFIER
22   | APPLY
23   | KEY
24   | ACTIONS
25   | STATE
26   | ENTRIES
27   | TYPE
28   ;
29
30 name
31   : nonTypeName
32   | TYPE_IDENTIFIER
```

```
33     ;
34
35 nonTableKwName
36     : IDENTIFIER
37     | TYPE_IDENTIFIER
38     | APPLY
39     | STATE
40     | TYPE
41     ;
42
43 optAnnotations
44     : /* empty */
45     | annotations
46     ;
47
48 annotations
49     : annotation
50     | annotations annotation
51     ;
52
53 annotation
54     : '@' name
55     | '@' name '(' annotationBody ')'
56     | '@' name '[' structuredAnnotationBody ']'
57     ;
58
59 parameterList
60     : /* empty */
61     | nonEmptyParameterList
62     ;
63
64 nonEmptyParameterList
65     : parameter
66     | nonEmptyParameterList ',' parameter
67     ;
68
69 parameter
70     : optAnnotations direction typeRef name
71     | optAnnotations direction typeRef name '=' expression
72     ;
73
74 direction
75     : IN
76     | OUT
77     | INOUT
78     | /* empty */
79     ;
80
81 packageTypeDeclaration
82     : optAnnotations PACKAGE name optTypeParameters
```

```
83      '(' parameterList ')'
84      ;
85
86 instantiation
87      : typeRef '(' argumentList ')' name ';'
88      | annotations typeRef '(' argumentList ')' name ';'
89      | annotations typeRef '(' argumentList ')' name '='
90      | typeRef '(' argumentList ')' name '=' objInitializer ';'
91      ;
92
93 objInitializer
94      : '{' objDeclarations '}'
95      ;
96
97 objDeclarations
98      : /* empty */
99      | objDeclarations objDeclaration
100     ;
101
102 objDeclaration
103     : functionDeclaration
104     | instantiation
105     ;
106
107 optConstructorParameters
108     : /* empty */
109     | '(' parameterList ')'
110     ;
111
112 dotPrefix
113     : '.'
114     ;
115
116 /***** PARSE
117     *****/
118
119 parserDeclaration
120     : parserTypeDeclaration optConstructorParameters
121     /* no type parameters allowed in the parserTypeDeclaration
122      */
123     '{' parserLocalElements parserStates '}'
124     ;
125
126 parserLocalElements
127     : /* empty */
128     | parserLocalElements parserLocalElement
129     ;
130
131 parserLocalElement
```

```
130      : constantDeclaration
131      | variableDeclaration
132      | instantiation
133      | valueSetDeclaration
134      ;
135
136 parserTypeDeclaration
137     : optAnnotations PARSE name optTypeParameters '('
138       parameterList ')'
139     ;
140
141 parserStates
142     : parserState
143     | parserStates parserState
144     ;
145
146 parserState
147     : optAnnotations STATE name '{' parserStatements
148       transitionStatement '}'
149     ;
150
151 parserStatements
152     : /* empty */
153     | parserStatements parserStatement
154     ;
155
156 parserStatement
157     : assignmentOrMethodCallStatement
158     | directApplication
159     | parserBlockStatement
160     | constantDeclaration
161     | variableDeclaration
162     | emptyStatement
163     | conditionalStatement
164     ;
165
166 parserBlockStatement
167     : optAnnotations '{' parserStatements '}'
168     ;
169
170 transitionStatement
171     : /* empty */
172     | TRANSITION stateExpression
173     ;
174
175 stateExpression
176     : name ';'
177     | selectExpression
178     ;
```

```
178 selectExpression
179     : SELECT '(' expressionList ')' '{' selectCaseList '}'
180     ;
181
182 selectCaseList
183     : /* empty */
184     | selectCaseList selectCase
185     ;
186
187 selectCase
188     : keysetExpression ':' name ';'
189     ;
190
191 keysetExpression
192     : tupleKeysetExpression
193     | simpleKeysetExpression
194     ;
195
196 tupleKeysetExpression
197     : "(" simpleKeysetExpression "," simpleExpressionList ")"
198     | "(" reducedSimpleKeysetExpression ")"
199     ;
200
201 simpleExpressionList
202     : simpleKeysetExpression
203     | simpleExpressionList ',' simpleKeysetExpression
204     ;
205
206 reducedSimpleKeysetExpression
207     : expression "&&&" expression
208     | expression ".." expression
209     | DEFAULT
210     | "_"
211     ;
212
213 simpleKeysetExpression
214     : expression
215     | DEFAULT
216     | DONTCARE
217     | expression MASK expression
218     | expression RANGE expression
219     ;
220
221 valueSetDeclaration
222     : optAnnotations
223       VALUESET '<' baseType '>' '(' expression ')' name ';'
224     | optAnnotations
225       VALUESET '<' tupleType '>' '(' expression ')' name ';'
226     | optAnnotations
227       VALUESET '<' typeName '>' '(' expression ')' name ';'
228     ;
```

```

228     ;
229
230 /***** CONTROL *****/
231
232 controlDeclaration
233     : controlTypeDeclaration optConstructorParameters
234     /* no type parameters allowed in controlTypeDeclaration */
235     '{' controlLocalDeclarations APPLY controlBody '}'
236     ;
237
238 controlTypeDeclaration
239     : optAnnotations CONTROL name optTypeParameters
240     '(' parameterList ')'
241     ;
242
243 controlLocalDeclarations
244     : /* empty */
245     | controlLocalDeclarations controlLocalDeclaration
246     ;
247
248 controlLocalDeclaration
249     : constantDeclaration
250     | actionDeclaration
251     | tableDeclaration
252     | instantiation
253     | variableDeclaration
254     ;
255
256 controlBody
257     : blockStatement
258     ;
259
260 /***** EXTERN *****/
261
262 externDeclaration
263     : optAnnotations EXTERN nonTypeName optTypeParameters '{'
264     methodPrototypes '}'
265     | optAnnotations EXTERN functionPrototype ';'
266     ;
267
268 methodPrototypes
269     : /* empty */
270     | methodPrototypes methodPrototype
271     ;
272
273 functionPrototype
274     : typeOrVoid name optTypeParameters '(' parameterList ')'
275     ;
276
277 methodPrototype

```

```
277      : optAnnotations functionPrototype ';'
278      | optAnnotations TYPE_IDENTIFIER '(' parameterList ')' ';'
279      ;
280
281 /***** TYPES *****/
282
283 typeRef
284   : baseType
285   | typeName
286   | specializedType
287   | headerStackType
288   | tupleType
289   ;
290
291 namedType
292   : typeName
293   | specializedType
294   ;
295
296 prefixedType
297   : TYPE_IDENTIFIER
298   | dotPrefix TYPE_IDENTIFIER
299   ;
300
301 typeName
302   : prefixedType
303   ;
304
305 tupleType
306   : TUPLE '<' typeArgumentList '>'
307   ;
308
309 headerStackType
310   : typeName '[' expression ']'
311   | specializedType '[' expression ']'
312   ;
313
314 specializedType
315   : prefixedType '<' typeArgumentList '>'
316   ;
317
318 baseType
319   : BOOL
320   | ERROR
321   | STRING
322   | INT
323   | BIT
324   | BIT '<' INTEGER '>'
325   | INT '<' INTEGER '>'
326   | VARBIT '<' INTEGER '>'
```

```

327     | BIT '<' '(' expression ')' '>'
328     | INT '<' '(' expression ')' '>'
329     | VARBIT '<' '(' expression ')' '>'
330     ;
331
332 typeOrVoid
333     : typeRef
334     | VOID
335     | IDENTIFIER      // may be a type variable
336     ;
337
338 optTypeParameters
339     : /* empty */
340     | typeParameters
341     ;
342
343 typeParameters
344     : '<' typeParameterList '>'
345     ;
346
347 typeParameterList
348     : name
349     | typeParameterList ',' name
350     ;
351
352 realTypeArg
353     : DONTCARE
354     | typeRef
355     | VOID
356     ;
357
358 typeArg
359     : DONTCARE
360     | typeRef
361     | nonTypeName
362     | VOID
363     ;
364
365 realTypeArgumentList
366     : realTypeArg
367     | realTypeArgumentList COMMA typeArg
368     ;
369
370 typeArgumentList
371     : /* empty */
372     | typeArg
373     | typeArgumentList ',' typeArg
374     ;
375
376 typeDeclaration

```

```
377      : derivedTypeDeclaration
378      | typedefDeclaration
379      | parserTypeDeclaration ';'
380      | controlTypeDeclaration ';'
381      | packageTypeDeclaration ';'
382      ;
383
384 derivedTypeDeclaration
385     : headerTypeDeclaration
386     | headerUnionDeclaration
387     | structTypeDeclaration
388     | enumDeclaration
389     ;
390
391 headerTypeDeclaration
392     : optAnnotations HEADER name optTypeParameters '{'
393       structFieldList '}'
394     ;
395
396 headerUnionDeclaration
397     : optAnnotations HEADER_UNION name optTypeParameters '{'
398       structFieldList '}'
399     ;
400
401 structTypeDeclaration
402     : optAnnotations STRUCT name optTypeParameters '{'
403       structFieldList '}'
404     ;
405
406 structFieldList
407     : /* empty */
408     | structFieldList structField
409     ;
410
411 structField
412     : optAnnotations typeRef name ';'
413     ;
414
415 enumDeclaration
416     : optAnnotations ENUM name '{' identifierList '}'
417     | optAnnotations ENUM typeRef name '{'
418       specifiedIdentifierList '}'
419     ;
420
421 errorDeclaration
422     : ERROR '{' identifierList '}'
423     ;
424
425 matchKindDeclaration
426     : MATCH_KIND '{' identifierList '}'
```

```
423     ;
424
425 identifierList
426     : name
427     | identifierList ',' name
428     ;
429
430 specifiedIdentifierList
431     : specifiedIdentifier
432     | specifiedIdentifierList ',' specifiedIdentifier
433     ;
434
435 specifiedIdentifier
436     : name '=' initializer
437     ;
438
439 typedefDeclaration
440     : optAnnotations TYPEDEF typeRef name ';'
441     | optAnnotations TYPEDEF derivedTypeDeclaration name ';'
442     | optAnnotations TYPE typeRef name ';'
443     | optAnnotations TYPE derivedTypeDeclaration name ';'
444     ;
445
446 /***** STATEMENTS *****/
447
448 assignmentOrMethodCallStatement
449     : lvalue '(' argumentList ')' ';'
450     | lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'
451     | lvalue '=' expression ';'
452     ;
453
454 emptyStatement
455     : ';'
456     ;
457
458 returnStatement
459     : RETURN ';'
460     | RETURN expression ';'
461     ;
462
463 exitStatement
464     : EXIT ';'
465     ;
466
467 conditionalStatement
468     : IF '(' expression ')' statement
469     | IF '(' expression ')' statement ELSE statement
470     ;
471
```

```
472 // To support direct invocation of a control or parser without
      instantiation
473 directApplication
474     : typeName '.' APPLY '(' argumentList ')' ';'
475     ;
476
477 statement
478     : assignmentOrMethodCallStatement
479     | directApplication
480     | conditionalStatement
481     | emptyStatement
482     | blockStatement
483     | exitStatement
484     | returnStatement
485     | switchStatement
486     ;
487
488 blockStatement
489     : optAnnotations '{' statOrDeclList '}'
490     ;
491
492 statOrDeclList
493     : /* empty */
494     | statOrDeclList statementOrDeclaration
495     ;
496
497 switchStatement
498     : SWITCH '(' expression ')' '{' switchCases '}'
499     ;
500
501 switchCases
502     : /* empty */
503     | switchCases switchCase
504     ;
505
506 switchCase
507     : switchLabel ':' blockStatement
508     | switchLabel ':'
509     ;
510
511 switchLabel
512     : DEFAULT
513     | nonBraceExpression
514     ;
515
516 statementOrDeclaration
517     : variableDeclaration
518     | constantDeclaration
519     | statement
520     | instantiation
```

```
521      ;
522
523 /***** TABLES *****/
524 tableDeclaration
525     : optAnnotations TABLE name '{' tablePropertyList '}'
526     ;
527
528 tablePropertyList
529     : tableProperty
530     | tablePropertyList tableProperty
531     ;
532
533 tableProperty
534     : KEY '=' '{' keyElementList '}'
535     | ACTIONS '=' '{' actionList '}'
536     | optAnnotations CONST ENTRIES '=' '{' entriesList '}' /*
537       immutable entries */
537     | optAnnotations CONST nonTableKwName '=' initializer ';'
538     | optAnnotations nonTableKwName '=' initializer ';'
539     ;
540
541 keyElementList
542     : /* empty */
543     | keyElementList keyElement
544     ;
545
546 keyElement
547     : expression ':' name optAnnotations ';'
548     ;
549
550 actionList
551     : /* empty */
552     | actionList optAnnotations actionRef ';'
553     ;
554
555 actionRef
556     : prefixedNonTypeName
557     | prefixedNonTypeName '(' argumentList ')'
558     ;
559
560 entriesList
561     : entry
562     | entriesList entry
563     ;
564
565 entry
566     : keysetExpression ':' actionRef optAnnotations ';'
567     ;
568
```

```
569 /***** ACTION
      *****/
570
571 actionDeclaration
572     : optAnnotations ACTION name '(' parameterList ')'
      blockStatement
573     ;
574
575 /***** VARIABLES
      *****/
576
577 variableDeclaration
578     : annotations typeRef name optInitializer ';'
      | typeRef name optInitializer ';'
579     ;
580
581
582 constantDeclaration
583     : optAnnotations CONST typeRef name '=' initializer ';'
584     ;
585
586 optInitializer
587     : /* empty */
      | '=' initializer
588     ;
589
590
591 initializer
592     : expression
593     ;
594
595 /***** Expressions
      *****/
596
597 functionDeclaration
598     : functionPrototype blockStatement
599     ;
600
601 argumentList
602     : /* empty */
      | nonEmptyArgList
603     ;
604
605
606 nonEmptyArgList
607     : argument
      | nonEmptyArgList ',' argument
608     ;
609
610
611 argument
612     : expression
      | name '=' expression
613     | DONTCARE
614     ;
```

```

615     ;
616
617 kvList
618     : kvPair
619     | kvList ',' kvPair
620     ;
621
622 kvPair
623     : name '=' expression
624     ;
625
626 expressionList
627     : /* empty */
628     | expression
629     | expressionList ',' expression
630     ;
631
632
633 annotationBody
634     : /* empty */
635     | annotationBody '(' annotationBody ')'
636     | annotationBody annotationToken
637     ;
638
639 structuredAnnotationBody
640     : expressionList
641     | kvList
642     ;
643
644 annotationToken
645     : ABSTRACT
646     | ACTION
647     | ACTIONS
648     | APPLY
649     | BOOL
650     | BIT
651     | CONST
652     | CONTROL
653     | DEFAULT
654     | ELSE
655     | ENTRIES
656     | ENUM
657     | ERROR
658     | EXIT
659     | EXTERN
660     | FALSE
661     | HEADER
662     | HEADER_UNION
663     | IF
664     | IN

```

```

665      | INOUT
666      | INT
667      | KEY
668      | MATCH_KIND
669      | TYPE
670      | OUT
671      | PARSER
672      | PACKAGE
673      | PRAGMA
674      | RETURN
675      | SELECT
676      | STATE
677      | STRING
678      | STRUCT
679      | SWITCH
680      | TABLE
681      | TRANSITION
682      | TRUE
683      | TUPLE
684      | TYPEDEF
685      | VARBIT
686      | VALUESET
687      | VOID
688      | "_"
689      | IDENTIFIER
690      | TYPE_IDENTIFIER
691      | STRING_LITERAL
692      | INTEGER
693      | "&&&"
694      | ". ."
695      | "<<"
696      | "&&"
697      | "| |"
698      | "=="
699      | "!="
700      | ">="
701      | "<="
702      | "++"
703      | "+"
704      | "| + |"
705      | "-"
706      | "| - |"
707      | "*"
708      | "/"
709      | "%"
710      | "| "
711      | "&"
712      | "^"
713      | "~"
714      | "["

```

```
715      | "]"
716      | "{"
717      | "}"
718      | "<"
719      | ">"
720      | "!"
721      | ":"
722      | ","
723      | "?"
724      | "."
725      | "="
726      | ";"
727      | "@"
728      | UNKNOWN_TOKEN
729      ;
730
731 member
732     : name
733     ;
734
735 prefixedNonTypeName
736     : nonTypeName
737     | dotPrefix nonTypeName
738     ;
739
740 lvalue
741     : prefixedNonTypeName
742     | THIS
743     | lvalue '.' member
744     | lvalue '[' expression ']'
745     | lvalue '[' expression ':' expression ']'
746     ;
747
748 %left ','
749 %nonassoc '?'
750 %nonassoc ':'
751 %left '||'
752 %left '&&'
753 %left '==' '!='
754 %left '<' '>' '<=' '>='
755 %left '|'
756 %left '^'
757 %left '&'
758 %left '<<' '>>'
759 %left '++' '+' '-' '++' '--' '|+' '|-'
760 %left '*' '/' '%'
761 %right PREFIX
762 %nonassoc ']' '(' '['
763 %left '.'
764
```

```
765 // Additional precedences need to be specified
766
767 expression
768     : INTEGER
769     | TRUE
770     | FALSE
771     | THIS
772     | STRING_LITERAL
773     | nonTypeName
774     | dotPrefix nonTypeName
775     | expression '[' expression ']'
776     | expression '[' expression ':' expression ']'
777     | '{' expressionList '}'
778     | '{' kvList '}'
779     | '(' expression ')'
780     | '!' expression %prec PREFIX
781     | '~' expression %prec PREFIX
782     | '-' expression %prec PREFIX
783     | '+' expression %prec PREFIX
784     | typeName '.' member
785     | ERROR '.' member
786     | expression '.' member
787     | expression '*' expression
788     | expression '/' expression
789     | expression '%' expression
790     | expression '+' expression
791     | expression '-' expression
792     | expression '++' expression
793     | expression '--' expression
794     | expression '<<' expression
795     | expression '>>' expression
796     | expression '<=' expression
797     | expression '>=' expression
798     | expression '<' expression
799     | expression '>' expression
800     | expression '!=' expression
801     | expression '==' expression
802     | expression '&' expression
803     | expression '^' expression
804     | expression '|' expression
805     | expression '++' expression
806     | expression '&&' expression
807     | expression '||' expression
808     | expression '?' expression ':' expression
809     | expression '<' realTypeArgumentList '>' '(' argumentList ')'
810     | expression '(' argumentList ')'
811     | namedType '(' argumentList ')'
812     | '(' typeRef ')' expression
813 ;
814
```

```
815 nonBraceExpression
816     : INTEGER
817     | STRING_LITERAL
818     | TRUE
819     | FALSE
820     | THIS
821     | nonTypeName
822     | dotPrefix nonTypeName
823     | nonBraceExpression '[' expression ']'
824     | nonBraceExpression '[' expression ':' expression ']'
825     | '(' expression ')'
826     | '!' expression %prec PREFIX
827     | '~' expression %prec PREFIX
828     | '-' expression %prec PREFIX
829     | '+' expression %prec PREFIX
830     | typeName '.' member
831     | ERROR '.' member
832     | nonBraceExpression '.' member
833     | nonBraceExpression '*' expression
834     | nonBraceExpression '/' expression
835     | nonBraceExpression '%' expression
836     | nonBraceExpression '+' expression
837     | nonBraceExpression '-' expression
838     | nonBraceExpression '|+' expression
839     | nonBraceExpression '|-' expression
840     | nonBraceExpression '<<' expression
841     | nonBraceExpression '>>' expression
842     | nonBraceExpression '<=' expression
843     | nonBraceExpression '>=' expression
844     | nonBraceExpression '<' expression
845     | nonBraceExpression '>' expression
846     | nonBraceExpression '!=' expression
847     | nonBraceExpression '==' expression
848     | nonBraceExpression '&' expression
849     | nonBraceExpression '^' expression
850     | nonBraceExpression '|' expression
851     | nonBraceExpression '++' expression
852     | nonBraceExpression '&&' expression
853     | nonBraceExpression '||' expression
854     | nonBraceExpression '?' expression ':' expression
855     | nonBraceExpression '<' realTypeArgumentList '>' '('
      argumentList ')'
856     | nonBraceExpression '(' argumentList ')'
857     | namedType '(' argumentList ')'
858     | '(' typeRef ')' expression
859     ;
```

Listing A.6: P4 complete grammar

Appendix B

CoDel P4 implementation

This appendix contains the source code of the P4 implementation of the CoDel algorithm, which was evaluated in the software implementation described in Chapter 3. The final decision to not use CoDel was made after the evaluation of the traffic patterns in the scenario described in Chapter 1, which showed that the traffic was mostly unresponsive, and therefore CoDel would not have been effective in this case. However, the implementation is still provided here for completeness and as a reference. The original authors and maintainers of the code are Ralf Kundel and Nikolas Eller, and the code is publicly available on GitHub ([15]).

B.1 Original CoDel P4 implementation

The design choice taken in the original implementation is to use a single P4 control block for CoDel, which handles the entire algorithm in a single place. The control block is called in the egress P4 block, where the queue ID is determined through the setting of the `meta.codel.queue_id` field. The CoDel block is then responsible for managing the queue and applying the CoDel algorithm to the packets in that queue.

The original source code of the CoDel P4 implementation is available on GitHub ([15]), and it is structured as per Listing B.1.

```
1 #define SOJOURN_TARGET 5000
2 #define CONTROL_INTERVAL 48w100000
3 #define INTERFACE_MTU 1500
4 #define NO_QUEUE_ID 32w64
5
6 register<bit<32>>(NO_QUEUE_ID) r_drop_count;
7 register<bit<48>>(NO_QUEUE_ID) r_drop_time;
8 register<bit<32>>(NO_QUEUE_ID) r_last_drop_count;
9 register<bit<48>>(NO_QUEUE_ID) r_next_drop;
10 register<bit<1>>(NO_QUEUE_ID) r_state_dropping;
11
12 control c_codel(inout headers hdr, inout metadata meta, inout
    standard_metadata_t standard_metadata) {
13
```

```
14     action a_codel_control_law(bit<48> value) {
15         meta.codel.drop_next = meta.codel.time_now + value;
16         r_next_drop.write((bit<32>) meta.codel.queue_id,
17             (bit<48>) meta.codel.drop_next);
18     }
19     action a_codel_init() {
20         meta.codel.ok_to_drop = 1w0;
21         meta.codel.time_now = (bit<48>)
22             standard_metadata.enq_timestamp + (bit<48>)
23             standard_metadata.deq_timedelta;
24         meta.codel.new_drop_time = meta.codel.time_now +
25             CONTROL_INTERVAL;
26         r_state_dropping.read(meta.codel.state_dropping,
27             (bit<32>) meta.codel.queue_id);
28         r_drop_count.read(meta.codel.drop_cnt, (bit<32>)
29             meta.codel.queue_id);
30         r_last_drop_count.read(meta.codel.last_drop_cnt,
31             (bit<32>) meta.codel.queue_id);
32         r_next_drop.read(meta.codel.drop_next, (bit<32>)
33             meta.codel.queue_id);
34         r_drop_time.read(meta.codel.drop_time, (bit<32>)
35             meta.codel.queue_id);
36     }
37     action a_go_to_drop_state() {
38         mark_to_drop(standard_metadata);
39         r_state_dropping.write((bit<32>) meta.codel.queue_id,
40             (bit<1>) 1);
41         meta.codel.delta = meta.codel.drop_cnt -
42             meta.codel.last_drop_cnt;
43         meta.codel.time_since_last_dropping = meta.codel.time_now
44             - meta.codel.drop_next;
45         meta.codel.drop_cnt = 32w1;
46         r_drop_count.write((bit<32>) meta.codel.queue_id,
47             (bit<32>) 1);
48     }
49     table t_codel_control_law {
50         actions = {
51             a_codel_control_law;
52         }
53         key = {
54             meta.codel.drop_cnt: lpm;
55         }
56         size = 32;
57     }
58     apply {
59         a_codel_init();
```

```
51     if (standard_metadata.deq_timedelta < SOJOURN_TARGET) {
52         meta.codel.reset_drop_time = 1w1;
53     }
54
55     if (meta.codel.reset_drop_time == 1w1) {
56         r_drop_time.write((bit<32>) meta.codel.queue_id,
57             (bit<48>) 0);
58         meta.codel.drop_time = 48w0;
59     }
60     else {
61         if (meta.codel.drop_time == 48w0) {
62             r_drop_time.write((bit<32>) meta.codel.queue_id,
63                 (bit<48>) meta.codel.new_drop_time);
64             meta.codel.drop_time = meta.codel.new_drop_time;
65         }
66         else {
67             if (meta.codel.time_now >= meta.codel.drop_time) {
68                 meta.codel.ok_to_drop = 1w1;
69             }
70         }
71     }
72
73     if (meta.codel.state_dropping == 1w1) {
74         if (meta.codel.ok_to_drop == 1w0) {
75             r_state_dropping.write((bit<32>)
76                 meta.codel.queue_id, (bit<1>) 0); // leave
77                 drop state
78         }
79         else {
80             if (meta.codel.time_now >= meta.codel.drop_next) {
81                 mark_to_drop(standard_metadata);
82                 meta.codel.drop_cnt = meta.codel.drop_cnt
83                     + 32w1;
84                 r_drop_count.write((bit<32>)
85                     meta.codel.queue_id, (bit<32>)
86                     meta.codel.drop_cnt);
87                 t_codel_control_law.apply();
88             }
89         }
90     }
91     else {
92         if (meta.codel.ok_to_drop == 1w1) {
93             a_go_to_drop_state();
94             if (meta.codel.delta > 32w1 &&
95                 meta.codel.time_since_last_dropping <
96                 CONTROL_INTERVAL * 16) {
97                 r_drop_count.write((bit<32>)
98                     meta.codel.queue_id, (bit<32>)
99                     meta.codel.delta);
100                 meta.codel.drop_cnt = meta.codel.delta;
```

```
90         }
91         r_last_drop_count.write((bit<32>)
            meta.codel.queue_id, (bit<32>)
            meta.codel.drop_cnt);
92         t_codel_control_law.apply();
93     }
94 }
95 }
96 }
```

Listing B.1: Original CoDel P4 implementation

The macros here proposed are expressed in microseconds, while the original specification of the CoDel algorithm uses milliseconds. This is done to ensure that the algorithm, designed specifically for the Simple Switch (gRPC), which uses microseconds as clock unit, operates correctly in this environment. This algorithm, as specified by the macro `NO_QUEUE_ID`, supports up to 64 queues, which is sufficient for the scenario described in Chapter 1 but may be a limitation in other scenarios: changing the value of the macro allows to increase the number of queues supported by this algorithm without requiring any other modifications to the source code.

When a packet reaches Egress stage and the CoDel control block is applied to it, the queue ID is set in the metadata field and the relevant metadata fields are initialized (`a_codel_init()`, line 50). The algorithm then checks if the packet stood in the queue for less than the `SOJOURN_TARGET` time: if so, the `reset_drop_time` field is set to 1, and the drop time is reset to 0. Otherwise, the drop time for the current packet is calculated and, if the current time is greater than or equal to the drop time, the packet is marked for dropping. Now, if the queue is in the dropping state, depending on the value of the `ok_to_drop` field, the algorithm either leaves the dropping state or drops the packet and updates the drop count. If the queue is not in the dropping state, the value of the `ok_to_drop` field is checked: if it is set to 1, the queue enters the dropping state and updates the drop count; in the end, the control law table is applied to the queue, which updates the drop time for the next packet. In this case, the relevant table (`t_codel_control_law`) has its entries filled by the Control Plane, but as the values are fixed and do not change during the execution of the algorithm, they can also be directly written as constant entries.

In the end, it is worth noting that this first implementation of CoDel uses fixed values for some timings, such as the sojourn target (as already mentioned, 5 milliseconds) and the control interval (100 milliseconds), which is the time between two consecutive checks of the queue state. These values are not configurable and are hardcoded in the source code, which may limit the flexibility of the algorithm in different scenarios. However, while the latter choice (control interval) is not a problem, the first one is a limitation that will be addressed in Section B.2.

B.2 Modifications made

The original implementation was modified to fit the scenario described in Chapter 1, in particular to handle the concept of Network Slicing.

The main modification applied to the original implementation is the introduction of a new Match-Action Table to handle the latency requirements of the slices, thus not relying on a single macro for the sojourn time, but rather on a per-slice basis. The new table, `t_codel_sojourn_target`, and its associated action `assign_sojourn_target`, are defined as described in Listing B.2.

```
1 action assign_sojourn_target(bit<32> max_latency) {
2     meta.codel.sojourn_target = max_latency;
3 }
4
5 table t_codel_sojourn_target {
6     actions = {
7         assign_sojourn_target;
8     }
9     key = {
10         meta.slice_id: exact;
11         meta.subslice_id: exact;
12     }
13     default_action = assign_sojourn_target(200);
14 }
```

Listing B.2: Latency target per-slice assignment

This table is applied in the `apply` block of the CoDel control block, right after the initialization of the metadata fields, and its entries are given by the Control Plane, and in particular by the P4 controller of the Agent, which is instructed at a high-level from the Orchestrator. The check for the sojourn time is then modified to use the value stored in the new, custom, CoDel metadata field `meta.codel.sojourn_target`.

The other modifications made to the original implementation are related to the handling of the queue ID, which now does not represent anymore the actual queue ID, but rather the slice ID and the subslice ID of the packet (in particular, their hash) as described before. This allows for keeping the status of the queue of the slice even when the associated queue changes dynamically due to the Network Slicing mechanisms explained beforehand in Chapter 3.

Moreover, the `t_codel_control_law` table now uses constant entries, as explained in Section B.1.

Bibliography

- [1] Ibrahim Afolabi et al. “Network Slicing and Softwarization: A Survey on Principles, Enabling Technologies, and Solutions”. In: *IEEE Communications Surveys and Tutorials* 20.3 (2018), pp. 2429–2453. DOI: [10.1109/COMST.2018.2815638](https://doi.org/10.1109/COMST.2018.2815638).
- [2] University of Antwerp. *IDLab UAntwerp website*. URL: <https://www.uantwerpen.be/en/research-groups/idlab/>.
- [3] Divyashri Bhat et al. “Application-based QoE support with P4 and OpenFlow”. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2019. DOI: [10.1109/INFCOMW.2019.8845180](https://doi.org/10.1109/INFCOMW.2019.8845180).
- [4] Yan-Wei Chen et al. “P4-Enabled Bandwidth Management”. In: *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 2019. DOI: [10.23919/APNOMS.2019.8892909](https://doi.org/10.23919/APNOMS.2019.8892909).
- [5] Yan-Wei Chen et al. “P4-TINS: P4-Driven Traffic Isolation for Network Slicing With Bandwidth Guarantee and Management”. In: *IEEE Transactions on Network and Service Management*. Vol. 19. 3. 2022, pp. 3290–3303. DOI: [10.1109/TNSM.2022.3159232](https://doi.org/10.1109/TNSM.2022.3159232).
- [6] P4 Language Consortium. *Performance of the P4 Behavioral Model v2*. URL: <https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md>.
- [7] Leandro C. de Almeida et al. “DESiRED - Dynamic, Enhanced, and Smart iRED: A P4-AQM with Deep Reinforcement Learning and In-band Network Telemetry”. In: *Computer Networks* 244 (2024), p. 110326. ISSN: 1389-1286. DOI: <https://doi.org/10.1016/j.comnet.2024.110326>. URL: <https://www.sciencedirect.com/science/article/pii/S1389128624001580>.
- [8] Chamitha De Alwis et al. “A Survey on Network Slicing Security: Attacks, Challenges, Solutions and Research Directions”. In: *IEEE Communications Surveys and Tutorials* 26.1 (2024), pp. 534–570. DOI: [10.1109/COMST.2023.3312349](https://doi.org/10.1109/COMST.2023.3312349).
- [9] Xenofon Foukas et al. “Network Slicing in 5G: Survey and Challenges”. In: *IEEE Communications Magazine* 55.5 (2017), pp. 94–100. DOI: [10.1109/MCOM.2017.1600951](https://doi.org/10.1109/MCOM.2017.1600951).
- [10] Kerim Gokarslan, Yagmur Sabucu Sandal, and Tuna Tugcu. “Towards a URLLC-Aware Programmable Data Path with P4 for Industrial 5G Networks”. In: *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*. 2021. DOI: [10.1109/ICCWorkshops50388.2021.9473859](https://doi.org/10.1109/ICCWorkshops50388.2021.9473859).

- [11] Jose Gomez et al. “Improving TCP Fairness in Non-Programmable Networks Using P4-Programmable Data Planes”. In: *2024 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*. 2024, pp. 102–107. DOI: [10.1109/BlackSeaCom61746.2024.10646283](https://doi.org/10.1109/BlackSeaCom61746.2024.10646283).
- [12] Hasanin Harkous et al. “Virtual Queues for P4: A Poor Man’s Programmable Traffic Manager”. In: *IEEE Transactions on Network and Service Management*. Vol. 18. 3. 2021, pp. 2860–2872. DOI: [10.1109/TNSM.2021.3077051](https://doi.org/10.1109/TNSM.2021.3077051).
- [13] Wenji He et al. “A P4-Based Approach to Traffic Isolation and Bandwidth Management for 5G Network Slicing”. In: *Tsinghua Science and Technology*. Vol. 30. 1. 2025, pp. 171–185. DOI: [10.26599/TST.2024.9010020](https://doi.org/10.26599/TST.2024.9010020).
- [14] Dr. Juha Heinanen and Dr. Roch Guerin. *A Two Rate Three Color Marker*. RFC 2698. Sept. 1999. DOI: [10.17487/RFC2698](https://doi.org/10.17487/RFC2698). URL: <https://www.rfc-editor.org/info/rfc2698>.
- [15] Ralf Kundel et al. “P4-CoDel: Active Queue Management in Programmable Data Planes”. In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2018, pp. 1–4. DOI: [10.1109/NFV-SDN.2018.8725736](https://doi.org/10.1109/NFV-SDN.2018.8725736).
- [16] Osel Lhamo et al. “RED-SP-CoDel: Random early detection with static priority scheduling and controlled delay AQM in programmable data planes”. In: *Computer Communications* 214 (2024), pp. 149–166. ISSN: 0140-3664. DOI: <https://doi.org/10.1016/j.comcom.2023.11.026>. URL: <https://www.sciencedirect.com/science/article/pii/S014036642300422X>.
- [17] Xin Li et al. “Network Slicing for 5G: Challenges and Opportunities”. In: *IEEE Internet Computing* 21.5 (2017), pp. 20–27. DOI: [10.1109/MIC.2017.3481355](https://doi.org/10.1109/MIC.2017.3481355).
- [18] Yi-Bing Lin, Chien-Chao Tseng, and Ming-Hung Wang. “Effects of Transport Network Slicing on 5G Applications”. In: *Future Internet*. Vol. 13. 3. 2021. DOI: [10.3390/fi13030069](https://doi.org/10.3390/fi13030069).
- [19] Kathleen Nichols et al. *Controlled Delay Active Queue Management*. RFC 8289. Jan. 2018. DOI: [10.17487/RFC8289](https://doi.org/10.17487/RFC8289). URL: <https://www.rfc-editor.org/info/rfc8289>.
- [20] Jose Ordonez-Lucena et al. “Network Slicing for 5G with SDN/NFV: Concepts, Architectures, and Challenges”. In: *IEEE Communications Magazine* 55.5 (2017), pp. 80–87. DOI: [10.1109/MCOM.2017.1600935](https://doi.org/10.1109/MCOM.2017.1600935).
- [21] *P416 Language Specification*. version 1.2.2. P4 Language Consortium. May 2021.
- [22] *P4Runtime Specification*. version 1.3.0. P4 Language Consortium. July 2021. URL: <https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html>.
- [23] Francesco Paolucci et al. “Enhancing 5G SDN/NFV Edge with P4 Data Plane Programmability”. In: *IEEE Network*. Vol. 35. 3. 2021, pp. 154–160. DOI: [10.1109/MNET.021.1900599](https://doi.org/10.1109/MNET.021.1900599).
- [24] Francesco Paolucci et al. “Latency control in service chaining using P4-based data plane programmability”. In: *Computer Networks*. Vol. 216. 2022. DOI: [10.1016/j.comnet.2022.109227](https://doi.org/10.1016/j.comnet.2022.109227).

- [25] Adrian Pekar et al. “P4toNFV: Offloading from P4 switches to NFV in programmable data planes”. In: *International Journal of Communication Systems* 37.5 (2024), e5687. DOI: [10.1002/dac.5687](https://doi.org/10.1002/dac.5687). eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/dac.5687>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/dac.5687>.
- [26] Buchammagari Avinash Reddy, Kshira Sagar Sahoo, and Monowar Bhuyan. “Securing P4-SDN Data Plane against Flow Table Modification Attack”. In: *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. 2024, pp. 1–5. DOI: [10.1109/NOMS59830.2024.10575461](https://doi.org/10.1109/NOMS59830.2024.10575461).
- [27] Ruben Ricart-Sanchez et al. “P4-NetFPGA-based network slicing solution for 5G MEC architectures”. In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019. DOI: [10.1109/ANCS.2019.8901889](https://doi.org/10.1109/ANCS.2019.8901889).
- [28] Ruben Ricart-Sanchez et al. “Toward hardware-accelerated QoS-aware 5G network slicing based on data plane programmability”. In: *Transactions on Emerging Telecommunications Technologies*. Vol. 31. 1. 2020. DOI: [10.1002/ett.3726](https://doi.org/10.1002/ett.3726).
- [29] Florentin Rochet and Olivier Pereira. “Dropping on the Edge: Flexibility and Traffic Confirmation in Onion Routing Protocols”. In: *Proceedings of the Privacy Enhancing Technologies Symposium 2* (2018), pp. 27–46. DOI: [10.1515/popets-2018-0011](https://doi.org/10.1515/popets-2018-0011).
- [30] Ondrusova Sandra. *Generic Network Slice Template*. Apr. 2023. URL: <https://www.gsma.com/newsroom/wp-content/uploads/NG.116-v9.0-2.pdf>.
- [31] Roshan Sedar et al. “Supporting Emerging Applications With Low-Latency Failover in P4”. In: *NEAT '18: Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*. 2018, pp. 52–57. DOI: [10.1145/3229574.3229580](https://doi.org/10.1145/3229574.3229580).
- [32] Alexandros Stoltidis, Kostas Choumas, and Thanasis Korakis. “Active queue management in 5G and beyond cellular networks using Machine Learning”. In: *Computer Communications* 236 (2025), p. 108108. ISSN: 0140-3664. DOI: <https://doi.org/10.1016/j.comcom.2025.108108>. URL: <https://www.sciencedirect.com/science/article/pii/S0140366425000659>.
- [33] Alexandros Stoltidis, Kostas Choumas, and Thanasis Korakis. “Active Queue Management in Disaggregated 5G and Beyond Cellular Networks Using Machine Learning”. In: *2024 19th Wireless On-Demand Network Systems and Services Conference (WONS)*. 2024, pp. 113–120. DOI: [10.23919/WONS60642.2024.10449568](https://doi.org/10.23919/WONS60642.2024.10449568).
- [34] Mohammad Parsa Toopchinezhad and Mahmood Ahmadi. “Machine learning approaches for active queue management: A survey, taxonomy, and future directions”. In: *Computer Networks* 262 (2025), p. 111174. ISSN: 1389-1286. DOI: <https://doi.org/10.1016/j.comnet.2025.111174>. URL: <https://www.sciencedirect.com/science/article/pii/S1389128625001422>.
- [35] Belma Turkovic et al. “Fast network congestion detection and avoidance using P4”. In: *NEAT '18: Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*. 2018, pp. 45–51. DOI: [10.1145/3229574.3229581](https://doi.org/10.1145/3229574.3229581).

- [36] Belma Turkovic et al. “P4QoS: QoS-based Packet Processing with P4”. In: *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. 2021. DOI: [10.1109/NetSoft51509.2021.9492539](https://doi.org/10.1109/NetSoft51509.2021.9492539).
- [37] Shuo Wang et al. “Demo: An SDN-driven Active Queue Management Architecture for Assisting Mission-Critical Traffic Flows in Industrial Automation Systems”. In: *Proceedings of the ACM SIGCOMM 2024 Conference: Posters and Demos*. ACM SIGCOMM Posters and Demos '24. Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 119–121. ISBN: 9798400707179. DOI: [10.1145/3672202.3673741](https://doi.org/10.1145/3672202.3673741). URL: <https://doi.org/10.1145/3672202.3673741>.
- [38] Zhixi Wang. *5G Standards and Progress*. Sept. 2018. URL: <https://www.itu.int/en/ITU-D/Regional-Presence/AsiaPacific/SiteAssets/Pages/ConformityandInteroperability/Session%202-2.pdf>.
- [39] Yan-Jing Wu et al. “Network Slicing for mMTC and URLLC Using Software-Defined Networking with P4 Switches”. In: *Electronics*. Vol. 11. 14. 2022. DOI: [10.3390/electronics11142111](https://doi.org/10.3390/electronics11142111).
- [40] Boyuan Xiang et al. “P4Com: Implementing P4-Based In-Network Data Reduction for Geo-Distributed Machine Learning”. In: *2025 10th International Conference on Computer and Communication System (ICCCS)*. 2025, pp. 8–13. DOI: [10.1109/ICCCS65393.2025.11069944](https://doi.org/10.1109/ICCCS65393.2025.11069944).
- [41] Zichen Xu, Ziyi Lu, and Zuqing Zhu. “Information-Sensitive In-Band Network Telemetry in P4-Based Programmable Data Plane”. In: *IEEE/ACM Transactions on Networking* 32.6 (2024), pp. 5081–5096. DOI: [10.1109/TNET.2024.3448244](https://doi.org/10.1109/TNET.2024.3448244).
- [42] Changzhen Zhang, Jun Yang, and Ning Wang. “An active queue management for wireless sensor networks with priority scheduling strategy”. In: *Journal of Parallel and Distributed Computing* 187 (2024), p. 104848. ISSN: 0743-7315. DOI: <https://doi.org/10.1016/j.jpdc.2024.104848>. URL: <https://www.sciencedirect.com/science/article/pii/S0743731524000121>.
- [43] Haijun Zhang et al. “Network Slicing Based 5G and Future Mobile Networks: Mobility, Resource Management, and Challenges”. In: *IEEE Communications Magazine* 55.8 (2017), pp. 138–145. DOI: [10.1109/MCOM.2017.1600940](https://doi.org/10.1109/MCOM.2017.1600940).
- [44] Henghua Zhang et al. “TT-INT: A Time-Threshold-based Lightweight In-Band Network Telemetry Scheme for P4-Enabled Programmable Networks”. In: *TechRxiv* (2025). DOI: [10.36227/techrxiv.175242570.03829130/v1](https://doi.org/10.36227/techrxiv.175242570.03829130/v1).
- [45] Kaiyi Zhang, Nancy Samaan, and Ahmed Karmouch. “A Machine Learning-Based Toolbox for P4 Programmable Data-Planes”. In: *IEEE Transactions on Network and Service Management* 21.4 (2024), pp. 4450–4465. DOI: [10.1109/TNSM.2024.3402074](https://doi.org/10.1109/TNSM.2024.3402074).
- [46] Shunliang Zhang. “An Overview of Network Slicing for 5G”. In: *IEEE Wireless Communications* 26.3 (2019), pp. 111–117. DOI: [10.1109/MWC.2019.1800234](https://doi.org/10.1109/MWC.2019.1800234).
- [47] Huancheng Zhou and Guofei Gu. “Cerberus: Enabling Efficient and Effective In-Network Monitoring on Programmable Switches”. In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 4424–4439. DOI: [10.1109/SP54263.2024.00016](https://doi.org/10.1109/SP54263.2024.00016).