Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Fachbereich Informatik
Arbeitsbereich Kommunikationsnetze

Master Thesis Computer Science

Implementation and Performance Evaluation of Snabb-Based Service Functions to Provide Externs in P4-Based Network Softwarization

Etienne Zink

31.08.2024

Reviewers

Prof. Dr. habil. M. Menth
Fachbereich Informatik
Arbeitsbereich Kommunikationsnetze
Eberhard Karls Universität Tübingen

Jun.-Prof. Dr. J. Brachthäuser
Fachbereich Informatik
Arbeitsbereich Software Engineering
Eberhard Karls Universität Tübingen

Supervisor

Fabian Ihle M.Sc.

Etienne Zink:
Implementation and Performance Evaluation of Snabb-Based Service Functions to Provide Externs in P4-Based Network Softwarization
Master Thesis Computer Science
Eberhard Karls Universität Tübingen
Time period: 01.05.2024 - 31.08.2024

Erklärung

Hiermit versichere ich, dass ich die vorliegende Master Thesis selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Master Thesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tübingen, den 31.08.2024

(Etienne Zink)

Contents

The programmability of switches was enabled due to Network Softwarization. Programming Protocol-Independent Packet Processors (P4) is a programming language for programming switches. Programmable switches are either implemented in software or hardware. Hardware switches have limited capabilities and are not as flexible as software ones. To extend their capabilities, they support features directly implemented in hardware. P4 describes those hardware features as *externs*. Not every algorithm is implementable on a hardware device, even with these externs. Therefore, a proposal is needed to enable the implementation of more sophisticated algorithms.

This thesis proposes an architecture to extend the capabilities of P4-programmable switches. The more sophisticated externs are implemented on a co-located bare-metal server. Snabb is used to implement the externs. Snabb is a framework for fast packet processing. The signaling between the switch and server is proposed as well. The signaling enables the switch to specify which extern needs to be applied by the co-located server. VLAN is used as the signaling header. Further, a custom Snabb header can be used to provide additional data. The signaling in the architecture is extensible for new externs, and the concept allows load distribution among multiple instances implementing the same extern.

Two externs are further proposed in this thesis. One is the *Delayer extern* and the other the *Packet Ordering Function (POF) extern*. The Delayer can delay packets according to a configured delay and jitter. The POF can order packets and supports possible packet loss. Both externs are evaluated according to their throughput, delay, and jitter. In comparison to the Network Emulator (NetEm), the Delayer supports a 10 times higher transmission rate of UDP traffic while being more accurate. The POF has similar throughput capabilities as the Delayer and introduces a predictable delay and jitter.

1. Introduction

This chapter first describes the problem to be solved by this thesis. Afterward, the objective of the later-presented proposal is presented. At the end, the structure of this thesis is described.

1.1. Problem Statement

Network Softwarization enabled fast prototyping and testing of new network functionalities. Researchers need programmable network devices to develop and test new algorithms and protocols. The Network Softwarization enabled this by breaking up the fixed network devices into more modular ones. These modular devices can be implemented in software or hardware. Software devices are the most flexible [36]. Due to this flexibility, they do not have as high a packet processing rate as specialized hardware ones. Specialized hardware devices are implemented with Network Processing Units (NPUs), Field Programmable Gate Arrays (FPGAs), or Application-Specific Integrated Circuits (ASICs). Hardware devices have a higher packet processing rate but are not as flexible as software ones [36]. Therefore, implementing new algorithms and protocols on specialized hardware devices is more challenging. Nevertheless, specialized hardware devices are more relevant because of the higher packet processing rates. These higher rates are relevant to proving the implemented concepts to be feasible on a commercial scale [12]. Researchers are especially interested in programmable, specialized hardware devices due to their higher packet processing rates.

The Intel® Tofino™ Switch-ASIC is a programmable ASIC used in hardware devices [1]. It is programmable through the programming language "P4". The language P4 is publicly specified and can be extended through *extern* objects or functions [39]. Externs are defined by the hardware that executes the P4 program. The Intel® Tofino™ itself specifies a set of hardware features as externs [2]. The Tofino™ compromises the constraint that packets are always processed at line rate [12]. Therefore, even with externs, not every algorithm or protocol is implementable if it isn't executable in line rate. Examples of currently unimplementable features are delays, reordering, or encryption. The first two are unimplementable due to the low buffer memory of 22MB [12]. The last is unimplementable due to the lack of an encryption extern. Therefore, a concept is needed to enable more sophisticated features for the Intel® Tofino™ Switch-ASIC.

1.2. Objective of this Proposal

In this proposal, the shortcomings of P4-based switches not being able to implement more sophisticated features should be addressed. A concept has to be proposed for how to implement and integrate sophisticated externs for P4-based switches. Therefore, a co-located server should be used. This server should implement sophisticated features with a Virtual Network Function (VNF) framework, like *Snabb*. Snabb is a framework for implementing fast packet processing. The VNFs implemented in Snabb should run on a bare-metal server for better performance. To seamlessly integrate these externs, a signaling between the P4-based switch and the co-located server has to be proposed as well. Two externs should be implemented to verify the proposed architecture and signaling. Furthermore, the capabilities of this proposal should be evaluated. Therefore, the imposed signaling and processing overhead has to be quantified by a baseline. Afterward, the maximal throughput and feature-specific metrics have to be measured and evaluated for the implemented externs. In summary, the more sophisticated features should be implemented on a co-located server through Snabb, and the capabilities of these implementations should be evaluated.

1.3. Structure of this Thesis

In chapter 2, the relevant background for this thesis is introduced. Especially Software-Defined Networking (SDN), P4, VNFs, and Snabb are introduced. Afterwards, chapter 3 presents the related work. The focus is on published papers and technical reports that propose the extension of programmable switches through external elements or evaluate VNFs. In chapter 4, the concept and implementation of the proposal are presented. First, the proposed architecture and signaling are described. Then, the load balancing in the architecture and, finally, two implemented externs are presented. Afterward, chapter 5 presents various experiments. The first experiments are evaluated to define a baseline for Snabb. Then, experiments evaluating the two externs are presented and evaluated. Finally, chapter 6 concludes the thesis. The conclusion summarizes the proposal, discusses it, and presents possible future work.

2. Technical Background

This chapter introduces all the advanced and still necessary concepts needed to understand this thesis. In the first section, the concept of SDN and the *control plane* and *data plane* of networking devices are introduced. Afterward, in the second section, the programming language P4 is described. This programming language is used to implement custom packet forwarding. In the third section, the Intel® Tofino™ is introduced. It is a P4-programmable switching-ASIC used for high throughput packet processing. This ASIC is used for the evaluation of this thesis. Afterward, in the fourth section, the concept of VNFs is described. VNFs are software implementations of traditionally hardware-based network services. A special focus is the implementation considerations of VNFs. The fifth and last section introduces Snabb. Snabb is a framework for fast packet processing running on commodity hardware.

2.1. Software-Defined Networking

This section introduces the concept of SDN and differentiates it from traditional networking. First, the evolution of the network programmability is presented. The term SDN and the corresponding terms *control plane* and *data plane* are then introduced. Afterward, the control plane and data plane are further introduced, and examples are given to further elaborate on them.

2.1.1. Evolution of Network Programmability

This subsection first presents how traditional networking devices were built. Afterward, the relevant *control plane* and *data plane* are introduced. Those are relevant to present the evolution of the network programmability from *fixed function* black boxes to *data plane programmable* devices.

Traditionally, network devices like routers or switches were only configurable *black boxes*. Due to this, the evolution of already deployed devices was limited [26]. Even the configuration of these was quite challenging. Different devices had only limited and vendor-specific control interfaces. Therefore, deploying new network protocols was a great challenge back in those days. Especially since the configuration only allowed changing different parameters inside of the device [12]. Still, the already-deployed algorithms would not change except for these parameters. SDN was one of the first attempts to overcome these challenges and split these configurable *black boxes* into modular programmable ones.

The algorithms and protocols used in network devices can be split into three different classes. These are the *data plane*, the *control plane*, and the *management plane*. The management plane has the smallest effect on packet processing [12]. Moreover, it is not considered in the context of this thesis. Therefore, a detailed

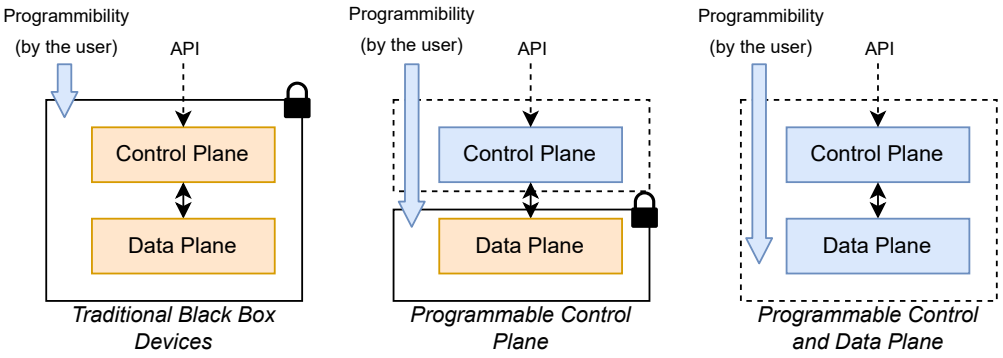

Figure 2.1.: Network programmability models (based on [12]).

description of the management plane is omitted. In contrast, the data plane and control plane have a great effect on packet processing. The control plane decides how the traffic has to be handled [8]. According to these decisions, the data plane forwards this traffic. This proposal leverages these two to solve the given problem. For further elaboration on these two algorithm classes (control plane and data plane), descriptions with examples are provided in subsections 2.1.2 and 2.1.3.

The SDN philosophy is to separate the data plane and the control plane [14]. As a first step towards this philosophy, the *control plane programmability* was introduced [12]. Due to the programmability, the users can bypass built-in control plane algorithms and introduce their own. This resulted in the development of OpenFlow as a protocol to leverage the control plane programmability [24]. The difference between the programmable control plane and a traditional *black box* device is shown in Figure 2.1. User-programmable algorithms are shown in blue. The provider's fixed or configurable algorithms are shown in orange. As the next step towards the SDN philosophy, *data plane programmability* was introduced [12]. Like in control plane programmability, the users are now able to introduce their own algorithms. Especially new algorithms running on switching ASICs for high-speed packet processing. Figure 2.1 visualizes as well the model of a programmable *control plane* and *data plane*. Nowadays, the data plane and control plane can be separated, and both might be programmable.

2.1.2. Control Plane

In this subsection, controllers that implement the behavior of a network are introduced. Further, different controller properties are presented, and the relevant ones are highlighted. Afterward, *control plane algorithms* are further elaborated due to examples. Two examples are given: a *short-circuit ARP* and *custom link aggregation*.

Devices implementing control plane algorithms are called *controllers*. A controller can either be centralized or decentralized [26]. One physically centralized controller imposes a single point of failure. Multiple physically decentralized controllers can solve this problem. On the one hand, one controller can be used as a backup for

another. On the other hand, approaches were proposed, deploying a single logical but physically distributed control plane. Even hierarchical approaches with local and global controllers were proposed [11]. All distributed approaches are subject to the trade-off described in the CAP theorem [9]. The CAP theorem states that out of three possible properties, at most two of them can be fulfilled for a distributed system. These three properties are: consistency, availability, and partition tolerance. Furthermore, controllers can be local or remote concerning the data plane [12]. Local controllers are co-located to the data plane. Remote controllers are located rather centralized and not as near to the data plane as local ones. For simplicity, only a remote, physically centralized controller is used in this thesis.

Control plane algorithms decide how traffic is handled [8]. They control the network behavior like signaling, network paths, or forwarding [22]. Examples of how control plane algorithms control the network behavior are a *shot-circuit ARP* or a *custom link aggregation*. In the *short-circuit ARP* the controller can be programmed to answer ARP requests itself. This can be useful if the data plane is not able to forward packets because an entry on how to forward the packet is missing. In such a case, the data plane forwards the packet to the controller. The controller either does not know the answer to the ARP request and floods the packet or knows the answer and sends an ARP response immediately. With this implementation, the flooding overhead on the network is reduced. Such an algorithm is only implementable due to the *control plane programmability*. The same is true for the *custom link aggregation*. One can implement an algorithm that classifies flows and forwards them to one of the aggregated links. Due to the programmability, the classification can happen on arbitrary fields supported by the data plane. The controller can do the classification and configure the data plane accordingly for future forwarding. It is important to only use header fields supported by the data plane. Else the controller cannot configure the data plane accordingly and has to do the forwarding all by itself. This would contradict the philosophy of SDN to separate the data plane and control plane since the control plane would act as the data plane. These two algorithms were examples of how the control plane controls the behavior of the network.

2.1.3. Data Plane

The data plane is generally organized in flow tables. Therefore, this subsection first introduces them. Then, the relevance of *data plane programmability* in addition to *control plane programmability* is highlighted. This subsection ends with an example of what data plane algorithms are. This example is how a *1+1 protection* algorithm can be implemented.

In addition to the control plane, the data plane forwards the packets according to the decisions made [8]. Generally, the data plane is organized as flow tables [22]. This is a common approach for traditional switches, routers, firewalls, or middleboxes. One scheme of such flow tables can be found in the OpenFlow specification [28]. The scheme is organized as a table. It consists of (flow) entries with specific fields. Examples of those can be *match*, *priority*, or *instruction fields*. Match fields are used to match against packets based on header fields or metadata. Priority fields are used to define precedence if multiple flow entries match. Instruction fields indicate how

to alter the packet processing. Those three are highly relevant since they define which algorithm is executed for a flow of packets. Programmable data planes like with P4 allow users to program these flow tables [5]. This is enabled by defining custom parsers for new header fields or custom instructions. Nevertheless, control plane programmability can already mimic this kind of programmability. However, implementing data plane algorithms in the control plane would not only contradict the SDN philosophy but also typically impose a significant performance degradation [12]. Therefore, the ability of data plane programmability to program the flow tables and introduce new packet parsing and processing was an advancement compared to control plane programmability.

For further clarification on what algorithms the data plane can implement, an example of how *1+1 protection* can be implemented is provided in the following [21]. The 1+1 protection mechanism duplicates configured traffic and sends each duplicate over a disjoint path. A common node on both paths then needs to only forward one duplicate of each packet and eliminate the other. Data plane programmability allows users to implement this algorithm themselves. They can define a custom *1+1 protection header* depending on their needs, located after the Ethernet or IP header. This new header includes arbitrary fields relevant to implementing the algorithm. Especially, users can implement the processing logic of tracking what packets were already forwarded and should be eliminated in the future. The data plane will expose a corresponding Application Programming Interface (API) to the control plane to configure this behavior. To summarize, the data plane can be programmed to support new protocols, consisting of new headers and algorithms that exceed the capabilities of current devices.

2.2. P4

This section introduces the programming language P4. First, the language properties and the development and deployment process are described. Afterward, P4 hardware abstractions called architectures, and especially the reference architecture Protocol-Independent Switching Architecture (PISA), are introduced. Thereafter, components of the language and architectures are further elaborated. Especially, data types, the parser and deparser, control blocks, and externs are described.

2.2.1. Language Properties

This subsection focuses on the language history and properties of P4. These properties were presented in the paper from Bosshart et al. [5] which proposed P4. Afterward, these properties are further elaborated in the development and deployment process of P4. It is further introduced, what users need to supply and what manufacturers do.

The P4 programming language was first introduced by Bosshart et al. [5] for programming the data plane. Afterward, it was standardized by the P4 Language Design Working Group [12]. There are currently two different standards specifying P4: $P4_{14}$ and $P4_{16}$. $P4_{14}$ was first specified in 2015 [38]. Due to limitations imposed by the language design, $P4_{16}$ was first specified 2017 to address these limitations [12].

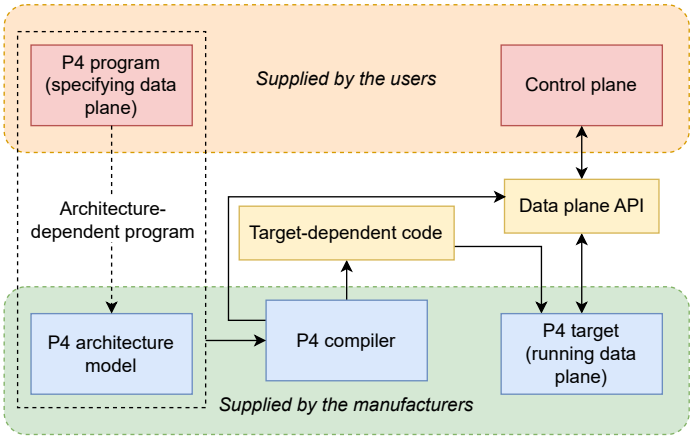

Figure 2.2.: P4 development and deployment process (based on [12, 39]).

Therefore, the development of $P4_{14}$ was suspended. In the following, P4 and $P4_{16}$ are used interchangeably, since only $P4_{16}$ is used in this thesis.

In their original paper, Bosshart et al. [5] stated three main properties of P4:

Reconfigurability is the ability of the controller "to redefine the packet parsing and processing in the field" [5].

Protocol independence means the data plane is independent of specific packet formats. The controller should be able to define a packet parser for arbitrary headers containing named fields with specific types. Afterward, Match-Action Tables (MATs) being able to process those headers should be defined by the controller.

Target independence describes the portability of a program written with P4 to nearly arbitrary switching ASICs. A target describes a specific kind of switching (software or hardware) device in this context [12]. The program written with P4 should be independent of the details of the underlying switching ASICs. This target-independent representation should then be compiled into a target-dependent program while taking the target's capabilities into account.

Those three properties can be found in the development and deployment process of P4 programs. Figure 2.2 visualizes this process. There are two major classes of components in the process: user-supplied and manufacturer-supplied components. The manufacturers need to supply a P4-capable target. Further, they need to include a P4 compiler, which compiles the target-independent P4 code into specific code for their target. To properly use the target, the users supply a P4 program implementing the desired data plane. Although the provided program is target-independent, it is still architecture-dependent. Architectures are introduced in subsection 2.2.2. Meanwhile, the architecture can be imagined as an abstract model of a class of targets. The manufacturer-supplied compiler then compiles the architecture-dependent code into a target-dependent one. This code can then be run on the target to realize

the desired data plane. Furthermore, the compiler generates a data plane API. This API can be used by a user-supplied control plane to configure the provided data plane on the target. In this process, the target independence is provided by the P4 programming language itself and the compiler. Reconfigurability is achieved through the data plane API and the possibility of independently developing P4 programs and then redeploying them onto the targets. The protocol independence cannot be shown by Figure 2.2 since the P4 program is only shown at a high level. In the following subsections, key components of the P4 programming language, like data types, parsers, control blocks, or externs, are introduced. These key components will show how the three main properties are further implemented in P4.

2.2.2. Architectures

To support the target independence, P4 hardware abstractions were introduced. This subsection defines these hardware abstractions called architectures. Further, the reference architecture PISA is introduced. Especially, the different components of this architecture and the data passed between them are described.

The P4 architectures are an intermediate layer in between the manufacturer-supplied target and the core P4 language [12]. They can be viewed as an abstract model representing the target's capabilities and a logical view of the processing pipeline. This processing pipeline consists of predefined P4-programmable blocks, like parser or control flows [39]. Further, the architecture defines for each programmable block a data plane interface, specifying the data passed in or out of this block. A target supports one or multiple architectures. Each architecture fully or partially describes the capabilities of the data plane. Manufacturers need to supply a compiler for each architecture supported by their target. This compiler will map the architecture-specific P4 program into a target-specific one [12]. Therefore, an architecture-specific P4 program can be compiled to and run on every target supporting this specific architecture. Architectures are one of the main enablers of one of the P4 main properties, *target independence*.

To further elaborate on architectures, an example architecture called PISA is introduced in the following. Figure 2.3 depicts the scheme of the PISA reference architecture. It consists of three major components [12]:

A programmable parser is a finite-state machine. It defines the order of packet headers. These headers are then extracted and deserialized into a structured, well-defined format. The parser is further described in subsection 2.2.4.

The programmable match-action pipeline consists of at least one or multiple Match-Action Units (MAUs). Each MAU consists of one or multiple MATs. These MATs are a kind of flow table. Like flow tables, MATs are separated into a match and action (instruction) logic. The match logic defines a set of data to match packets on. Afterward, a match-specific action with supplied action data is executed to perform arithmetic operations or header modifications. The pipeline can further be subdivided into an ingress and egress part. Both can be connected through a packet replication engine with a traffic manager.

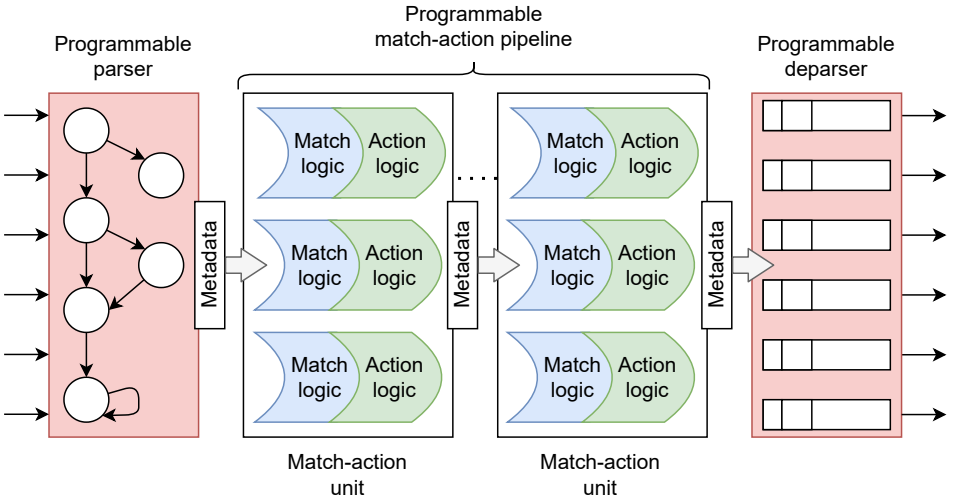

Figure 2.3.: PISA reference architecture (based on [12]).

A programmable deparser is the counterpart of the parser, which serializes packets. The deparser is further described in subsection 2.2.4.

Figure 2.3 further depicts metadata being passed between the different blocks. This metadata does not include the packet payload [12]. The payload bypasses the programmable match-action pipeline and reaches the programmable deparser unchanged. Packet metadata can be divided into three different classes of metadata [12]:

Packet headers represent the protocol headers. They are extracted in the parser and emitted in the deparser.

Intrinsic metadata is specific to the architecture. It serves as the input or output of fixed-function components. Therefore, it can be used to derive information (e.g., the input port) or control their behavior (e.g., set the output port).

User-defined metadata is a temporary storage of custom information later needed, similar to custom variables in other programming languages.

2.2.3. Data Types

In this subsection, the data types present in P4 are introduced. There are two major classes of data types: base types and derived types. For each type, relevant examples are presented. At the end, an example code for a derived type implementing a protocol header is presented.

P4 is a statically typed language consisting of base types and derived types [39]. The most relevant base types are the `match_kind`, boolean values (`bool`), constant integer values (`int`), and fixed width (`w`) unsigned bit-strings (`bit<w>`). The `match_kind` is used to represent different implementations of table lookups. In the

```
typedef bit<48> mac_addr_t;
typedef bit<16> ether_type_t;

header ethernet_h {
    mac_addr_t dst_addr;
    mac_addr_t src_addr;
    ether_type_t ether_type;
}

struct header_t {
    ethernet_h ethernet;
}
```

Listing 2.1: Sample P4 code to define a `struct` containing an Ethernet `header`.

core language, there are three different `match_kind`s defined: `exact`, `ternary`, and `lpm`. With the `exact` match, the provided value has to exactly match the expected value. A `ternary` match provides an expected value and a bitmask of the width of the expected value. The bitmask defines which bits should be considered for the match. Each bit set to one has to be considered for the match, and each bit set to zero does not care for the match. So, the provided value matches the expected value if the provided value is equal to the expected value at every bit-position of the bitmask being set to one. Therefore, a `ternary` match with a bitmask, where every bit is set to one, is the same as an `exact` match. Last, the `lpm` (longest-prefix match) is a special kind of `ternary` match. Here, the bitmask consists of `n` consecutive bits set to one from left to right. In addition to these three `match_type`s, architectures can support further ones.

These base types can further be combined into derived types. Two of the most common ones are the `header` and the `struct` type [12]. The `header` data type is similar to a C struct, except the additional *validity* field [39]. If the validity is true, the header is considered to be emitted; otherwise, it is not. It further consists of named fields with fixed-width types, like bit-strings, integers, or booleans. They correspond to packet protocol headers, like Ethernet or IP. Headers are extracted by the parser and emitted by the deparser. The second derived data type is a `struct`. A struct in P4 is similar to a struct in C. In contrast to a header, a struct can be composed of fields of arbitrary types [12]. Therefore, they can be composed of fields of other structs or headers. Furthermore, structs are used to define user-defined metadata. Due to these data types, P4 can express all possible protocol headers a packet contains.

Listing 2.1 provides an example of declaring an Ethernet header and header struct. At first, two custom types `mac_addr_t` and `ether_type_t` are declared. These two are used to express fields in the Ethernet protocol header. Afterward, an Ethernet header is defined as a `header` type. This header consists of the destination and source MAC addresses and an ethertype. Lastly, a *header struct* is declared. This

header struct depicts all possible headers that can be parsed from a packet. In this case, only an Ethernet header can be parsed and further processed. This code presents the basic structure to further define additional headers and structs.

2.2.4. Parser and Deparser

In the following subsection, the components parser and deparser are introduced. Therefore, the Finite State Machine (FSM) defining a parser is described. Afterward, an example of a FSM implementing a simple Ethernet frame parser is presented. Lastly, the deparser and its differences from the parser are described.

In P4, a FSM defines a parser [39]. The FSM defines one start and two final states. The start state is called `start` and the two final states are called `accept` and `reject`. While the `accept` state indicates a successful parsing, the `reject` state indicates an unsuccessful parsing. Unsuccessful parsing happens, e.g., if unintended packet headers are present. Additionally, users can define custom (intermediate) states. In these custom states, three main language features are used to define the FSM [39]:

Data extraction can be used to extract a header from the incoming packet. Therefore, the method `extract` is called on the incoming packet. An argument, representing the packet header to extract the header, needs to be provided.

Transitions are used to transfer control to another state. This other state can be a predefined or custom one. The `transition` statement is used to perform this state transition.

Selections are kind of similar to switch statements found in other programming languages. A major difference is the fact that selections in P4 are expressions and not statements. Therefore, the `select` expression can be used in combination with `transition` statements. This enables the user to transition into a state according to a header field value.

Figure 2.4 depicts a sample parser for an Ethernet frame with an IP header. In this figure, the custom states are shown in blue, and the predefined ones in gray (Start), red (Reject), and green (Accept). The parser starts in the *Start* state. It then transitions into the *Ethernet* state where the ethernet header is extracted. After the data extraction, a conditional transition is performed based on the ethertype field of the Ethernet header. The `select` expression in combination with the `transition` statement is used therefore. If the ethertype indicates an IP header, the control is passed to the *IP* state; otherwise, to the *Reject* state. The *IP* state will again extract its corresponding header and then transition into the *Accept* state, which indicates a successful parsing. In contrast, the *Reject* state indicates an unsuccessful parsing. This leads to intrinsic metadata indicating this unsuccessful parsing or the packet will be dropped immediately. This sample parser shows the *protocol independence* of P4 because arbitrary protocols can be defined and parsed.

In P4 the programmable deparser is used for serializing outgoing packets [39]. A parser deserializes an incoming packet into a structured, well-defined format. After passing all MAUs, the struct containing all headers reaches the deparser. To serialize

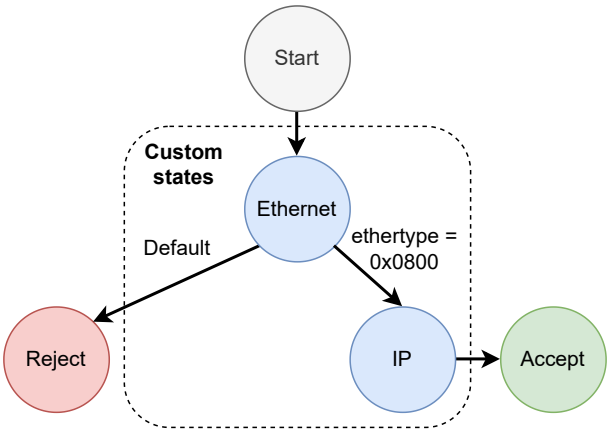

Figure 2.4.: Sample FSM of a P4 parser for Ethernet frames with an IP header (based on [39, 12]).

data onto the outgoing packet, the deparser needs to call the `emit` method of the outgoing packet. An argument containing the data to serialize needs to be provided to the method. It can be e.g., of the types `header` or `struct`. It should be noted that only valid headers are serialized. Furthermore, the deparser serializes the data in the order of the calls of the `emit` method. Therefore, the first data being serialized is the first data to be present in the packet. In contrast to the FSM defining the parser, the deparser consists of *just* a block of code defining the order and which headers are serialized.

2.2.5. Control Blocks

This subsection focuses on the control blocks in P4. First, the further used concepts and names will be introduced. Afterward, the implementation of MATs is described. Lastly, it is presented how packets are matched by a MAT.

In P4, control blocks are responsible for manipulating and transforming the metadata, especially the headers [39]. Control blocks are declared with the `control` keyword followed by a name and parameters. These parameters are used to pass in and out headers, intrinsic metadata, or user-defined metadata. The control body is an imperative program in which MAUs can be invoked. MAUs are represented by `table`s. Therefore, they are sometimes referred to as MATs. This naming conventions contradict the one from PISA shown in Figure 2.3. There are the control blocks from P4 called Match-Action Units and the Match-Action Units from P4 are called Match-Action Tables. In the context of P4, it is common to call them *control blocks*, or simply *controls*, and *Match-Action Tables* [12]. Therefore, these concepts are called *controls* and *Match-Action Tables* in this thesis. Controls can be user-defined, or depending on the architecture, predefined ones can exist, like an ingress control or an egress control.

The control body can declare `action`s and `table`s [39]. Actions are kind of functions, which can read and write data. Further, they can contain values (action data)

set by the control plane. Therefore, they are the main concept for the control plane to influence the behavior of the data plane. Special actions exist like the `apply` action. This action is used to invoke a control or `table`. The `apply` action of a control can be programmed by the users to define the processing algorithm [12]. Match-Action Tables in P4 are declared by one or multiple match keys, a set of possible (user-defined) actions, and additional attributes. The match keys consist of pairs of the form `(expression: match_kind)` [39]. A `match_kind` describes the algorithm to perform the lookup/match. They are of the type `match_kind` which was introduced in subsection 2.2.3 "Data Types". The `expression` describes header fields or metadata to be matched on. Entries for the MATs are populated by the control plane. The data plane is reconfigurable due to the capability of users programming `action`s and `table`s and the ability to populate the entries through the control plane.

A packet is processed by a MAT as follows [12]:

- The lookup key is constructed based on the defined header fields or metadata.
- A match against all entries is performed based on the lookup key and the lookup algorithm defined by the `match_kind`.
- If the most specific match is found, the corresponding action with its action data is called.
- If no match is found, a default action is invoked.

2.2.6. Externs

In P4, externs are used to extend the functionality of the core language by architecture-specific objects or functions [39]. Therefore, the keyword `extern` is used. Externs are similar to abstract classes in other programming languages. They declare the interface between architecture-specific objects and the data plane. Most of the externs are instantiated through a constructor [12]. All other methods are then called on this instance. Other externs, like functions, do not need an instantiation. Externs like a counter or a register allow the users to preserve a state between packet processing. Counters preserve the number of bytes and packets already processed. They increment each value according to the currently processed packet. Registers are used to preserve arbitrary values, like the last processed TCP sequence number. The data plane or control plane may be able to access this state and act depending on it.

2.3. Intel® Tofino™

In the following, the switching ASIC Intel® Tofino™ is introduced. First, the technical details of this chip are presented. Afterward, the Tofino Native Architecture is described. The Tofino Native Architecture (TNA) is the P4 architecture implemented by the Intel® Tofino™ . Especially, its differences from other architectures are highlighted.

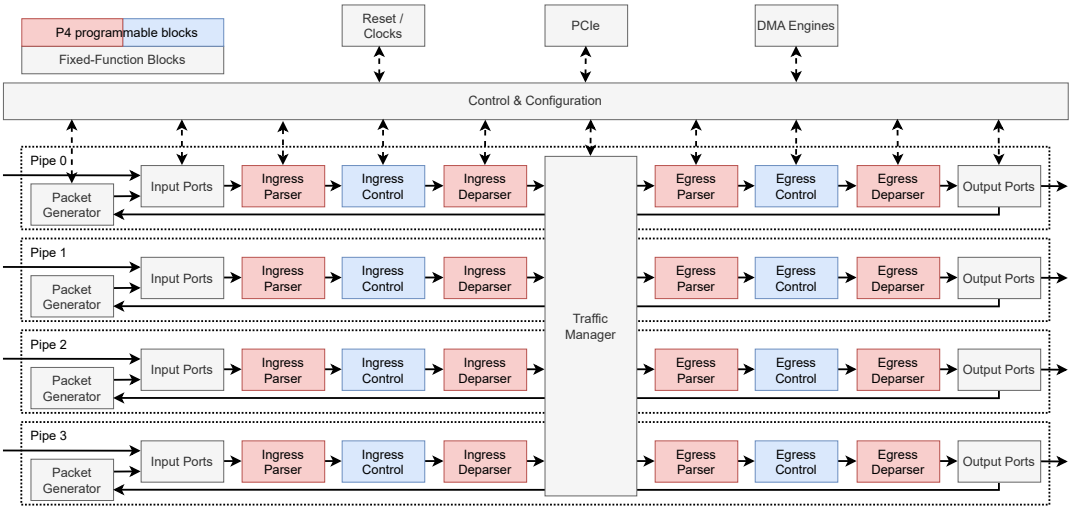

Figure 2.5.: Scheme of the Tofino Native Architecture with four pipes (based on [2]).

2.3.1. Technical Details

The Intel® Tofino™ [1] was the first user-programmable switch ASIC for Ethernet networks [12]. It was originally built by Barefoot Networks as an ASIC-based P4 target. Since 2019, Barefoot Networks has been a part of Intel®. Therefore, the Tofino™ now is part of the Intel® product line. The Tofino™ switch ASIC is designed for high throughputs. It supports 65 ports each running at 100 Gigabit per second (Gbps). Its successor, the Tofino 2, even supports them running at 400 Gbps. All packets are processed at line rate by the Intel® Tofino™ independent on the P4 program currently specifying the data plane. Therefore, not every valid P4 program can be run on the Tofino™ . Otherwise, this constraint cannot be fulfilled. For example, the Tofino™ has a buffer size of 22 MB. Therefore, it would not be able to delay packets at 100 Gbps for multiple microseconds. The same assumption holds for an arbitrary number of hardware operations. Each operation takes some time. Due to the limited time for processing each packet, only a finite number of operations can be supported. This constraint, to always operate at line rate, makes the Intel® Tofino™ a powerful and, at the same time, challenging switch ASIC.

2.3.2. Tofino Native Architecture

In this subsection, the architecture implemented by the Intel® Tofino™ is introduced. This architecture is the TNA. First, the TNA and its differences from other architectures are described. Here, the focus is based on the architectural scheme. Afterward, the differences are further described based on the TNA specification. This includes the different programmability of components and the provided externs.

The Intel® Tofino™ implements the TNA as its P4 architecture [2]. Figure 2.5 depicts the TNA with four pipes. The TNA extends the Portable Switch Architecture (PSA) [12]. In contrast to the PISA, the PSA defines an ingress and egress part consisting both of a parser, a match-action pipeline, and a deparser [40]. Therefore,

the PSA can be thought of as two PISAs, one for ingress and one for egress. This concept is further developed by the TNA. The TNA consists of two or four identical pipes [12]. Each pipe supports up to 16 ports, and every pipe can run its own P4 program independently. Every pipe can be thought of as a single PSA. In contrast to the PSA, each pipe of the TNA has an additional packet generator, input ports, and output ports. Furthermore, a *traffic manager* connects the ingress deparser and the egress parser. The main difference between the PISA and the TNA is that a TNA pipe consists of one ingress and one egress PISAs connected through a traffic manager.

Further differences between the TNA and the PISA are the programmability of components and the hardware features defined as externs. The parsers, deparsers, and controls are fully programmable components in the TNA. In contrast, the packet generator, traffic manager, input ports, and output ports of the TNA are not programmable and are only configurable. Not present in Figure 2.5 are the externs and the data plane interfaces of each of those components. These are defined in the public version of the TNA [2]. One extern to mention is the `Hash` extern. Through this extern, the Tofino™ can compute a hash over arbitrary data. The extern is instantiated with a predefined or custom hash algorithm. Some of the predefined ones are Cyclic Redundancy Checks (CRCs). Therefore, the TNA not only defines the components of the processing pipeline; it also defines hardware features and data plane interfaces.

The TNA defines further externs. These externs are not as relevant to this thesis as the `Hash` extern. Still, they define the hardware capabilities of the Tofino™. Therefore, an overview of most of the TNA externs is presented in the following:

Checksum verifies and re-computes header-only checksums.

Counter tracks the packet and/or byte count.

Digest sends messages from the data plane to the control plane.

Hash calculates a deterministic hash of arbitrary fields.

Meter measures and detects if a flow is arriving slower or faster than configured.

Mirror creates a copy of a packet and sends it to a destination.

Parser Counter counts an arbitrary value while parsing.

Random generates pseudo-random numbers according to a uniform distribution.

Register preserves a state that can be read or written while packet processing.

Resubmit repeats the ingress processing of a packet.

2.3.3. P4 Traffic Generator

The P4 Traffic Generator (P4TG) was first introduced by Lindner et al. [20] in 2023. It is a P4-based traffic generator running on an Intel® Tofino™ switching-ASIC. Therefore, it can generate up to 10× 100 Gbps traffic. This traffic is split

among ten ports, each supporting a maximum of 100 Gbps. While generation, the P4TG measures rates directly in the data plane, like the Layer 1 transmission rate. Further, the sent traffic can be received at any of these ten ports. If so configured, it can record packet loss, packet reordering, and round-trip times. It further measures the Layer 1 receive rate. The latest release can be found in the public GitHub repository [19].

2.4. Virtual Network Functions

This section introduces the concept of VNFs. First, the difference between traditional network functions and virtual ones is described. Afterward, the relationship between the two concepts VNF and SDN is described. Both concepts are often mixed up; therefore, both need to be differentiated. Lastly, implementation considerations regarding the *location* to run a VNF are introduced. Various locations introduce different trade-offs, which need to be considered.

2.4.1. Traditional vs. Virtual Network Functions

The first concept of virtualizing network functions was presented by Chiosi et al. [6] introducing Network Function Virtualization (NFV) [44]. Back then, they called them *virtual network appliances* but nowadays VNFs is more common. Chiosi et al. [6] proposed the NFV to tackle the challenge of deploying new network services. Traditionally, new network services, like firewalls, network address translations, or deep packet inspections, were deployed through hardware-based appliances. Therefore, network operators need to integrate and operate a variety of (proprietary) hardware appliances. Especially the integration is difficult due to the required space and power for each new service. NFV was proposed to overcome these obstacles. New network functions (VNF) should be implemented rather in software than in hardware. These VNFs can then be run on any commodity or virtualized infrastructure, like servers or Virtual Machines (VMs). Further, the functions can be moved to or instantiated in any location inside the network. Only commodity infrastructure needs to be present at these locations. Possible locations can be data centers, network nodes, or end-user premises. This leads to increased network scalability, resource utilization, and reduced capital and operational expenses [44].

2.4.2. Relationship to Software-Defined Networking

The two concepts, NFV and SDN, are often used together [7]. Nevertheless, Chiosi et al. [6] state that NFV and, therefore, VNFs are independent of SDN. Rather, NFV and SDN are two complementary concepts. SDN has the goal of creating a network abstraction. NFV, on the other hand, has the goal of reducing space and energy consumption and capital and operational expenses. Still, both are software-centric approaches to network management [7], but both contribute to network function disaggregation "[...] on different axes [...]" [7]. Figure 2.6 depicts those two different axes. One axis is the disaggregation of the control plane and data plane. SDN only influences this vertically drawn axis. The other axis represents the implementation

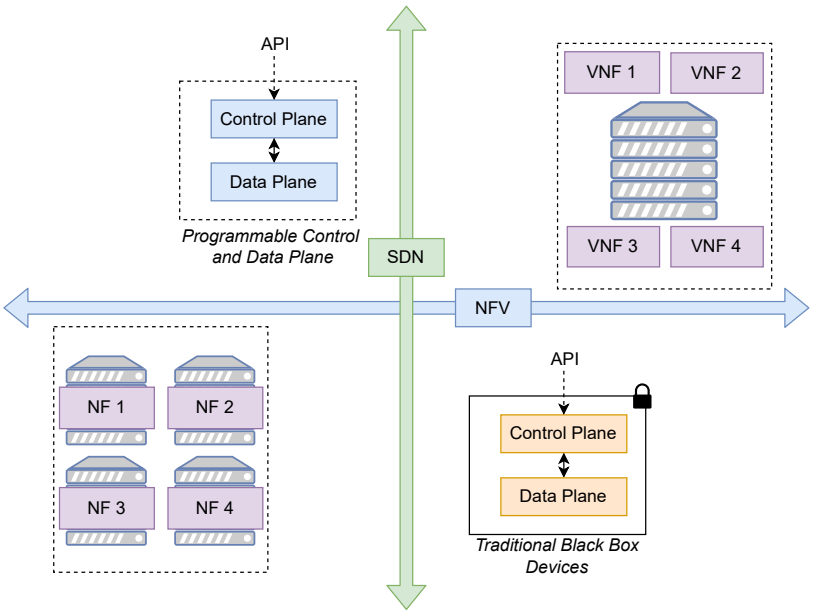

Figure 2.6.: Scheme of orthogonal concepts of SDN and VNF (based on the description of [7]).

of network functions in hardware (one function per box) or software (multiple VNFs running on one commodity infrastructure). NFV only influences this horizontally drawn axis. These two concepts can be arbitrarily combined. One combination is the traditional approach with no control plane and data plane disaggregation and only hardware-based services used. Another combination is control plane and data plane disaggregation, and VNFs (possibly extending the data plane) are used, running on commodity infrastructure. Therefore, SDN and NFV, especially VNFs, can be combined but do not require each other.

2.4.3. Implementation Considerations

In this subsection, the implementation considerations regarding the implementation of a VNF are introduced. It is relevant to consider the *location* in which the VNF is later run. Depending on this decision, different trade-offs are imposed on the VNF.

The *location* where a VNF is run in the Operating System (OS) has a significant impact on the packet processing rate [41]. Figure 2.7 depicts three different placements inside the OS. In the following, the three different placements are presented from left to right. First, the VNF can be run in the user space on top of the kernel networking stack. When a packet enters the system, it is received at the Network Interface Card (NIC). The packet is then processed by the NIC driver and the kernel networking stack. Afterward, it is passed from the kernel space to the user space and then processed by the VNF. To circumvent these overheads, VNFs can be directly placed in the kernel space as in-kernel VNFs. Their code is injected at runtime and the execution may happen at different hook points in the kernel

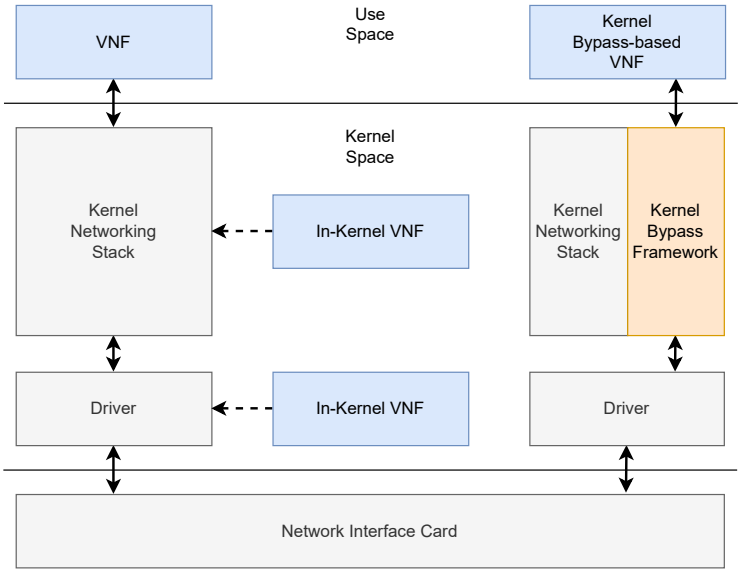

Figure 2.7.: Different VNF placements inside the Operating System (based on [41]).

networking stack [32]. Therefore, the packet does not need to be passed from the kernel space to the user space, and maybe not even the whole stack needs to be processed. A third approach would be still running the VNF in the user space but bypassing the kernel networking stack. This is also known as kernel bypass. Due to this approach, the packet is still passed from the kernel space to the user space, but it circumvents the overhead imposed by the networking stack. It is crucial to consider which placement to use for the use case that the VNF implements.

The three different VNF placements impose different limitations and processing overheads. The following limitations and overheads can be identified:

1. The kernel networking stack introduces an unnecessary overhead for VNFs not relying on it [32]. Furthermore, the networking stack is programmed for general usage and has multiple processing layers. Therefore, the imposed overhead is significant to the packet processing rate.

2. The passage of packets from the kernel space to the user space imposes a not negligible overhead due to the context switch [41].

3. In-kernel VNFs are under restrictions to ensure the integrity of the system [25]. Some of them are: a limited number of assembly instructions; no backward jumps are allowed; and no events like timeouts can be handled since only packet-driven processing is possible.

Evaluations of Parola et al. [32] show, that a kernel bypass VNF is only a little worse in the processing rate compared to an in-kernel VNF. Therefore, the trade-off between the programming restrictions and the processing overhead has to be considered before implementing a VNF for a use case.

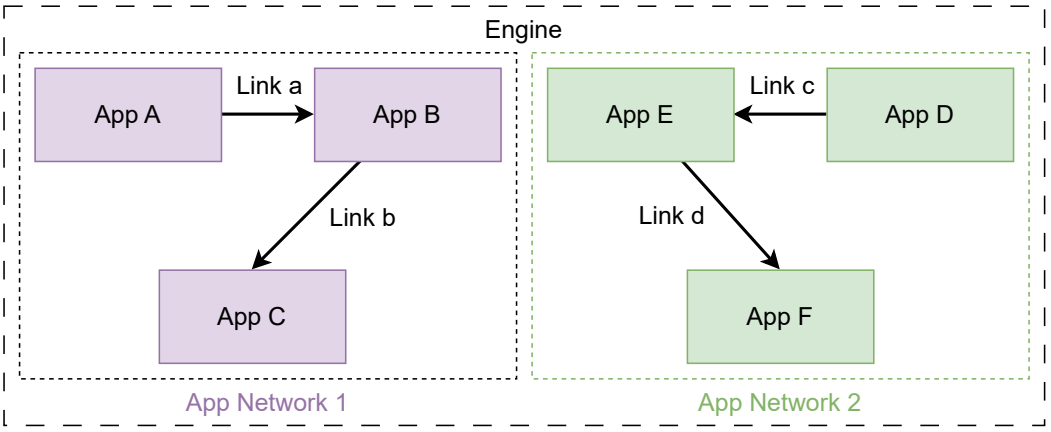

Figure 2.8.: Overview of Snabb components and their relationships.

2.5. Snabb

Snabb is a framework used to implement fast packet processing. In this section, Snabb and especially its relevant components are introduced. First, the general framework and its implementation are described. Afterward, in the second and third subsections, *apps* and the *app network* are further introduced. Both are the major components of implementing a custom packet processing behavior. Thereafter, special features and apps are introduced, like workers for multi-processing or a NIC driver. Lastly, the Foreign Function Interface (FFI) to incorporate C code into Snabb is described.

2.5.1. Packet Processing Framework

This subsection introduces Snabb. First, an overview of the project and its implementation are given. Afterward, the main components of Snabb are introduced. Those are the *engine*, *apps*, and the *app network*.

Snabb (formerly *Snabb Switch*) is a framework for fast packet processing [10]. It is an open-source framework published on GitHub. Its first commit was in 2014, and still, its open-source community improves Snabb to this day. The latest version is *Snabb 2024.06 "Faye"* released in June 2024. Snabb is mostly developed in Lua, leveraging the LuaJIT compiler [29]. Lua is an easy-to-learn high-level programming language [10]. LuaJIT is a just-in-time compiler for Lua. It optimizes the performance of Lua to be competitive with C. Furthermore, Snabb implements a kernel bypass. Therefore, no kernel overhead is imposed when running Snabb. To run Snabb, it has to be compiled into a single binary. Afterward, this binary has to be run with `sudo` permissions. Otherwise, no kernel bypass would be possible. Due to these properties, Snabb can be used to implement VNFs. Snabb already includes some VNFs. Examples are the traffic generator *packetblaster* or *lwAFTR* which is an implementation to support 4-over-6 tunneling. These examples show that Snabb can be successfully used to implement VNFs through a kernel bypass.

Snabb is built extensible to support custom network designs [30]. Therefore,

Snabb compromises two kinds of layers. The lower layer is the Snabb core. On top of this layer, a custom network design can be realized. These two layers are abstractions to separate the predefined and custom functionalities. Still, they need to be combined through interfaces to interwork with each other. In the following, the major components of both layers are presented. Figure 2.8 depicts these components and their relationships. The Snabb core is a runtime environment to execute the custom network design. This runtime environment is called *engine*. The custom network design can be thought of as a kind of *main routine*. It is a Lua script that drives the Snabb stack and specifies the *network* to execute. This network is the *app network* and should not be mistaken for the communication network that Snabb extends. The network consists of *apps* and *links*. Apps are one way to implement custom functionalities. Libraries are another way. They define collections of utilities. In contrast, apps define *executed* behavior. Apps are combined through *links* into a *network*. This is the network run by Snabb. The engine ensures that this network is running properly. It pushes packets through the network, restarts failed apps, and reports the network status. Snabb is extensible due to the separation of its core and the custom network that is run.

2.5.2. Apps

In this subsection, Snabb *apps* are further introduced. Therefore, the components, like fields and methods, are described. Afterward, an example of a simple packet-forwarding data plane is presented. This further elaborates on the introduced fields and methods used for apps.

Apps are used to specify the packet processing behavior in the custom network design. They represent an isolated implementation of a specific function [30]. Snabb defines in its reference an interface for its apps. This interface is like an abstract class in other programming languages. It specifies some accessible fields and mandatory or optional methods to implement. Two important fields are the `input` and `output`. Those fields are read-only and are initialized by the engine. They are tables containing named input and output links. Therefore, both can be thought of as maps, mapping input, and output link names to their instances. Input links are used to receive packets from other apps. Output links are used to transmit packets to other apps. Each app can have an arbitrary number of input or output links. Apps contain more fields, but they are not as relevant as these two. For the methods, two relevant methods are introduced in the following: `new` and `push`. The first method `new` is the only mandatory one. It is used to create a new instance of the declared app with an arbitrary number of arguments passed. The second method `push` is used to push packets through the network. This method is periodically called to process packets, e.g., receive some from an input and transmit them to an output [10]. Since this method can be programmed by the users, arbitrary packet processing functionality can be implemented. The two fields, `input` and `output`, and the method `push` are relevant for implementing custom network functionality through an app.

Listing 2.2 depicts a simple example of how to implement an app. Therefore, a simple forwarding app is introduced. First of all, an empty table (like an object) is declared with the name `Forward`. Afterward, the two functions `new` and `push`

```
Forward = {}

function Forward:new()
    return setmetatable({}, { __index = Forward })
end

function Forward:push()
    local iface_in = assert(
        self.input.input,
        "<input> not found"
    )
    local iface_out = assert(
        self.output.output,
        "<output> not found"
    )

    while not link.empty(iface_in) do
        local pkt = link.receive(iface_in)
        link.transmit(iface_out, pkt)
    end
end
```

Listing 2.2: Sample implementation of a forwarding app.

are declared as methods associated with `Forward`. The method `new` initializes the app and returns the instance. This is similar to the code provided in the *Getting Started* by the Snabb repository [10]. In the second method, `push`, two local variables containing the input and output links are declared. Both are accessed through the predefined read-only fields `input` and `output`. Named links with the same name as those fields are then accessed. Afterward, three methods associated with the core library `link` are used: `empty`, `receive`, and `transmit`. As their names imply, they are used to check if a link is empty, to receive packets from a link, or to transmit packets to a link. This app implements the behavior of sending packets received from the input link to the output link while the input link is not empty.

2.5.3. App Network

This subsection further elaborates on *app networks* in Snabb. First, it describes how app networks are built. In particular, the ingress in and egress out of a network are described. Afterward, an example is given of how a simple app network is configured. Thereby, the syntax used for this is presented.

An app network is used to combine multiple apps, which implement specific functions, to implement a desired behavior [30]. The app network consists of apps and links. Each link connects two apps. Links are implemented as ring buffers. The

```
local c = config.new()
config.app(c, "NIC", connectx.IO, ...)
config.app(c, "Forward", Forward, ...)

config.link(c, "NIC.output -> Forward.input")
config.link(c, "Forward.output -> NIC.input")

engine.configure(c)
```

Listing 2.3: Sample app network configuration of a simple forwarder connected to a NIC.

link library exposes API functions to interact with this ring buffer, e.g., to receive or transmit packets. Links are not used to receive traffic from the outside and push it into the network or to transmit packets out of the network. The ingress into the network and the egress out of the network are defined by specific apps. For example, a *source* app can be used as an *ingress* or *sink* as an *egress*. Both will create or consume the packets inside the network. Another example of an ingress or egress would be a *NIC* app, like the *ConnectX* or its *IO* app. It can be used for both ingress and egress into or out of the network. The NIC app does not generate or consume the packets by itself. Rather, it acts as an interface to a real NIC. Therefore, the packets received on this NIC are pushed into the network, and packets sent by the NIC app are sent by the real NIC. Links combine apps to an app network which contains ingress or egress apps into or out of the network to interact with the underlying communication network.

Listing 2.3 depicts an example configuration of an app network. This network consists of a NIC and a simple forwarding app. The NIC app is an interface to a ConnectX [27] NIC. First of all, a local configuration has to be declared. Afterward, all apps need to be configured. The configuration is done by specifying a name for the app, the app, and additional arguments expected by the app. A name is needed to identify the configured app. Its name is used to configure links. After the configuration of all apps, links need to be configured. Links are configured through string literals. The string literals contain an app name with a named output link and an app name with a named input link. Both links are connected by the char sequence `->`. Therefore, on the left of the `->` is an output connected to an input on the right. Lastly, the engine is configured according to this configuration.

2.5.4. Worker

In Snabb, workers can be used to leverage multi-processing for a higher packet processing rate [30]. When running Snabb, one single process is started. For this process, a custom network design is provided and is run by the engine. This single process is run on a single CPU core. Therefore, no true multi-processing is experienced. For real multi-processing, Snabb introduces workers. Workers are im-

plemented in a core library. Therefore, they are a native tool provided by Snabb to leverage multi-processing. To utilize workers, they need to be initialized by the single process running at the start. This process spawns the workers as child processes. Each worker can do everything the ordinary Snabb process can. For example, they can define different app networks and run them by the engine. This behavior is defined upon the creation of each worker. It is expressed by a string containing the function and arguments to be executed by the worker [10]. When the workers are running, they can be executed or pinned to arbitrary CPU cores. Therefore, true multi-processing is experienced. The ordinary Snabb process is still able to monitor or terminate its child processes. Worker processes can each execute different app networks while still being monitored by the ordinary Snabb process that spawned them.

2.5.5. ConnectX Driver

The *ConnectX* and its *IO* app are special drivers for a ConnectX [27] NIC, implementing a kernel bypass [10]. Currently, the drivers support the ConnectX 4-6 [10]. The *ConnectX* app serves as the driver to configure the NIC. To initialize the *ConnectX* app, the PCI address of the NIC and queues need to be provided. Additional fields, like the send and receive queue sizes, can be provided as well. Here, a queue is a Lua table, at least consisting of a queue `id`. Additionally, a `vlan` id and a `mac` address can be provided. The VLAN ID and MAC address are then used to map traffic with this VLAN ID and destination mac address to this specific queue. If two queues specify the same VLAN ID and MAC address, the traffic is load-balanced according to the 3-tuple or 5-tuple. The 3-tuple consists of the IP source address, IP destination address, and IP protocol field. The 5-tuple consists of the 3-tuple and additionally includes the Layer 4 source port and Layer 4 destination port. To attach to these queues, the *IO* app is needed. It can be initialized with the same PCI address used for the *ConnectX* app and one of the provided queue `id`s. This app is then used to process packets received by the corresponding hardware queue or to transmit traffic through it. To receive or transmit traffic, Snabb directly reads or writes packets from or to the NIC. Since no kernel networking stack is involved, this is part of implementing the kernel bypass. Due to the open-source character of Snabb, these apps can be modified. For example, the creation of the hardware receive queues can be altered to activate the VLAN stripping. With this configuration, the receive queue will strip the VLAN header, if present, and swap the ethertypes. Both apps can be configured or reprogrammed to match the desired behavior.

2.5.6. Foreign Function Interface

For better performance, Snabb supports a FFI to directly call C functions or to use C data structures [29]. Therefore, the FFI extension of the LuaJIT [29] is directly compiled into Snabb [10]. This FFI extension does not need an additional binding language. It directly parses C source code. However, it does not support all C language features, like pre-processing or some constant expression evaluations. Nevertheless, Snabb uses the FFI extensively. Two major examples are the packet and

link implementations. Both data types are declared in C header files as structs. The FFI is used to access the struct fields and to initialize or free instances. Furthermore, Snabb libraries provide APIs so that users do not need to interact directly with the C struct. These APIs are implemented in Lua. Due to this, the implementation of custom functionalities becomes more idiomatic. Users do not need to think about whether they are interacting with instances or pointers from C. Snabb uses the FFI to leverage its performance, while it provides APIs for the idiomatic implementation of custom functionalities.

3. Related Work

This chapter presents the related work for this thesis. First, the related work regarding the extension of programmable switches is presented. Programmable switches can be extended through external memory or compute to enable more complex network functions. Afterwards, the related work regarding the evaluation of VNFs is presented. Various papers exist that compare different software switches. But most of them only compare simple functionalities or packet switching between virtual assets, like VMs or containers, and not physical ones.

3.1. Extending Programmable Switches through external Elements

This section presents related work regarding the extension of programmable switches, like the Intel® Tofino™ , through external elements. These elements can be computation or storage elements. Many proposals use Remote Direct Memory Access (RDMA) [33] to extend their data plane with external memory. RDMA assumes reliable and in-order delivery. To ensure this, mechanisms for reliable data transfer are needed. TCP is a common protocol for implementing such mechanisms. It is an example of how these mechanisms impose a significant overhead to ensure reliable and in-order delivery.

In various papers, the extension of programmable switches by external memory is used. For example, the architecture *TEA* [16] provides virtual tables for programmable switches. These virtual tables are larger than regular ones. They are implemented on external DRAM and are accessed through RDMA. Further, Langlet et al. [18] introduce *Direct Telemetry Access*. This also leverages RDMA to access external memory. That memory is used to directly save telemetry reports into queryable data structures. In addition, Scazzariello et al. [35] introduce *RIBOSOME* which extends programmable switches with external memory and external compute. For external memory, RDMA is used. *RIBOSOME* leverages external Central Processing Units (CPUs) and FPGAs for compute. These proposals show how common and adapted the extension of programmable switches with external memory is.

For the implementation of network functions, it is relevant where the external element running the network functions is located. The location imposes restrictions and overheads. Kianpisheh et al. [15] describe two disadvantages of in-network implementations of network functions. Co-designed schemes can solve the capability restrictions of in-network implementations. They leverage general-purpose compute elements and are located beside network elements. Therefore, they can extend the capabilities, while imposing a little overhead. Kianpisheh et al. mention the paper

of Mai et al. [23] that leverages this scheme. They propose an in-network computation on programmable switches for simple tasks and a mobile edge computation for more intense computing tasks. Their solution proposes a mobile edge computation that is not directly located next to all programmable switches. Therefore, the extended capabilities are not *directly accessible* from all switches. In addition, Zeng et al. [43] introduces an architecture called *Tiara*. Here, programmable switches are used for the fast path implementation of a Layer 4 load balancer. Commodity servers are co-located to implement a slow path. Therefore, this architecture implements a co-designed scheme as well. These proposals show that the co-location of general-purpose compute is not new. They either co-locate the external element but implement a slow-path behavior, or they locate the element farther away and implement a fast-path behavior. None of these proposals co-located the external element *and* implemented a fast-path behavior.

3.2. Virtual Network Function Evaluations

In the following, related work presenting different VNF evaluations is presented. A special focus was on related work evaluating either Snabb or a ConnectX 5 NIC.

Paolino et al. [31] first proposed Snabb in their paper. They proposed Snabb as a virtual switch for NFV infrastructure. Therefore, it should be used for switching between VMs or NICs. They further included evaluations comparing the throughput of Snabb compared to other NFV switches. In their experiments, they conducted unidirectional and bidirectional experiments for a single VM or VM to VM test case. Especially the bidirectional single VM experiment is highly relevant to this thesis. Snabb achieved nearly as good results as other NFV switches. The key difference in the work of Paolino et al. [31] in contrast to his work is that they use 10 Gbps as the traffic rate and use Snabb to forward the traffic to a VM. In contrast, this work will use higher traffic rates and will run Snabb on a bare-metal server. Further, Snabb implements VNFs as externs for a P4-programmable switch and is not only used for traffic forwarding.

Around the same time, similar experiments were conducted using DPDK. Kourtis et al. [17] present an evaluation of a deep packet inspection implementation implemented with DPDK. Once again, a traffic rate of 10 Gbps was used. However, two experiments comparing the throughput of a physical testbed and a virtual one were conducted. The virtual testbed only achieved approximately 81% of the physical testbed. Therefore, the implementation running directly on the bare-metal server had significantly greater performance. In contrast to this thesis, their paper implemented a different use case and used a different tool to implement their VNF. Still, their evaluations showed that an implementation should rather run directly on the bare-metal server than inside a VM.

The benchmarks for DPDK further shed light on what DPDK is capable of. The last benchmark, including a ConnectX 5 NIC, was conducted in 2023 [37]. They used twelve cores; each core had one receive queue, a transmission rate of 100 Gbps, and 8192 IP flows. Further, they used Layer 3 forwarding for this test. With these configurations, the benchmark should measure the highest zero-packet loss rate.

This is the rate at which no packet loss occurs. Even with these configurations, it was not possible to achieve a throughput of 100 Gbps for the zero-packet loss rate. These results are not representative since a great number of CPU cores and IP flows were used. Nevertheless, this benchmark shows that even DPDK is not able to fully utilize the NIC.

A similar experiment was conducted in 2022 using Snabb as the software switch for a ConnectX 5 NIC [34]. Three major experiments were run. One measures the maximum transmission rate. Another measuring the maximum receive rate. Both are not as relevant since they either generated packets by an app and transmitted them, or they received packets and an app dropped them immediately. Therefore, no forwarding or more sophisticated VNF was applied. The third experiment, measuring the forwarding rate, is more relevant. Here, packets were received and then transmitted back. This experiment was conducted with six to twelve workers, each running on its own core. Even those cannot perform the forwarding at a line rate of 100 Gbps. These experiments show that even with a large number of workers and simple functionality implemented, the line rate cannot be reached.

Further work was done comparing different software switches. In 2019, Ara et al. [3] compared different software switches connecting co-located containers. These experiments presented Snabb as the best software switch. One year later, Ara et al. [4] published a more elaborate evaluation. In this evaluation, they presented Snabb as the worst-performing software switch concerning its throughput and delay. This contradicts their first observation and questions the experiments they conducted in the first place. Another year later, Zhang et al. [45] presented their evaluations of seven different software switches. Once again, the software switches connected co-located virtual assets, like VMs or containers. But they included a physical-to-physical experiment as well. This physical-to-physical experiment forwards packets between two NICs through the software switch under test. In their experiments, no switch was the best in all scenarios. The physical-to-physical experiment is the most relevant one. In this experiment, Snabb achieved a throughput of 8.74 Gbps out of 10 Gbps. Therefore, it scored the third-last rank. In all of these experiments, only simple forwarding was evaluated, with no more complicated functionalities. Therefore, those can only be used as a baseline and not as a reference for the capabilities of each switch.

4. Concept and Implementation

This chapter presents the actual work done in this thesis. It includes the concept as well as the implementation of the work done. First, the proposed architecture to extend P4-programmable switches with externs, implemented at a co-located server, is presented. Then, the signaling used between the switch and the bare-metal server is described. Afterward, the usage of the signaling for load balancing is presented. Lastly, the externs implemented in Snabb are introduced.

4.1. Architecture

This section introduces the proposed architecture for extending the capabilities of programmable switches through external network functions. First, the general architecture and the necessary systems are introduced. Then, it is described how packets are forwarded in this architecture. Afterward, the usage of Snabb to implement the external network functions is described. Especially the usage of the receive and transmission queues. Lastly, the parallel processing in Snabb through multiple workers, queues, and NICs is described.

An extendable and transparent architecture is needed to extend the capabilities of programmable switches through external network functions. Figure 4.1 depicts the proposed architecture of this thesis. Two different systems are needed for this architecture: a P4-programmable switch and one or multiple co-located bare-metal servers. Packets from hosts are forwarded to the programmable switch. The control plane of the switch configures the behavior for this traffic. It can configure traffic to be only processed by the switch or to extend its capabilities through external network functions. These external network functions are hosted on the bare-metal server. The server can configure an arbitrary number of network functions, unless its hardware and the signaling between the server and switch support this number. Therefore, the provided external network functions are extendable. The configured traffic is then forwarded to the server to be further processed. This extension is transparent for the hosts because the signaling is only local between the switch and the server. Therefore, it makes no difference for the hosts if their packets are processed only by the switch or additionally by the server. After processing, the server sends the packets back to the switch. According to the configured behavior, the switch then further forwards the packets to their next destination. This architecture provides an extendable and transparent extension of the capabilities of a P4-programmable switch through a co-located bare-metal server.

Snabb is used for an extensible and fast implementation of external network functions. For higher throughput, Snabb reads and writes packets directly to and from the NIC through its driver. The ConnectX driver is used in the prototype of this thesis. This driver can configure one or multiple receive and transmission queues.

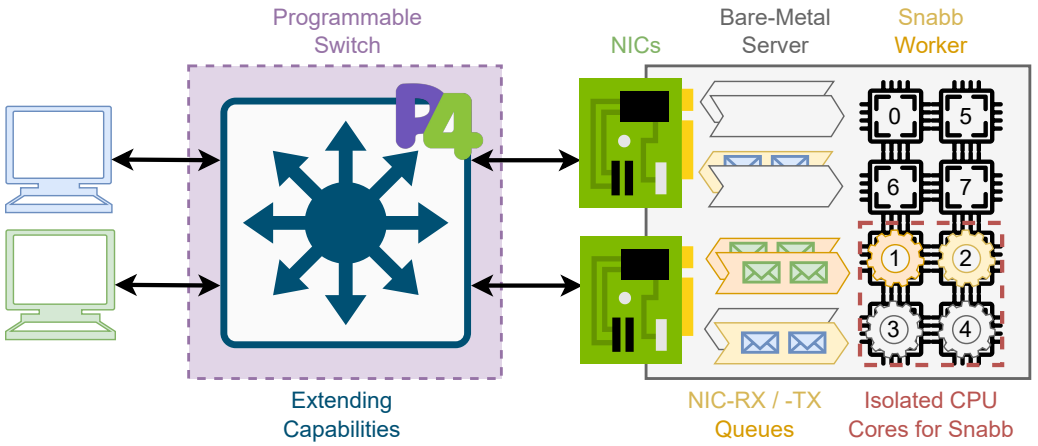

Figure 4.1.: Detailed architecture to extend the capabilities of a programmable switch.

Figure 4.1 depicts these queues as *rectangles with one outer and one inner tip* between the NIC and the CPU cores. The *rectangles* pointing towards the CPU cores are the receive queues, and the ones pointing towards the NIC are the transmission queues. These queues are directly configured on the NIC. Each packet received at the NIC is placed in one of those receive queues. The NIC can either place the packets according to the 3-tuple or 5-tuple or through a static VLAN mapping. Therefore, every receive queue is associated with a specific VLAN ID. The packet is placed in this queue if it has this VLAN ID. If no corresponding VLAN ID or header is present, the packet is placed in a default queue. This architecture proposes the use of such VLAN-associated receive queues. Every queue is further used in Snabb for a specific extern. Therefore, the VLAN IDs are used to signal, which Snabb implemented extern has to be applied.

The server implementation can be further enhanced to support parallel processing. Therefore, each extern is running on its own worker. Each worker is further pinned to a specific isolated CPU core. This ensures that no other process is scheduled on this core and interferes with the extern. Figure 4.1 depicts the workers as gears. They are only pinned to the isolated CPU cores framed in red. Further, each worker can execute a different extern, or multiple workers can execute the same extern. The last is used to distribute the load among multiple workers. Further, the load can be distributed to one worker through multiple queues, e.g., one worker can process the packets received by two queues. Thereby, it is possible to receive packets on one queue and transmit them on another. The only restriction is that the same worker needs to be processing both queues. If the NIC itself becomes the throughput bottleneck, multiple NICs can be used. Then the load can be distributed among all NICs. Even the reception of packets on one NIC and the transmission on another is possible. This is depicted by the yellow worker in Figure 4.1, which receives packets on the lower NIC and transmits them on the upper NIC. But the same problem as for queues holds: this is only possible if the worker has access to queues on those different NICs. Parallel processing is achieved through multiple workers, which can

serve multiple queues and NICs.

4.2. Signaling between Switch and Server

The switch needs to signal which extern needs to be applied to a packet. Therefore, the switch needs to add protocol headers that indicate the VNF implementing the extern to be applied. This section introduces the signaling proposed between the switch and the server. The switch uses a VLAN header to indicate to the server which extern needs to be applied. Further, a custom Snabb header is proposed to supply additional data. In the following, the signaling header stack and its fields are described. Afterward, the signaling in the architecture will be elaborated.

4.2.1. General Signaling

In this subsection, the signaling between the switch and the server is described. At first, the proposal of only a custom Snabb header is introduced. Then, it is described why this does not work and why an additional VLAN header is needed. Lastly, the signaling header stack and the header fields are described.

For the signaling between the switch and the server, a VLAN header is used. Further, a custom Snabb header is proposed. At first, only a custom Snabb header was intended. This Snabb header was placed between the Ethernet and the IP header. It consisted of an ethertype, opcode, and length. According to the length field, optional data was included. The length specifies how many bytes the data has. The data can either be extracted by the data plane or configured by the control plane. This proposal could not be distributed to different receiving queues at the NIC. A distribution is only possible with a VLAN ID, the 3-tuple, or the 5-tuple. With this protocol stack, none of these three is accessible by the NIC. Therefore, all traffic is mapped to the default queue, and no parallel processing is possible. Because of the inability to handle the custom Snabb header and therefore no distribution across receive queues, an additional VLAN header is proposed.

Figure 4.2 ❶ depicts the proposed signaling header stack. First, an Ethernet header is added to the original packet to send the traffic from the switch to the server. Afterward, a VLAN header is added. This imposes an overhead of 4 bytes. A VLAN header has multiple fields, two of which are a VLAN ID and an ethertype. The VLAN ID is proposed to identify the receive queue. Running multiple VNFs and therefore multiple externs on one queue and worker imposes a resource and processing overhead. This overhead can be avoided. Only one extern should be run on one queue and worker. Therefore, the VLAN ID now identifies not only the receive queue but also the extern. The section 4.3 “Load Balancing” will further elaborate on how the VLAN ID identifies an extern. Due to this, the resources of the queues are not shared. Further, no additional processing is needed to parse and extract the originally proposed opcode of the Snabb header since the VLAN ID already specifies the extern to apply.

After the VLAN header, a (now optional) Snabb header can be added. Figure 4.2 ❷ depicts the header stack containing the optional Snabb header for additional data. This custom Snabb header is indicated with the ethertype `0x54BB` (read as

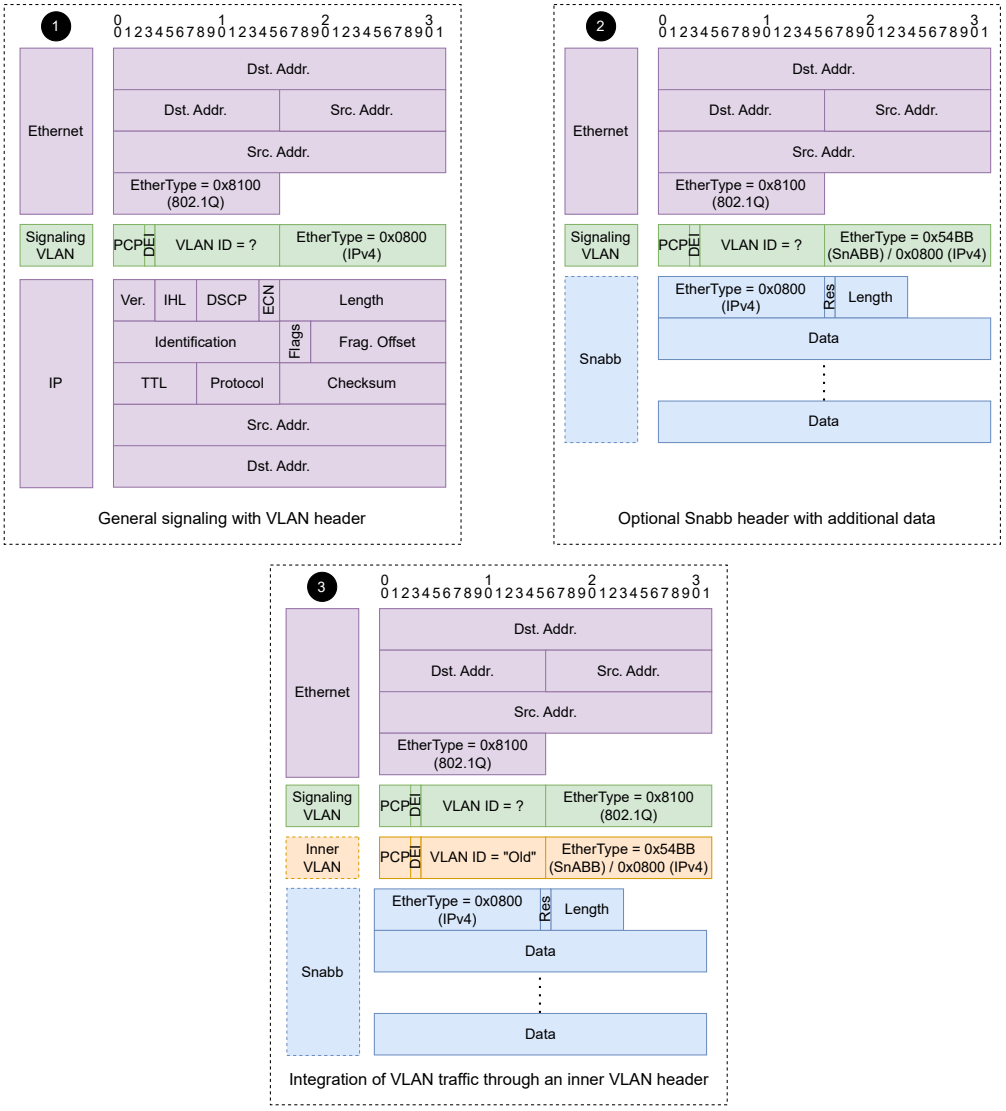

Figure 4.2.: Signaling header stacks between switch and server.

SnABB). The use of the VLAN ID to identify the extern leads to an unnecessary opcode field in the custom Snabb header. Because of this, the final Snabb header does not include an opcode field. Therefore, only the ethertype, the data length, and the optional data are included in the Snabb header. If no data is present, the whole Snabb header is excessive since no information is carried. Therefore, the whole Snabb header is now defined as optional. Additionally, the data field is now mandatory in the optional Snabb header. Therefore, if a Snabb header is used, data needs to be present. The length of the data can be between 1 and 127 bytes. The additional overhead imposed by the Snabb header is between 4 and 130 bytes. With the VLAN header, this adds up to an overhead of 8 to 134 bytes. The Snabb data can either be set by the control plane or extracted through the data plane with special actions. After the optional Snabb header, the original Layer 3 header,

like an IP header, is preserved. For simplicity, the Layer 3 header is not shown in Figure 4.2 ❷. The general signaling only includes one mandatory VLAN header and an optional Snabb header containing additional data.

4.2.2. Integration of VLAN Traffic

In this subsection, the integration of VLAN traffic and, therefore, the implementation of stacked VLANs are described. First, the standardized concept of stacked VLANs (IEEE 802.1ad) is presented, and why this cannot be used. Afterward, the signaling header stack, including the custom integration of stacked VLANs, is introduced. Especially, the changes to the first proposed signaling header stack are pointed out.

Traffic sent to the switch may already contain a VLAN header. Every traffic being sent from the switch to a server needs to comply with the signaling introduced in subsection 4.2.1 "General Signaling". Therefore, the traffic being sent may need to have two VLAN headers. To support multiple stacked VLAN headers, IEEE introduced *QinQ* in the *IEEE 802.1ad* standard. The P4TG supports the generation of QinQ traffic. This traffic was used to test if the *Mellanox ConnectX 5 NIC* in combination with the Snabb *ConnectX* app supports QinQ. With this test, it was confirmed that the ConnectX app does not support QinQ, even though the ConnectX NIC supports multiple VLANs. To be more precise, if VLAN stripping is enabled, the resulting frames are invalid. The inner VLAN header is being stripped, and therefore a QinQ header is present without an inner VLAN header. Because of the lack of support for QinQ by the ConnectX 5 NIC, a workaround is needed to support the integration of VLAN traffic into the signaling.

Figure 4.2 ❸ depicts the proposed signaling header stack, which integrates VLAN traffic. Like in Figure 4.2 ❷, the IP header is neglected for simplicity but is still present. In contrast to Figure 4.2 ❷, now two VLAN headers are present instead of only one. Both figures depict the signaling VLAN header in green, which is placed right after the Ethernet header. The original VLAN header is depicted in orange and is placed in between the signaling VLAN header and the optional Snabb header. It is used as the *inner VLAN* header. Depending on whether a Snabb header is used or not, the original VLAN header needs to be modified. If a Snabb header is used, the ethertype of the original VLAN header needs to be set to the Snabb ethertype. The original VLAN ethertype is then placed into the Snabb header. After a packet is processed by an extern, the switch needs to undo this modification. Further changes are needed for the signaling VLAN header. Originally, it contained the Snabb or Layer 3 ethertype. Now, the ethertype of the signaling VLAN header always indicates the inner VLAN header through the VLAN ethertype `0x8100`. VLAN traffic can be integrated by using two VLAN headers, replacing the ethertypes accordingly, and using the VLAN ethertype for the signaling VLAN header.

4.2.3. Signaling in the Architecture

The in Figure 4.2 presented header stacks are pushed by the switch for the signaling to the server. It signals which receive queue, and therefore which extern, is applied

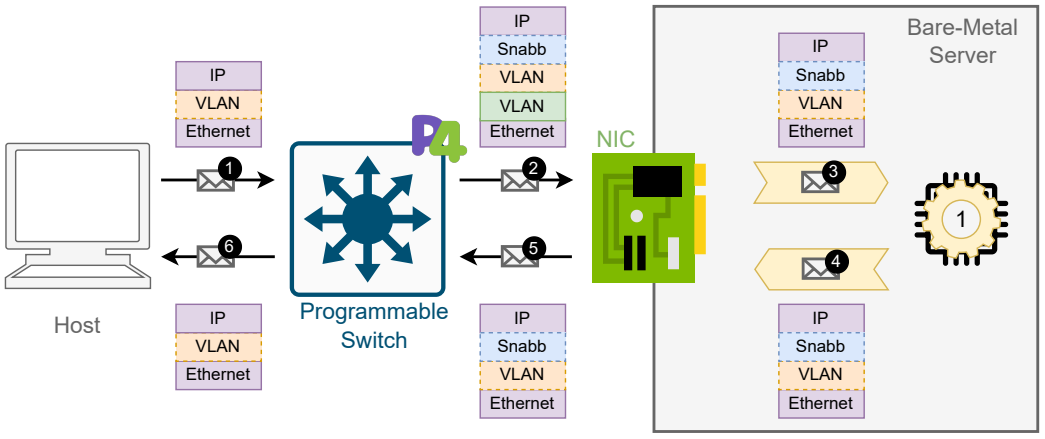

Figure 4.3.: Signaling header stack forwarded in the architecture.

to the packet. This signaling can be deployed between the switch and one or multiple servers. VLAN IDs identifying the extern to apply need to be unique only in the context of one server. If multiple servers implement externs, they can reuse the VLAN IDs used by other servers.

Figure 4.3 depicts the forwarding of a packet sent by a host. The packet is sent to a switch, which then forwards the packet to an extern implemented as a VNF on a server. The original packet sent consists of an Ethernet, IP, and optional VLAN header. All headers have the color scheme used in Figure 4.2. Further, optional headers are drawn with dashed lines. The host sends this packet to the P4-programmable switch (1). The switch is configured by the control plane to apply an extern implemented on the server. Therefore, the switch needs to signal the server, which VNF needs to be applied as the extern. For the signaling, the switch pushes a new VLAN header as the signaling header (2). This new VLAN header is drawn in green. If additional data is required, the blue-drawn Snabb header will be pushed as well. The NIC of the server receives the packet. It is configured to strip down the outermost VLAN header. This results in a packet without the signaling VLAN header (3). Further, the ethertype of the Ethernet header is replaced by the ethertype of the VLAN header. Due to this behavior, the VLAN header does not further impose any overhead on the Snabb implementation or the link back to the switch. After stripping down the VLAN header, the NIC enqueues the packet to a receive queue. Then, Snabb applies the extern. Afterward, the processed packet is enqueued in the transmission queue (4) and then sent back to the switch (5). At the switch, the optional Snabb header is stripped down (6). Now, all signaling headers are removed. The packet is now further sent to its next destination. The signaling is transparent for all other systems since only the switch and server need to know about the signaling headers.

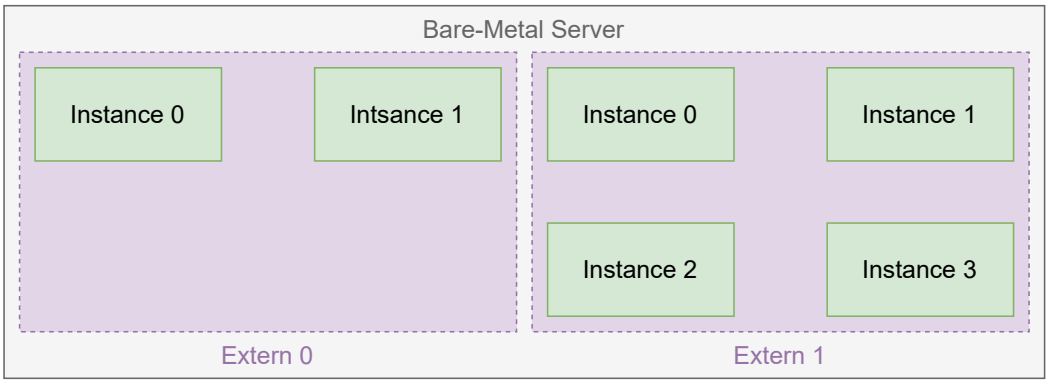

Figure 4.4.: Logical organization of externs and their instances.

4.3. Load Balancing

This section introduces how the traffic processed by the server can be load-balanced across different instances implementing the same extern. First, the need for the introduction of custom load balancing is presented. Afterward, the idea for the hierarchical organization of the VLAN ID based on IP subnetting is introduced. Lastly, the calculation of the signaling VLAN ID is described.

4.3.1. Hierarchical Organization of the Signaling VLAN ID

In subsection 4.2.1 “General Signaling”, the use of the VLAN ID to identify the extern to apply was introduced. This subsection further elaborates on the structure and organization of the VLAN IDs. Especially the distribution among multiple VNFs implementing the same extern is described.

Different VLAN IDs can be used to identify different externs. This enables the parallel processing of packets through different externs. Another desirable property is the parallel processing of multiple VNFs implementing the same extern. Multiple VNFs implementing the same extern are further called *instances*. Snabb already introduces load balancing to enable this property. Multiple receive queues, and therefore externs can share the same VLAN ID. The traffic is load-balanced according to the 3-tuple or 5-tuple among these queues. This mechanism does not work if a Snabb header is present. The NIC could not access the 3-tuple or 5-tuple, and all traffic is enqueued to the default queue. This results in no load balancing across instances. Therefore, custom load balancing needs to be implemented.

The custom load balancing needs to distribute the traffic uniformly among all instances. Externs and their instances can be logically organized, as depicted in Figure 4.4. The different externs are shown in violet and are separated by the dashed lines. These externs are run as *instances*. Instances are depicted in green. The externs define the behavior. Each instance executes a specific behavior. An instance is a process running the code implementing one extern. Therefore, the instances implement the externs as VNFs. This logical organization introduces a kind of hierarchy. Externs are the high-level behaviors, and their instances are the

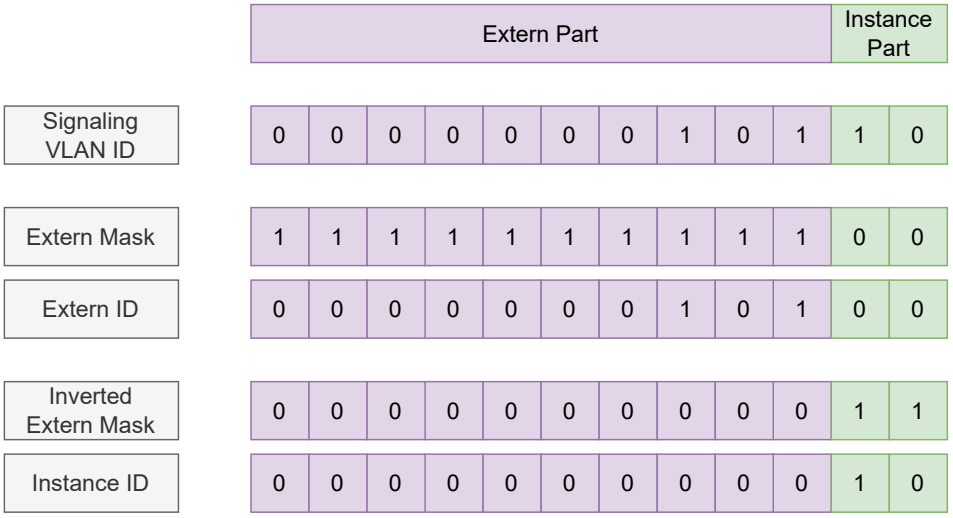

Figure 4.5.: Subdivision of the signaling VLAN ID; its parts, IDs, and masks.

lower-level processes. This hierarchy can be represented in the signaling. The VLAN ID identifying the extern to be applied can be organized hierarchically. Therefore, the VLAN ID is subdivided into multiple parts.

The subdivision into multiple parts is already used for IP subnetting. An IP address is a 32-bit value. These 32 bits can be subdivided into a network and a host part. A subnet mask defines how many bits are used for the network part. The network part is used to identify the destination subnet. The rest of the 32 bits are used to identify the destination host inside the subnet. This idea is transferred to the organization of the 12-bit-wide VLAN IDs. A VLAN ID has an *extern part* and an *instance part*. The extern part identifies which extern should be applied. The instance part defines the instance, which applies the extern. Therefore, the *extern part* is equal to the network part, and the *instance part* is equal to the host part in IP subnetting.

Figure 4.5 depicts the subdivision of the signaling VLAN ID. Especially, all different parts, IDs, and masks are shown. The length of the network part is defined by a network mask in IP subnetting. The same holds for the *extern part* for the signaling VLAN ID. An *extern mask* is used to define the length of the extern part. The bits of the extern mask are set as follows: Every bit in the extern part is set to one, and every bit in the instance part is set to zero. In IP subnetting, the IP address and subnet mask can be used to calculate the network address. The same calculation can be used to calculate the *extern ID* out of the extern mask and the signaling VLAN ID. The extern ID identifies a set of instances implementing the same extern. It corresponds to the ID present in the extern part of the signaling VLAN ID. Therefore, all bits set to one in the extern part of the signaling VLAN ID are set to one in the extern ID as well; all others are set to zero. The ID present in the instance part is defined as the *instance ID*. It can be calculated out of the *inverted* extern mask and the signaling VLAN ID. All bits set to one in the instance

part of the signaling VLAN ID are set to one in the instance ID; all others are set to zero. The instance part of the signaling VLAN ID is used for load balancing among multiple instances implementing the same extern.

4.3.2. Calculation of the Signaling VLAN ID

The switch pushes a VLAN header to indicate to the server which extern needs to be applied. To identify the extern, the extern part of the signaling VLAN ID is used. The ID present in the extern part is called the *extern ID*. For load balancing, the instance part of the signaling VLAN ID is used. The *instance ID* is the ID present in the instance part. In the following, the configuration and calculation of the signaling VLAN ID are described. Especially, the uniform distribution among multiple instances is presented. Lastly, static load balancing is described.

The control plane needs to add a MAT entry to forward traffic to the server and apply an extern. The MAT entry matches the destination IP address and the ingress port, each ternary. Therefore, different IP destinations, and especially the difference in whether the traffic was sent by the server or not, can be distinguished. Each MAT entry needs to supply three action data values:

Egress port to send data to.

Extern mask to define the length of the extern part in the VLAN ID.

Base VLAN ID which is used in combination with the configured extern mask to calculate the extern ID. The extern ID specifies which extern needs to be applied.

The signaling VLAN ID can be calculated with this action data and the packet headers. Figure 4.6 depicts the calculation of the signaling VLAN ID. All IDs and masks have the same width and color schemes for their parts, as depicted in Figure 4.5. The supplied action values are depicted in orange. The *inverted extern mask* and the *flow hash* are depicted in blue. Both are intermediate results. The final IDs, especially the final *signaling VLAN ID*, are depicted in gray.

To calculate the extern ID identifying the extern to apply, the configured extern mask and base VLAN ID are used. Both are combined through the bitwise AND operation; see Figure 4.6. This results in the extern ID, in which the bits that are already set to one in the extern part of the base VLAN ID are set to one. The instance ID is used for load balancing among multiple instances implementing the extern. Typically, the 3-tuple or 5-tuple is used to distribute the load. A common approach is the calculation of a hash based on the fields in these tuples. The hash uniformly distributes the calculated value across the output space. If the output space has the same dimension as the instance part, the hash uniformly distributes the value across the instance part. This approach is used for custom load balancing. The `Hash` extern of the TNA is used to calculate a hash based on the 3-tuple. As the hashing algorithm, the pre-defined `CRC32` is used. The output space is limited to the size of a VLAN ID, which is 12 bits. This calculated hash is the same for each packet in one flow. Therefore, this hash is called the *flow hash*. To calculate the instance ID, the flow hash and the inverted extern mask are used. Both are

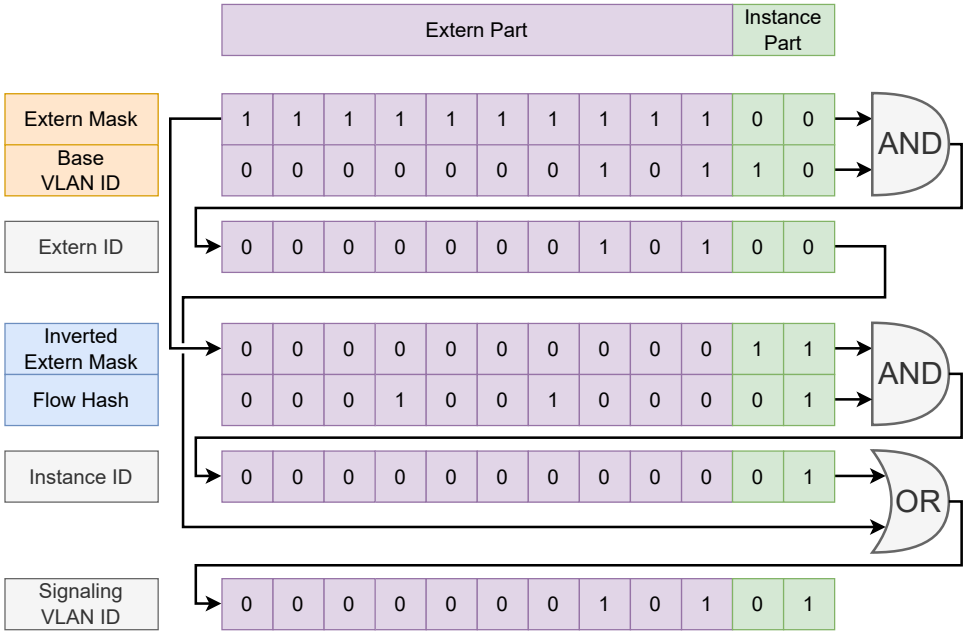

Figure 4.6.: Calculation of the signaling VLAN ID.

combined through the bitwise AND operation; see Figure 4.6. This results in the instance ID, where only the bits are set to one which are already set to one in the instance part of the flow hash. The *signaling VLAN ID* is now calculated based on the calculated extern ID and the calculated instance ID. Both are combined through the bitwise OR operation; see Figure 4.6. This results in the signaling VLAN ID, where only the bits are set to one, which are set to one in either the extern ID or instance ID. The calculated signaling VLAN ID is the VLAN ID of the signaling VLAN header added by the switch.

Another approach for load balancing is static load balancing. A static load balancing can be achieved by a special configuration of the extern mask. The extern mask needs to have every bit set to one. This results in the instance ID being zero. Therefore, only the extern ID is used to calculate the signaling VLAN ID. To be more precise, the extern ID is the same as the configured base VLAN ID. This can be used to specify the extern part as well as the instance part of the signaling VLAN ID. Therefore, the base VLAN ID defines the signaling VLAN ID statically. When using static load balancing, the base VLAN ID specifies which extern is applied *and* which instance applies the extern.

4.4. Parsing of the Snabb Header Data in P4

This section introduces the custom algorithm to parse the data in the Snabb header. First, the approach supported by P4 is described, and why this cannot be applied. Afterward, the custom approach and its origin based on the calculation of binary values are described. Lastly, an alternative algorithm with different parsing states

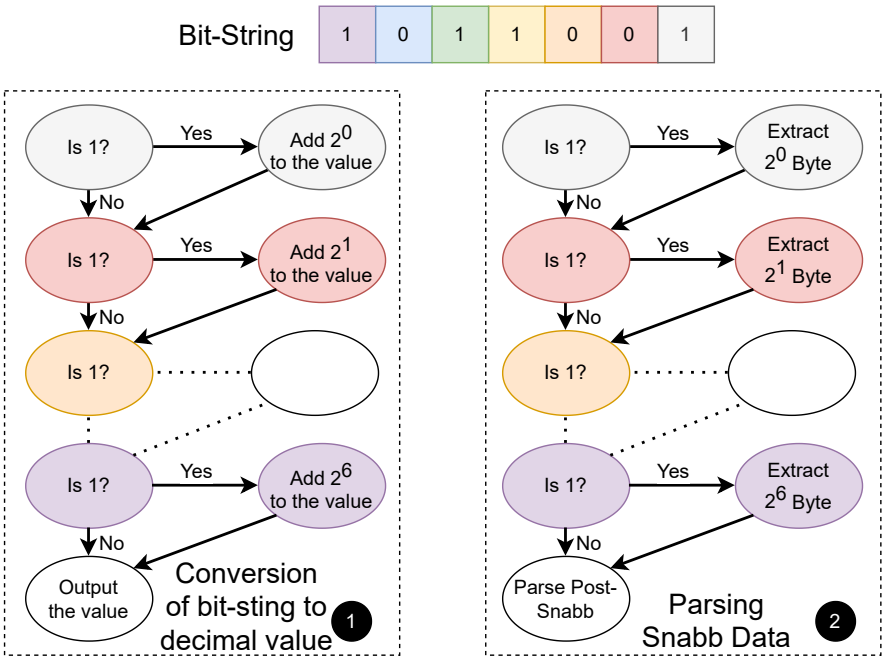

Figure 4.7.: Conversion of a bit-string to a decimal value vs. Snabb data parsing.

is presented. This alternative confirms that the proposed algorithm has good state-scaling properties.

The P4-programmable switch pushes the signaling VLAN header and, optionally, a Snabb header containing additional data. Later, the switch needs to parse the optional Snabb header and strip it down. The Snabb data is between 1 and 127 bytes long. Therefore, a dynamic way of parsing the data is needed. P4 offers header stacks for parsing variable-length data. Header stacks are a kind of array. They contain multiple values of the same header type. Therefore, a header stack containing 127 byte-headers is needed. Since each header includes the validity field, this would result in a larger overhead. Further, the use of a header stack imposed unexpected side effects. Because of these reasons, a custom parsing algorithm and header organization for the variable-length data are proposed.

Figure 4.7 depicts a bit-string. The index of the bits is i. The Least Significant Bit (LSB) has the index $i = 0$. In this example, the Most Significant Bit (MSB) has the index $i = 6$. Further, Figure 4.7 (1) depicts the conversion of the value of the bit-string into a decimal value. To calculate the value, each bit needs to be inspected. Starting from the LSB, each bit is checked to see if it is set to one. If it is set to one, then 2^i is added to the value; otherwise, nothing is added. When all bits are inspected, the final value is computed. This value is then output.

A similar algorithm is proposed to parse the variable-length data in the Snabb header. This algorithm requires seven different headers instead of 127 same headers when using a header stack. Each header is of a different size in bytes. One header contains one byte, one contains two bytes, one contains four bytes, and so on until

the last one contains 64 bytes. Figure 4.7 ❷ depicts the parsing of the Snabb data. The algorithm is the same as for the conversion of the binary value into a decimal value. Each bit is inspected from the LSB until the MSB. If one bit is set to one, 2^i bytes are extracted into the header containing that many bytes; otherwise, no data is extracted. Once all bits are checked, the Snabb data parsing has finished, and the data is stored in the headers. Due to the similarity of the algorithms in ❶ and ❷, the correctness can be inferred from ❶. Therefore, this algorithm can parse any Snabb data length between 1 and 127 bytes while having less storage overhead than a header stack.

This algorithm scales with the number of bits used for the length of the Snabb data and not the maximal number of bytes in the data. Another approach is the use of one state for each distinct length. This would result in as many states as different lengths. If the length is n bits long, this would result in $\Theta(n) = 2^n$ different states. The proposed algorithm scales with the number of bits used for the length. The number of states needed is $\Theta(n) = n$. Therefore, the number of states scales linearly in the proposed algorithm instead of exponentially with the number of bits used for the length.

4.5. Externs in Snabb

In this section, two externs implemented in Snabb are introduced. First, the general implementation concept is described. Especially the app network is introduced, which will be the same for both externs. Afterward, the *Delayer extern* and the *POF externs* are described. The *Delayer* delays packets according to a configured delay and jitter. The *POF* orders out-of-order packets while supporting possible packet loss.

4.5.1. General Implementation Concept

This subsection introduces the general concept of externs being implemented in Snabb. Especially the used apps, the proposed app network, and the number of workers used are introduced.

The externs implemented in this thesis all follow one high-level concept. This concept is depicted in Figure 4.8. At the start of the Snabb program, a single process is started. This is the *main process*. The main process is depicted in orange. First, it initializes the NIC driver apps. Every app of the same process or worker is depicted in the same color. In this case, the ConnectX apps. The *NIC In* and *NIC Out* can be the same NIC. In this case, only a single app is used. Afterward, the main process initializes one worker process for each extern instance.

Figure 4.8 depicts worker processes in gray. Every worker runs a single VNF implementing an extern. Further, multiple workers can run different or the same extern. Therefore, each worker is an *instance*. Each worker implements its own app network. The app network consists of two to three apps. One app is always the one implementing the extern. The IO apps are the Snabb representation of the receive and transmission queues. The number of *IO* apps depends on the number of NICs

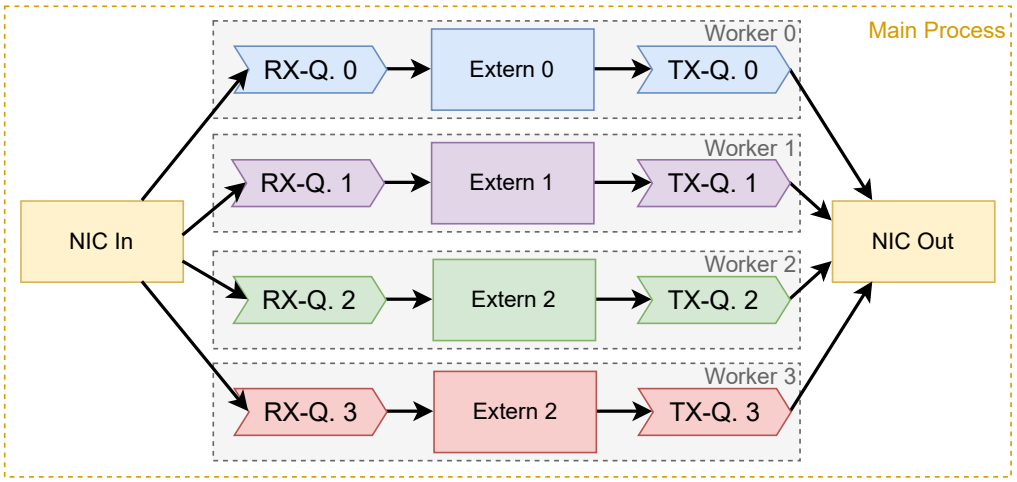

Figure 4.8.: General concept of the Snabb externs regarding the apps, app networks, and workers.

used. If two different NICs are used, two IO apps are present in the app network. Still, each worker only uses one receive queue and one transmission queue. This results in really short app networks. Further, each network is a straight network without any branches. This app network is proposed because it only imposes a little *network* overhead. If a longer or more complicated app network is used, this imposes some overhead. The engine running the network pushes packets through the network. The more apps a network has, the more `push` methods need to be called. Another overhead is imposed by branches in the network. If the engine pushes packets first through a *branch* app, this results in packets from the *root* app not being pushed until the point they could. It gets more complicated if the network forms a loop. Therefore, each worker runs a simple and short app network.

4.5.2. Delayer

In this subsection, the delay extern *Delayer* is introduced. This extern enables packets to be delayed. Therefore, a delay and jitter can be configured. In the following, the concept of the Delayer is introduced. Afterward, the implementation is described.

The Delayer is an extern implemented in Snabb. To be more precise, the Delayer is an app implementing the extern. It consists of a configured *delay*, a configured *jitter*, and a *buffer*. The delay is the mean of the time a packet is delayed. The jitter is the standard deviation of the time a packet is delayed. Therefore, the delay (mean) and jitter (standard deviation) define a continuous distribution of the time a packet is delayed. For the concept of the Delayer, a continuous, uniform distribution of the time that a packet is delayed is assumed. The buffer is used to buffer tuples consisting of packets and the time they are going to be sent. These three members (delay, jitter, and buffer) are the only ones needed for the Delayer app.

Figure 4.9 depicts the concept of the Delayer as an activity diagram. It depicts

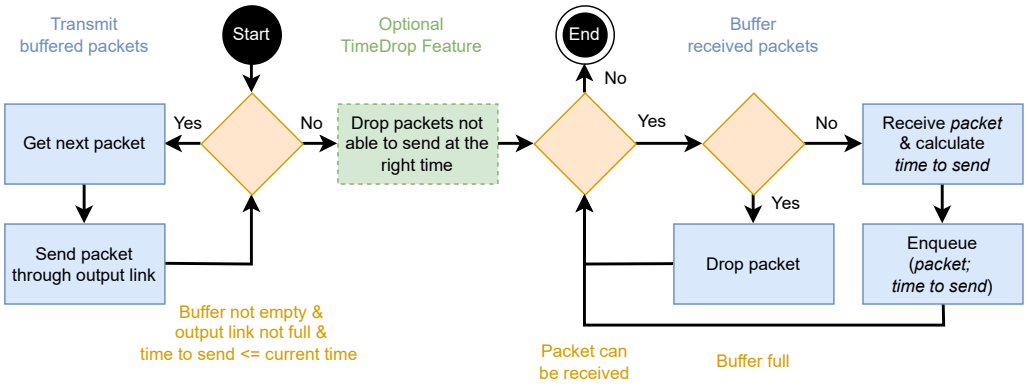

Figure 4.9.: Activity diagram of the concept of the Delayer extern.

branching behavior in orange and executed behavior in blue. The activity diagram depicts one iteration of the engine pushing packets through the app. In each iteration, the engine begins at the *start*. When reaching the end, the engine pushed all packets as much as possible. Then it starts pushing packets through other apps. Beginning at the start, the Delayer is invoked. It starts running its custom behavior. The first part is the transmission of buffered packets. It is depicted on the left. The Delayer checks if packets are buffered. If packets are buffered, it checks whether the output link is full. If no packets are buffered or the output link is full, the Delayer continues with its second part. Otherwise, the Delayer checks if the buffered packets are destined to be sent. Therefore, it checks if the stored *time to send* is less or equal to the current time. If this is true, the packet is destined to be sent. Then, the Delayer retrieves the packet. Afterward, it sends the packet through the output link. This behavior continues until one of the conditions above is not met. If one of the conditions is not met, the second part of the Delayer is executed.

Figure 4.9 depicts the second part of the Delayer on the right. The second part of the Delayer receives packets and buffers them if possible. Therefore, the Delayer checks if a packet can be received. If no packet can be received, the Delayer reaches its end. Otherwise, it checks if the buffer is full. If the buffer is full, the packet is received and dropped immediately. Otherwise, the Delayer buffers the packet. This continues until no packet can be received anymore. The buffering works as follows: First, the Delayer receives a packet. Then, it calculates the *time to send* the packet. This time to send is calculated based on the current time, the delay, and the jitter. The delay and jitter are used to generate a random variable. This random variable represents the time to delay the packet. It is generated by a continuous, uniform distribution. Afterward, the tuple `(packet; time to send)` is enqueued to the buffer. Because of the uniform distribution of the packet delay, later-enqueued packets may need to be sent before earlier ones. Therefore, the buffer is sorted by the *time to send*. The Delayer stores the received packets in a data structure that allows the retrieval of the packet with the lowest *time to send*.

In between these two parts, an optional feature can be executed: *TimeDrop*. This feature is depicted in green. Packets are delayed too long if the first part of the

Delayer is stopped because of the insufficient capacity of the output link. This results in packets being longer delayed than calculated. Therefore, it can be a good idea to discard the packets that are not able to be sent. The Delayer can either execute the feature or not. This optional feature is included since this is an improvement that may seem to be accurate. In section 5.3 "Packet Delay" the Delayer is evaluated with and without the feature.

The Delayer leverages the FFI and Rust for its implementation. Rust code is compiled into C binary code. This code can be called through the FFI. Rust is used to implement the buffer with a `BinaryHeap`. A `BinaryHeap` is used instead of a custom data structure since no proposed structure and sorting were as good as the `BinaryHeap`. The `BinaryHeap` buffers the tuples and sorts them according to the *time to send* ascending. Therefore, Rust is used for a simple and performant implementation of the buffer and uniform distribution.

4.5.3. Packet Ordering Function

This subsection introduces the *POF extern*. It implements packet ordering while supporting possible packet loss. In the following, the concept of this extern is described based on an activity diagram.

The *POF extern* is an extern implemented in Snabb. Its implementation is based on the algorithm in the informational RFC9950 [42]. It consists of the *last sequence number sent*, two *buffers*, and a *maximal delay*. The last sequence number sent is used to calculate the next expected sequence number. If a packet has the expected sequence number, it is directly forwarded. Otherwise, it needs to be buffered. The two buffers are one *sequence number buffer* and one *timer buffer*. A sequence number buffer is used to buffer packets and their sequence numbers. Packets are buffered until their sequence number is the expected one. Then, the appropriate packet is sent. If packet loss accrues, packets are buffered indefinitely. Therefore, timers are needed to send packets if packet loss is present. A *timer buffer* is used to store the timers. The *maximal delay* defines how long a packet needs to wait until packet loss is assumed. If packet loss is detected, the packet whose timer has been exceeded and buffered packets with smaller sequence numbers need to be forwarded. The POF extern implements packet ordering while supporting possible packet loss.

Figure 4.10 depicts the concept of the POF extern as an activity diagram. The color scheme and shapes are the same ones in the Delayer activity diagram depicted in Figure 4.9. The branching behavior is depicted in orange. Executed behavior is depicted in blue. The *Start* position is the start at which the engine starts to push packets through the app. When reaching the *End*, the engine pushed the packets as far as possible and now continues pushing packets through other apps. The activity diagram depicts two major parts. The first one is the *packet ordering* through buffering and transmitting received packets. The second one is the *packet loss detection* through timers and the transmission of buffered packets. Those two parts are further described in the following paragraphs.

The *packet ordering* is depicted on the left of Figure 4.10. Starting at the *start*, it is checked if packets can be received. If not, the second part is executed. Otherwise, the packet ordering logic is executed. First, the next packet is fetched. Afterward,

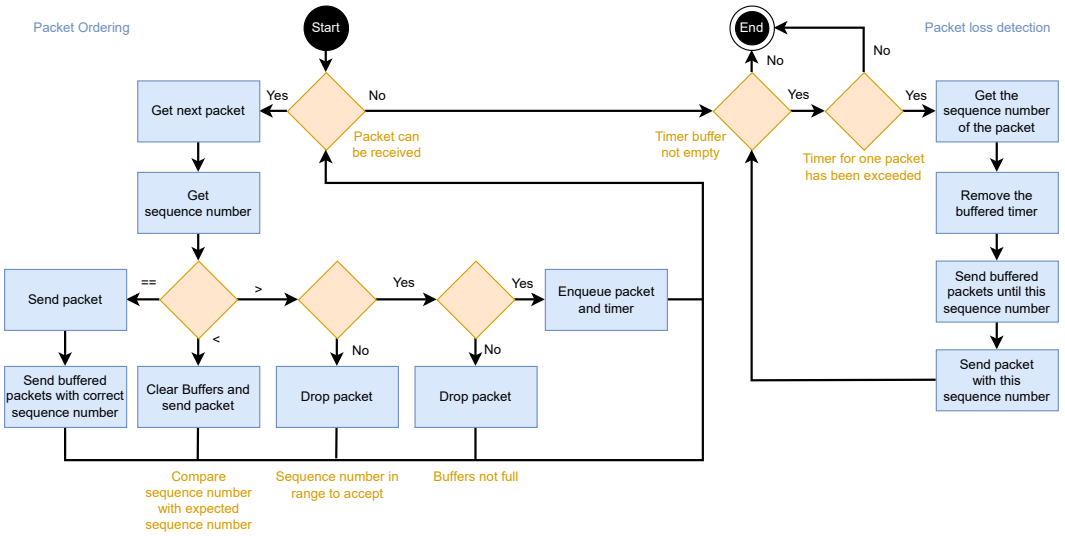

Figure 4.10.: Activity diagram of the concept of the POF extern.

the sequence number is retrieved. In this implementation, an ordering header is introduced between the IP and UDP headers. This ordering header includes a sequence number. After retrieving the sequence number, it is compared with the expected sequence number. If the retrieved sequence number is equal to the expected one, the packet is forwarded immediately through the output port. Afterward, all buffered packets with consecutive sequence numbers are sent until one sequence number is missing. If the retrieved sequence number is less than the expected one, sequence number wrapping is detected. Another case where the retrieved sequence number is less than the expected one is the following: A packet with a higher sequence number than expected was received and exceeded the maximal delay. It is therefore sent, and the next expected sequence number is adjusted accordingly. Afterward, a packet with a lower sequence number arrives. This is an illegal behavior defined in RFC9950 [42]. It is stated that this will cause out-of-order packets. This implementation adheres to the same restriction. Therefore, the use of the retrieved sequence number being less than the expected one as a signal for sequence number wrapping is valid. When sequence number wrapping appears, all buffers are cleared, and the packet is sent. Lastly, if the retrieved sequence number is larger than the expected one, the packet needs to be buffered. First, it needs to be checked if the retrieved sequence number is in the expected range. This is needed if sequence number wrapping is experienced and delayed packets are received afterward. Without the check for the expected range, these packets would be sent out-of-order. Therefore, packets outside of the expected range are dropped. Packets that are not able to be buffered because the buffers are full are dropped as well. If all these checks are passed, the packet and a timer are enqueued in the buffers, respectively. This part is executed while packets can be received. Received packets are either immediately sent, buffered, or dropped.

The *packet loss detection* is depicted on the right of Figure 4.10. The following steps are executed until either the timer buffer is empty or no timer has been

exceeded. As the first step, the sequence number of the first timer that has been exceeded is retrieved. Then, the timer is removed from the buffer. Afterward, all packets buffered with a smaller sequence number than the retrieved one are sent. At this point, all packets with a lower sequence number than the retrieved one are sent in order. Then, the packet with the retrieved sequence number is sent. If further timers have been exceeded, the same steps are applied. If a timer has been exceeded and the packet was previously sent by these steps, the timer is removed and no further packets are sent. After this part, all packets buffered longer or equal to the *maximal delay*, and their sequentially (buffered) predecessors were forwarded.

The *POF extern* leverages the FFI and Rust as well, like the *Delayer extern* does. Rust is used to implement both buffers. They are implemented as `BinaryHeap`s. The *sequence number buffer* contains the sequence number and the corresponding packet. The *timer buffer* contains the *time to send* (calculated based on the receive time and maximal delay) and the corresponding sequence number. Both buffers are sorted according to the sequence number and time to send them, respectively. Both buffers and the *last sequence number sent* are fields in a Rust struct. Rust provides the methods to access their fields and behavior accordingly. Lua code is used to implement the logic and call the Rust code.

5. Evaluation

This chapter evaluates the proposed architecture and externs. First, the experiment modalities, like the experiment setup and the methodology used, are described. Afterward, three main categories of experiments are conducted. The first experiments serve to create a baseline for using Snabb. Then, the Delayer extern is evaluated and compared to a reference tool. Lastly, experiments measuring the POF extern are conducted.

5.1. Experiment Modalities

In this section, the modalities of the conducted experiments are presented. First, the general setup of all experiments is presented. The setup is similar to the proposed architecture. Additionally, all systems used and their configurations are described. Afterward, the general execution of each experiment is presented. Then, the calculation of the final metrics is presented. To present the experiment results, three diagrams are included for each experiment. The scheme of these three diagrams is described in this section.

5.1.1. Setup

The setup for the experiments is presented in the following to ensure the reproducibility of each experiment. Figure 5.1 depicts the setup containing all systems and their wiring. All systems are wired with fiber-optic cables capable of 100 Gbps. In total, a maximum of three systems are used. Those three consist of two switches and one bare-metal server. Both switches are an *Edgecore Wedge 100BF-32X*. This is a P4 programmable switch containing an Intel® Tofino™ switching-ASIC. The leftmost switch (switch 1) is only used for running the latest version of P4TG. P4TG is used to generate UDP traffic and to do the measurements. This switch is connected to the switch in the middle (switch 2). Switch 2 can either run the P4 implementation of the concept or a P4TG instance. The P4 implementation is run to verify the concept. For better measurements, a P4TG instance can be run to directly send traffic to the Snabb program. Therefore, the plain Snabb throughput and delay are measured without any system in between. This switch 2 is connected to a bare-metal server. The bare-metal server is running *Ubuntu 22.04.4* on an *Intel® Xeon® Gold 6134 @ 3.20GHz* eight-core CPU. From these eight cores, cores 1-4 are isolated for only running the externs implemented by Snabb. Furthermore, the server has two *Mellanox ConnectX-5 100GbE Dual-Port* NICs and a single Non-Uniform Memory Access (NUMA) node. On the server, hyperthreads are disabled, and 64 huge pages each of size 1 GB are configured. Further, the Snabb version *Snabb 2024.06 "Faye"*

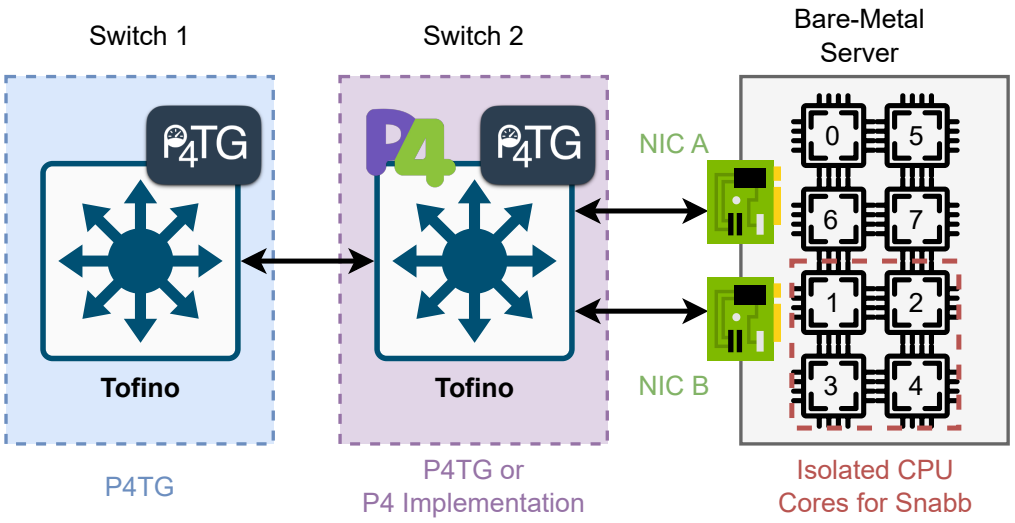

Figure 5.1.: Scheme of the experiment setup.

is used. These configurations are used according to the *performance tuning* in the Snabb repository [10].

5.1.2. Methodology

The general execution of an experiment is described in the following to ensure the reproducibility of each experiment. Each experiment is conducted six times. At the start of each experiment, the Snabb program and the P4TG are configured and started. After both systems have started, a time of 30 seconds is waited. In these 30 seconds, the Snabb program can initialize all resources needed since some are only initialized if traffic is received. After the transient phase of the Snabb program is over, the measurement starts. Therefore, the P4TG is restarted so that the transient phase measurements are cleared. Then the measurement takes 125 seconds. These 125 seconds are split into a 3-second transient phase of the P4TG after each restart, a 120-second measurement, and a 2-second buffer to ensure enough data points are collected. In the end, the Snabb program and the P4TG are both stopped. With this approach, six independent measurements are conducted. To evaluate these six measurements, first, a mean of the relevant metric is generated. This results in six independent means. Afterward, the mean and standard deviation of these six means are calculated. With these values, not only the final metric but also a confidence interval were determined. This execution of one experiment is then done for each relevant combination of transmission rate and frame size.

All experiments are conducted with multiple frame sizes, transmission rates, and flow numbers. The frame sizes are: 64, 256, 512, 1024, and 1518 bytes. 64-byte frames are the smallest, and 1518-byte frames are the largest frame sizes possible for an Ethernet frame without VLAN headers. For a better understanding of the influence of the frame size, intermediate frame sizes are needed. As the transmission rates, 10 Gbps, 20 Gbps, and 30 Gbps are used per flow. These rates are considered

to fit the traffic rates of real-world flows. Typically, not a single flow but multiple flows are processed at a time. Therefore, the experiments are conducted with a single flow and two flows. The single-flow experiments show the *raw* properties of each extern without multi-processing, like throughput, delay, and jitter. The multi-flow experiments depict the scalability of the externs through multi-processing. When multiple flows are used, they are engineered to be equally distributed among all workers. Further, as many workers as flows are used. These frame sizes, transmission rates, and numbers of flows are considered representative.

5.1.3. Diagram Scheme

In the following, each experiment includes three diagrams. These three are structured the same for each experiment. Further, each of the three diagrams has a similar scheme but differs in the data presented. Therefore, their scheme is described in this subsection. Each experiment description then focuses on the data present in the diagrams and not their scheme.

An example of the general structure of all diagrams is depicted in Figure A.1. This and all other diagrams are included in the appendix. On the x-axis, the different configured transmission rates and the flow number are depicted. The number of flows is indicated by the *1x* in front of the transmission rate. If two flows are configured, there is a *2x* in front of the transmission rate. Therefore, each transmission rate depicts the rate of a single flow. The presented rates are the ones defined in subsection 5.1.2: 10 Gbps, 20 Gbps, and 30 Gbps. Further, subsection 5.1.2 defines five frame sizes to be used: 64, 256, 512, 1024, and 1518 bytes. They are depicted in different colors. The mapping from the color to the frame size is presented in the legend. Further, the frame sizes are depicted from left to right in ascending order for each transmission rate. On the y-axis, the scale of the metric to be depicted is presented. The metric is either the receive rate/throughput given in Million Packets per Second (Mpps) or a time given in a multiple of seconds. A bar is used to indicate the value of the metric for each frame size and transmission rate combination. The value of each bar is the mean of the metric, as described in subsection 5.1.2. The value is further depicted as a text with either a black or white-colored font. Further, the confidence interval for each value is presented as an orange interval. The significance level α is depicted in the legend. Each diagram is subject to this scheme.

All diagrams with a green color scheme depict the Layer 1 receive rate/throughput. The units of the rates are Mpps. Mpps is used instead of Gbps. Mpps defines how many packets per second are transmitted or received, independent of the packet size. If the transmission rate is greater than the receiving rate, packet loss is present. In contrast, Gbps is dependent on the packet size. If the transmission rate is greater than the receive rate, either packet loss or different frame sizes are present. Without further description, the rate difference cannot be interpreted correctly. In some of the experiments, the frame size changes, e.g., a send VLAN header is stripped down. Mpps is used for an unambiguous interpretation of the rate difference. If a rate difference is present, this is due to the presence of packet loss and nothing else. The diagrams further include reference values. These reference values are depicted

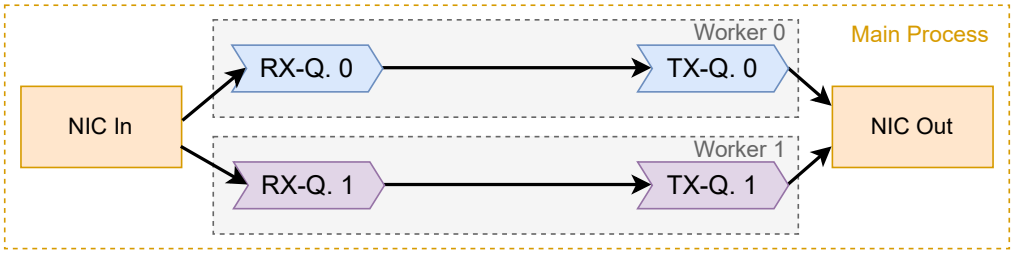

Figure 5.2.: Snabb network of the Baseline experiment.

as a horizontal red line with red-colored text. Their meaning is presented in the legend. It can be, e.g., the *mean transmission rate* or the *mean Snabb receive rate*. All diagrams with a blue color scheme depict the measured delay (round-trip time). All diagrams with a red color scheme depict the measured jitter. Both jitter and delay are measured in multiples of seconds. The unit is either ms or μs, depending on the order of magnitude.

5.2. Snabb Baseline

In the following, the baselines of Snabb are evaluated. The baselines are for throughput, delay, and jitter. To be more precise, the throughput is the rate at which Snabb processes packets. It is measured as the receiving rate at the P4TG. The following experiments will evaluate the maximal throughput, the base delay, and the base jitter. Figure 5.2 depicts the Snabb network used in the experiments. One or multiple NICs are initialized by the main process. Depending on the experiment, one or multiple flows and, therefore, one or multiple workers are used. Each worker then initializes the *IO* apps for the receive and transmission queues. If only one NIC is used, the same IO app is used for receiving and transmission. If multiple NICs are used, one NIC is used as the receiving and one as the transmission NIC. None of them is used for receiving as well as transmission. In that case, two different IO apps are used. No additional app is in between the receiving queue and the transmission queue. Both queues are directly connected.

5.2.1. Single NIC

In this experiment, the same *ConnectX* app and therefore the same NIC is used for the *NIC In* and *NIC Out*.

Single Flow

The throughput of this experiment is depicted in Figure A.1. As a reference, the *Mean TX Rate* is given. It is shown that at a transmission rate of 20 Gbps, the 64-byte frame size cannot be forwarded at the line rate. Just over 50% of the transmission rate is reached. At 30 Gbps, the 64 and 256-byte frames both cannot be forwarded at the line rate. The 64-byte frames reached a rate of around 35% and

the 256-byte frames around 93% of the transmission rate. Every other frame size can be forwarded at the line rate at every transmission rate. The mean throughputs all include a really small confidence interval. Therefore, the measured rates are representative.

Figure A.2 depicts the delay. The delay is largest for 64-byte frames, and most of the time gets lower with a larger frame size. Further, the delay gets higher with higher transmission rates. The biggest rise is at 256-byte frame size, from 20 Gbps to 30 Gbps. The delay increased by more than 23 times. All other frame size and transmission rate combinations have a delay of roughly $4 - 6\mu s$. As for the throughputs, really small confidence intervals are present. Therefore, the measured delays are representative.

The last diagram in Figure A.3 depicts the jitter. Most of the time, the jitter is in between $0.2 - 0.5\mu s$ and has a very small confidence interval. Exceptions are the frame size and transmission rate combinations, which were already exceptions in the delay. 64-byte frames have a jitter of roughly $4.2\mu s$ and a confidence interval of $+/- 1\mu s$. The combinations of 64-byte frames at a 20 Gbps transmission rate and 64- and 256-byte frames at a 30 Gbps transmission rate have a jitter of roughly $1 - 2\mu s$. Therefore, the jitter of most of the combinations is really small.

Multiple Flows

In the following, the differences compared to the single-flow experiment are described. Figure A.4 depicts the throughput. The same combinations as in the single flow experiment could not be forwarded at the line rate. Additionally, the combination of 64-byte frames at a 10 Gbps transmission rate could not be forwarded at the line rate either. These combinations further show that the usage of workers does not scale the throughput accordingly. The throughput of 256-byte frames at 30 Gbps is 20.243 Mpps. In the single flow experiment, it was 12.4 Mpps. This is an increase by a factor of roughly 1.6, which is less than the factor 2 by which the number of workers was increased. The same holds for the other combinations, which could not be forwarded at the line rate. Still, the confidence intervals are small, and therefore the measurements are representative.

Figure A.5 and Figure A.6 depict the delay and jitter, respectively. Both have spikes at the same frame size and transmission rate combinations like in the single flow experiment. These spikes are greater compared to the single-flow experiment. Further, the overall delay increased from roughly $4 - 6\mu s$ to roughly $4 - 8\mu s$. The same holds for the overall jitter. It increases from $0.2 - 0.5\mu s$ to $0.3 - 0.65\mu s$. Additionally, the jitter spikes increased slightly. Further, the confidence intervals of the jitter got larger while the ones for the delay stayed small. Therefore, the measured delay is representative, but the measured jitter shows significant fluctuations. This experiment showed that while multiple workers increase the throughput, the delay and jitter get slightly worse.

5.2.2. Multiple NICs

In this experiment, two different *ConnectX* apps and therefore two different NICs are used for the *NIC In* and *NIC Out*.

Single Flow

Figure A.7 depicts the throughputs measured in this experiment. Compared with the single NIC experiment, the same frame size and transmission rate combinations are forwarded at the line rate. Additionally, the combination of 256-byte frames at a 30 Gbps transmission rate is forwarded at the line rate as well. Further, the throughput of 64-byte frames is larger with two different NICs than with a single one. This shows that the NIC was a bottleneck in the single NIC setup. The now-measured throughputs are the ones where only Snabb is the bottleneck. Therefore, this experiment shows the real Snabb baseline without further bottlenecks.

Figure A.8 and Figure A.9 depict the delay and jitter, respectively. The overall delay stayed the same compared to the single NIC experiment. The combination of 64-byte frames at a 10 Gbps transmission rate now has only half the delay as before. Further, the delay of 256-byte frames at 30 Gbps decreased by a factor of roughly 14. Still, most of the delays cannot be improved by using multiple NICs. In contrast, the jitter got a little bit worse. Especially the confidence intervals became greater. Therefore, the measured jitter shows greater fluctuations when using multiple NICs than when using a single NIC.

Multiple Flows

The measured throughputs are depicted in Figure A.10. In contrast to the single NIC multiple flow experiment, the 64-byte frames at a 10 Gbps transmission rate are forwarded at the line rate. Now, only the combinations of 64-byte frames at a 20 Gbps or 30 Gbps transmission rate cannot be forwarded at the line rate. A further enhancement compared to the single NIC multi-flow experiment is the scaling of the throughput. Both combinations not forwardable at the line rate increased by a factor of 1.95, while the number of workers increased by a factor of 2. This is an increase in the factor of roughly 0.14 compared to the single NIC factor. Still, the increase in throughput is not as high as the increase in the number of workers.

Figure A.11 and Figure A.12 depict the delay and jitter, respectively. Compared to the single-flow experiment, the delays increased slightly. This is the same behavior as for the single NIC multi-flow experiment. The same is true for the jitter. It increased compared to the single-flow experiment. In contrast to the single NIC multi-flow experiment, the jitter and the confidence intervals are slightly smaller. Therefore, the usage of multiple NICs increases the throughput while slightly reducing the delay and jitter.

5.3. Packet Delay

In the following, different packet delay implementations are evaluated. An experiment of two different delay and jitter combinations is conducted for each implementation. A combination of 5 ms delay and 0.1 ms jitter and then a combination of 50 ms delay and 1 ms jitter. First, the Linux NetEm is evaluated. NetEm is used to compare the custom Snabb implementation to a well-known tool. Afterward, the Snabb Delayer described in subsection 4.5.2 is evaluated. This extern is evalu-

```
sudo tc qdisc add
    dev <iface> root netem
    limit <limit>
    delay <delay>ms <jitter>ms
```

Listing 5.1: Command to configure NetEm.

ated in a single-flow and multi-flow experiment to show its scalability. Further, the experiments are conducted with the *TimeDrop* feature enabled and disabled.

5.3.1. NetEm Reference

In this subsection, NetEm is evaluated as a reference for a well-known tool implementing packet delaying. It is a queueing discipline implemented in the Linux kernel. NetEm is configurable with a delay, jitter, and limit [13]. For the delay and jitter, the same values are used as for the later evaluated Snabb implementation. The two combinations of 5 ms delay with 0.1 ms jitter and 50 ms delay with 1 ms jitter are used. As the limit, the same limit as for the Snabb Delayer is used. The limit is 262144 packets. NetEm is configurable with the command presented in Listing 5.1. In the following experiments, transmission rates of 1, 2, and 3 Gbps are used in contrast to the experiments with Snabb. They are used because NetEm cannot forward UDP traffic (at the line rate) at higher transmission rates.

Delay 5ms and Jitter 0.1ms In this experiment, the delay is 5 ms with a 0.1 ms jitter. Figure A.13 depicts the throughput of NetEm in this configuration. All measured throughputs have small confidence intervals. Therefore, the measured throughputs are representative. Out of all frame size and transmission rate combinations, only five are forwardable at the line rate: 512-byte frames at a 1 Gbps transmission rate and 1024- and 1518-byte frames at a 1 and 2 Gbps transmission rate. Further, four combinations are forwardable at nearly line rate: 256-byte frames at a 1 Gbps transmission rate, 512-byte frames at a 2 Gbps transmission rate, and 1024- and 1518-byte frames at a 3 Gbps transmission rate. Therefore, 9 out of 15 combinations are forwardable at (nearly) the line rate.

Figure A.14 and Figure A.15 depict the delay and jitter, respectively. Most of the measured delays have a small confidence interval. In contrast, only half of the measured jitter has a small confidence interval. Therefore, the delays are representative, while the measured jitter shows significant fluctuations. Based on the measurements, no correlation between whether a combination is forwardable at (nearly) line rate and whether the configured delay and jitter are met can be deduced. No combination met both the configured delay and the configured jitter. Out of the combinations forwardable at (nearly) line rate, the maximal absolute error of the delay is $5.584\,\text{ms} - 5\,\text{ms} = 0.584\,\text{ms}$ for 1518-byte frames at a 2 Gbps transmission rate. Therefore, the maximal relative error of the delay is $\frac{0.584\,\text{ms}}{5\,\text{ms}} = 11.68\,\%$. The maximal absolute error of the jitter is $0.315\,\text{ms} - 0.1\,\text{ms} = 0.215\,\text{ms}$ for 1518-byte

frames at a 3 Gbps transmission rate. Therefore, the maximal relative error of the jitter is $\frac{0.315\,\text{ms}}{0.1\,\text{ms}} = 215\,\%$. These maximal errors show that NetEm is not accurate at this configured delay and jitter.

Delay 50ms and Jitter 1ms Figure A.16 depicts the throughput of NetEm in this configuration. In this experiment, the same combinations of frame size and transmission rate are forwardable at (nearly) the line rate, as in the experiment with 5 ms delay and 0.1 ms jitter. Some throughputs are slightly smaller than before. This is also true for the maximal throughput. Overall, the increase in the configured delay and jitter did not change the throughputs significantly.

Figure A.17 and Figure A.18 depict the delay and jitter, respectively. Like before, no correlation between whether a combination is forwardable at (nearly) the line rate and whether the configured delay and jitter are met can be deduced. In contrast to before, the jitter is now not higher but lower than the configured one. Further, most of the confidence intervals of the measured delays are quite large. Therefore, the measured jitter shows significant fluctuations. Out of the combinations forwardable at (nearly) line rate, the maximal absolute error of the delay is $51.392\,\text{ms} - 50\,\text{ms} = 1.392\,\text{ms}$ for 512-byte frames at a 2 Gbps transmission rate. Therefore, the maximal relative error of the delay is $\frac{1.392\,\text{ms}}{50\,\text{ms}} = 2.784\,\%$. The maximal absolute error of the jitter is $1\,\text{ms} - 0.598\,\text{ms} = 0.402\,\text{ms}$ for 512-byte frames at a 1 Gbps transmission rate. Therefore, the maximal relative error of the jitter is $\frac{0.402\,\text{ms}}{1\,\text{ms}} = 40.2\,\%$. The maximal absolute errors did not significantly change compared to the lower-configured delay and jitter. In contrast, the relative errors got significantly smaller. The maximal relative error of the delay is only 23.8 % of the previous value. The maximal relative error of the jitter was even reduced to 18.7 % of the previous value. This reduction can be explained by roughly the same absolute errors, which are relatively smaller compared to higher values.

5.3.2. Snabb Delayer – TimeDrop Disabled

Figure 5.3 depicts the Snabb network used in the experiments. This network is similar to the one used in section 5.2 “Snabb Baseline”. The only difference is that the receiving queue and the transmission queue are not directly connected. The *Delayer* app is placed in between. Therefore, received packets are delayed and not sent immediately. In these experiments, the *TimeDrop* feature is disabled by commenting out the code in the Delayer app. In the following, the described throughput diagrams have a different reference. Not the transmission rate, but the throughput of subsection 5.2.1 “Single NIC” is used as a reference. This baseline is used because the following experiments are conducted with a single NIC. Therefore, it can be compared to how much the Delayer throughput differs from the baseline.

Single Flow

Delay 5ms and Jitter 0.1ms Figure A.19 depicts the throughput of the Delayer in this configuration. Only the following combinations of frame size and transmission rate are forwardable at the line rate: 512-byte frames at 10 Gbps and 1024- and 1518-

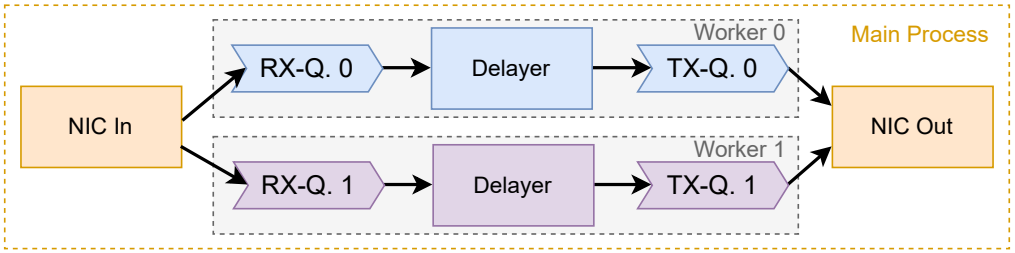

Figure 5.3.: Snabb network of the Delay experiment.

byte frames at 10, 20, and 30 Gbps. Every other combination cannot be forwarded at the line rate. The maximum throughput is between 3.5 and 4.1Mpps. Therefore, the throughput stays roughly the same at higher transmission rates. This behavior is the same as in the baseline when a combination cannot be forwarded at the line rate.

Figure A.20 and Figure A.21 depict the delay and jitter, respectively. The delay was configured to be 5 ms, while the jitter was configured to be 0.1 ms. Both diagrams show that the delay and jitter are only the configured ones if the combination is forwardable at the line rate. Out of these combinations, the maximal absolute error of the delay is $5.029\,\text{ms} - 5\,\text{ms} = 0.029\,\text{ms}$ for 1024-byte frames at a 30 Gbps transmission rate. Therefore, the maximal relative error of the delay is $\frac{0.029\,\text{ms}}{5\,\text{ms}} = 0.58\,\%$. The maximal absolute error of the jitter is $0.104\,\text{ms} - 0.1\,\text{ms} = 0.004\,\text{ms}$ for 1518-byte frames at a 10 Gbps transmission rate. Therefore, the maximal relative error of the jitter is $\frac{0.004\,\text{ms}}{0.1\,\text{ms}} = 4\,\%$. Both the delay and jitter are quite accurate for the combinations forwardable at the line rate.

These evaluations are now compared with the ones from NetEm. First, the Snabb Delayer has a correlation between the forwardability at the line rate and whether the configured delay and jitter are met. If a frame size and transmission rate combination is forwardable at the line rate, it meets the configured delay and jitter. The Snabb Delayer forwards packets at ten times higher transmission rates. While doing so, the maximal relative error of the delay is 5 % of the maximal relative error of NetEm. Further, the maximal relative error of the jitter is 1.8 % of the maximal relative error of NetEm. The Snabb Delayer reaches much higher throughputs while being more accurate than NetEm.

Delay 50ms and Jitter 1ms Figure A.22 depicts the throughput of the Delayer in this configuration. In this experiment, the same combinations of frame size and transmission rate are forwardable at the line rate, as in the experiment with 5 ms delay and 0.1 ms jitter. One exception is the combination of 1024-byte frames at a 30 Gbps transmission rate. This combination is not forwardable at the line rate. Therefore, only 6 out of 15 combinations are forwardable at the line rate. The maximum throughput is around 2.6 Mpps. This is consistent with the expected behavior. Packets are buffered longer while the buffer size stays the same. Therefore, more packets need to be dropped since the buffer is already full. Because of this, the maximum throughput decreases if the configured delay and jitter are increased.

The delay and jitter are depicted in Figure A.23 and Figure A.24, respectively. Like in the experiment before, both configured values are only met in the combinations forwardable at the line rate. The maximal absolute error of the delay is $50.044\,\text{ms} - 50\,\text{ms} = 0.044\,\text{ms}$ for 1024-byte frames at a 20 Gbps transmission rate. Therefore, the maximal relative error of the delay is $\frac{0.044\,\text{ms}}{50\,\text{ms}} = 0.088\,\%$. The maximal absolute error of the jitter is $1.024\,\text{ms} - 1\,\text{ms} = 0.024\,\text{ms}$ for 1518-byte frames at a 20 Gbps transmission rate. Therefore, the maximal relative error of the jitter is $\frac{0.024\,\text{ms}}{1\,\text{ms}} = 2.4\,\%$. In contrast to the smaller configured delay and jitter, the absolute errors are roughly the same or a little bit higher, while the relative errors got smaller. That behavior of the absolute errors can be explained by fluctuations in the processing time of Snabb, which are relatively constant. These constant fluctuations in the processing time have a smaller effect relative to larger values. This shows that the delay and jitter get more accurate (smaller relative error) with a higher configured delay and jitter, while the absolute error stays roughly the same.

These evaluations are now compared with the ones from NetEm. As previously, the correlation between the forwardability at the line rate and the compliance of the configured delay and jitter is still present while not being present for NetEm. The Snabb Delayer reduced the maximal relative error of the delay to 3.16 % of the maximal relative error of NetEm. Further, the maximal relative error of the jitter is 5.97 % of the maximal relative error of NetEm. These reductions are smaller than the ones for a smaller configured delay and jitter. Still, the Snabb Delayer reaches much higher throughputs while being more accurate than NetEm.

Multiple Flows

Delay 5ms and Jitter 0.1ms Figure A.25 depicts the throughput of the Delayer in this configuration. Compared with the single flow experiment, the same frame size and transmission rate combinations are forwardable at the line rate. Like in the baseline when changing from single-flow to multi-flow, the throughputs do not scale with the same factor as the workers are scaled. The throughput of 256-byte frames at a 10 Gbps transmission rate increased from 3.57 Mpps to 7.023 Mpps. This is an increase by a factor of 1.96. Therefore, the throughput nearly scales with the factor of the number of workers.

Figure A.26 and Figure A.27 depict the delay and jitter, respectively. Like previously, only the frame size and transmission rate combinations forwardable at the line rate met the configured delay and jitter. The maximal absolute error of the delay is $5.037\,\text{ms} - 5\,\text{ms} = 0.037\,\text{ms}$ for 1024-byte frames at a 30 Gbps transmission rate. Therefore, the maximal relative error of the delay is $\frac{0.037\,\text{ms}}{5\,\text{ms}} = 0.74\,\%$. The maximal absolute error of the jitter is $0.1\,\text{ms} - 0.098\,\text{ms} = 0.002\,\text{ms}$ for 1518-byte frames at a 20 Gbps transmission rate. Therefore, the maximal relative error of the jitter is $\frac{0.002\,\text{ms}}{0.1\,\text{ms}} = 2\,\%$. Compared to the single-flow experiment, the maximal error of the delay got slightly worse. In contrast, the maximal error of the jitter is only half as large as before. The use of multiple flows increases the throughput while not significantly changing the accuracy of the configured delay or jitter.

Delay 50ms and Jitter 1ms Figure A.28 depicts the throughput of the Delayer in this configuration. The same frame size and transmission rate combinations as in the single flow experiment could not be forwarded at the line rate. Additionally, the combination of 1518-byte frames at a 30 Gbps transmission rate could not be forwarded at the line rate either. Therefore, only 5 out of 15 combinations are forwardable at the line rate. Compared to the experiment with a lower configured delay and jitter, two fewer combinations can be forwarded at the line rate. No frame size is forwardable at the line rate at a 30 Gbps transmission rate when using multiple flows.

The delay and jitter are depicted in Figure A.29 and Figure A.30, respectively. In contrast to the single flow experiment, not only the frame size and transmission rate combinations forwardable at the line rate meet the configured delay and jitter. Also, the 1518-byte frames at a 30 Gbps transmission rate met the configured delay and jitter. The maximal absolute error of the delay is $50.099\,\text{ms} - 50\,\text{ms} = 0.099\,\text{ms}$ for 1518-byte frames at a 30 Gbps transmission rate. Therefore, the maximal relative error of the delay is $\frac{0.099\,\text{ms}}{50\,\text{ms}} = 0.198\,\%$. The maximal absolute error of the jitter is $1.033\,\text{ms} - 1\,\text{ms} = 0.033\,\text{ms}$ for 1518-byte frames at a 30 Gbps transmission rate. Therefore, the maximal relative error of the jitter is $\frac{0.033\,\text{ms}}{1\,\text{ms}} = 3.3\,\%$. Compared to the experiment with a lower configured delay and jitter, the maximal error of the delay got better, and the maximal error of the jitter got slightly worse. Compared to the single-flow experiment, both the maximal error of the delay and jitter got slightly worse. Therefore, the use of multiple flows increases the throughput as well if a higher delay and jitter are configured, but the maximal errors of the delay and the jitter get slightly worse.

5.3.3. Snabb Delayer – TimeDrop Enabled

The Snabb network stays the same as for the experiments, with the *TimeDrop* feature disabled. It is depicted in Figure 5.3. The difference in this experiment is the *Delay* extern in the network. In its code, the part implementing the *TimeDrop* feature is commented in. Therefore, this feature is enabled.

Delay 5ms and Jitter 0.1ms Figure A.31 depicts the throughput of the Delayer in this configuration. The frame size and transmission rate combinations forwardable at the line rate do not differ in this experiment compared to the experiment with the *TimeDrop* feature disabled. The other combinations differ a lot. Most of these have much worse throughput. The throughput even gets worse with higher transmission rates. This is consistent with the expected behavior of the TimeDrop feature. It drops packets that cannot be forwarded in time. Therefore, some buffered packets get dropped. With higher transmission rates, more packets get buffered at around the same time. This leads to many packets being dropped because the Snabb link is full. Therefore, the buffer is alternating between being full and being empty. The buffer is filled until it is full. After the delay time, nearly all packets need to be sent. Most of the packets cannot be sent and are dropped. Therefore, the buffer went from full to empty. This explains the significant decrease in throughput. The use of the *TimeDrop* feature makes the throughput much worse.

The *TimeDrop* feature was proposed to improve the delay and jitter. Figure A.32 and Figure A.33 depict the delay and jitter, respectively. Like in the experiment with the feature disabled, only the combinations forwardable at the line rate meet the configured delay and jitter. The maximal absolute error of the delay is 5.013 ms − 5 ms = 0.013 ms for 1024-byte frames at a 20 Gbps transmission rate and 10518-byte frames at a 10 and 30 Gbps transmission rate. Therefore, the maximal relative error of the delay is $\frac{0.013\,\text{ms}}{5\,\text{ms}} = 0.26\,\%$. The maximal absolute error of the jitter is 0.102 ms − 0.1 ms = 0.002 ms for 1024-byte frames at a 10 Gbps transmission rate and 1518-byte frames at a 20 Gbps transmission rate. Therefore, the maximal relative error of the jitter is $\frac{0.002\,\text{ms}}{0.1\,\text{ms}} = 2\,\%$. Both absolute and relative errors are only half the error compared to not using the TimeDrop feature. The usage of the *TimeDrop* feature made the delay and the jitter more accurate while much worsening the throughput of frame size and transmission rate combinations not being able to be forwarded at the line rate.

Delay 50ms and Jitter 1ms Figure A.34 depicts the throughput of the Delayer in this configuration. Compared to the experiment with the same delay and jitter and *TimeDrop* disabled, the same frame size and transmission rate combinations are forwardable at the line rate. The throughput of all the other combinations got worse. This is the same behavior as when increasing the delay and jitter without using the feature. Still, the throughputs are much worse than without this feature. Therefore, the relevant throughputs did not change while all others got worse.

Figure A.35 and Figure A.36 depict the delay and jitter, respectively. The delay and jitter are increased when enabling this feature for a small delay and jitter. Like before, the configured delay and jitter are only met by the frame size and transmission rate combinations forwardable at the line rate. The maximal absolute error of the delay is 50.081 ms − 50 ms = 0.081 ms for 1518-byte frames at a 10 Gbps transmission rate and 10518-byte frames at a 10 and 30 Gbps transmission rate. Therefore, the maximal relative error of the delay is $\frac{0.081\,\text{ms}}{50\,\text{ms}} = 0.162\,\%$. The maximal absolute error of the jitter is 1.021 ms − 1 ms = 0.021 ms for 1024-byte frames at a 10 Gbps transmission rate and 1518-byte frames at a 20 Gbps transmission rate. Therefore, the maximal relative error of the jitter is $\frac{0.021\,\text{ms}}{1\,\text{ms}} = 2.1\,\%$. Compared to the experiment with the feature disabled, the absolute and relative errors of the delay are nearly twice as big. The errors in the jitter got slightly smaller. Still, the maximal relative error of the delay is negligibly small. Therefore, the usage of the *TimeDrop* feature increases the accuracy of the delay and jitter for the relevant combinations while worsening the throughput, delay, and jitter for all others.

5.4. Packet Ordering

Figure 5.4 depicts the Snabb network used in the experiments. This network is similar to the one used in subsection 5.3.2 “Snabb Delayer – TimeDrop Disabled”. The difference is that only one worker, and therefore only one flow, is used. To execute this experiment, packet scrambling needs to happen. The P4TG is not able to generate packets with scrambled sequence numbers. Therefore, the P4-programmable

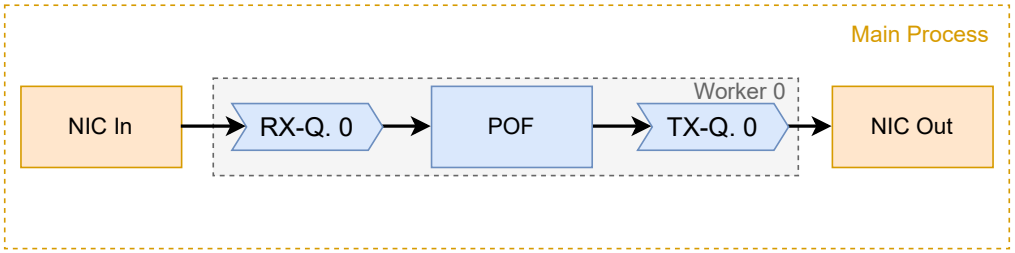

Figure 5.4.: Snabb network of the Packet Ordering experiment.

switch running the P4 implementation is extended. Packets entering from the P4TG have a probability of 1/4 to be recirculated. Packets are scrambled due to the probabilistic recirculation. Further, an ordering header is pushed with the sequence number the P4TG added. A maximal delay of 16 μs was measured for the probabilistic recirculation. For the experiment, the maximal delay was set to 50 μs to configure a sufficiently large delay. In the following, the described throughput diagram has a different reference than the Delayer. Not the throughput of subsection 5.2.1 "Single NIC", but the transmission rate, is used as a reference. The transmission rate is used because the P4TG had a slightly higher transmission rate than when executing the baseline experiment. Therefore, the POF throughput would seem to be higher than the baseline throughput. To circumvent this misunderstanding, the transmission rate is used as the reference. In the following paragraphs, the measurements of the experiment are described. The validity of the code was verified by the P4TG which did not record any out-of-order packets in the experiments.

Figure A.37 depicts the throughput of the POF extern. Out of the 15 frame size and transmission rate combinations, six are forwardable at (nearly) the line rate. Those are 512-byte frames at a 10 Gbps transmission rate, 1024-byte frames at a 10 and 20 Gbps transmission rate, and 1518-byte frames at a 10, 20, and 30 Gbps transmission rate. These are the same combinations forwardable at the line rate by the Delayer configured with a 50ms delay and a 1 ms jitter. These combinations are the relevant ones. The maximal throughput is around 3.7 Mpps. This is a similar maximal throughput as the single-flow Delayer configured with 5ms delay and 0.1 ms jitter experienced. In contrast to the other experiment, the throughputs of the combinations not forwardable at the line rate have greater confidence intervals. Therefore, these combinations show significant fluctuations in their throughput. This was also experienced when the experiment was conducted. During the execution, significant throughput drops were experienced. Since this is only true for the irrelevant combinations, these observations can be neglected. The POF has similar throughput capabilities as the Delayer.

Figure A.38 and Figure A.39 depict the delay and jitter, respectively. Both are the lowest for the relevant combinations. The irrelevant combinations all have a higher delay, a higher jitter, and a larger confidence interval. Therefore, a correlation between the forwardability at the line rate and the delay and jitter is present. The delay is between 57 μs and 63 μs for the relevant combinations. This is expectable since the configured *maximal delay* is 50 μs. Therefore, the experienced delay has to

be slightly bigger. The jitter is between 2 μs and 8 μs. In contrast to the delay, the jitter has slightly bigger confidence intervals. Therefore, the POF extern introduces a small and predictable delay and jitter.

6. Conclusion

This chapter concludes the thesis. First, a summary is given describing the proposal of this thesis. Afterwards, a discussion is held about the proposal. Finally, possible work for the future is presented.

6.1. Summary

The objective of this thesis was to overcome the shortcomings of P4-based switches not being able to implement more sophisticated features. To enable these features, a co-located server should implement externs as VNFs. This thesis proposes an architecture and signaling to enable the extension of P4-based switches. A P4-programmable switch and a co-located bare-metal server are used. The switch implements a configurable data plane. This data plane is configured by a control plane. The control plane configures flows to be processed by the externs implemented at the co-located server. Those flows can either be simple or VLAN-based Ethernet-IP flows. An additional VLAN header is used to signal which extern needs to be applied. This VLAN header is further used to distribute the load upon multiple instances implementing the same extern. A custom Snabb header is introduced to provide additional data, either set by the data plane or the control plane. Two externs are implemented as a prototype of this architecture. Those are a *Delayer extern* and a *POF extern*. The Delayer extern can delay packets according to a configured delay and jitter. The POF extern can order packets arriving out-of-order according to a sequence number present in a header. Finally, experiments were conducted to evaluate the imposed overhead and extern-specific metrics. The baseline shows that Snabb imposes an overhead and cannot forward each combination of frame size and transmission rate at the line rate. Further, different combinations impose different delays and jitters. The multi-flow experiments show that Snabb is scalable if multiple flows combined with multi-processing are used. Compared to NetEm, the Delayer extern forwards UDP traffic at a 10 times higher transmission rate while being more accurate in the configured delay and jitter. The POF extern orders packets and supports possible packet loss while having similar throughput capabilities as the Delayer and introducing a predictable delay and jitter.

6.2. Discussion

This section includes a discussion of this thesis. First, the positive points of what is done and the proposal are mentioned. Afterward, some drawbacks regarding the use of Snabb, the need to rerun some experiments, and some missing experiments are stated.

The evaluation and prototype show that the proposed architecture and signaling are working. Two externs were implemented successfully. Further, the load balancing is successfully applied in the multi-flow experiments. The proposed architecture and signaling are extensible enough to support new externs and be seamlessly integrated. This is due to the extensible concept of signaling through the VLAN header. Further, the configurable data plane enables the seamless integration of new externs at runtime. The experiments show that the implemented externs are not only correctly working but also working at rather high throughputs.

The conducted experiments further show that Snabb is not working well in overload situations. Overload is when the combination of frame size and transmission rate is not forwardable at the line rate. These combinations share similar throughputs, which is the maximal throughput. Further, they all have a higher delay and jitter than the combinations forwardable at the line rate. Additionally, they share greater confidence intervals for the delay and jitter. This shows that Snabb has some kind of *hiccups* when performing under overload. These hiccups result in throughput dips and higher round-trip times. This leads to a greater mean of the delay and the jitter. The measurements under overload are not deterministic. This led to the need to rerun multiple experiment combinations because they had significantly different measurements. A further drawback of this thesis is that it only includes *end-to-end* experiments and evaluations. Only a baseline and the experiments for the two externs were conducted. No more fine-grained experiments regarding, e.g., the overhead of using the FFI and Rust, the maximal overhead imposed by using the Snabb header, or the use of more workers than two were conducted.

6.3. Future Work

In the future, some aspects can be further elaborated on. First, more fine-grained experiments and evaluations can be done. This will enable a more sophisticated analysis of what features implemented will impose what overhead. For example, the use of the FFI and Rust needs to be measured. Rust enables the usage of structs and logic present in its standard library. These structs can be directly implemented in C. The logic can be implemented in Lua. It needs to be evaluated if the use of Rust imposes a significant overhead over the manual implementation of such logic. Further, the maximal scalability of Snabb needs to be evaluated. It is currently unknown if multiple workers will fully utilize the NIC. The experiments show that one single NIC imposes a bottleneck. This bottleneck could be circumvented by using multiple NICs. It could not be shown if more than two workers with an aggregated bandwidth of 100 Gbps will forward packets at the line rate when using multiple NICs. Furthermore, all experiments were conducted using the P4TG, which only supports UDP traffic. The kernel implementation for forwarding UDP traffic is worse than for TCP. Therefore, the experiments of NetEm need to be conducted and compared for TCP traffic as well. Additionally, more externs need to be implemented in the future. In this thesis, only two different externs were implemented. More use cases can be solved if more sophisticated externs are implemented. One example would be the implementation of a *cryptography extern* to enable IP- or MAC-sec.

List of Acronyms

API Application Programming Interface
ASIC Application-Specific Integrated Circuit
CPU Central Processing Unit
CRC Cyclic Redundancy Check
FFI Foreign Function Interface
FPGA Field Programmable Gate Array
FSM Finite State Machine
Gbps Gigabit per second
LSB Least Significant Bit
MAT Match-Action Table
MAU Match-Action Unit
MSB Most Significant Bit
NetEm Network Emulator
NFV Network Function Virtualization
NIC Network Interface Card
NPU Network Processing Unit
NUMA Non-Uniform Memory Access
Mpps Million Packets per Second
OS Operating System
P4 Programming Protocol-Independent Packet Processors
P4TG P4 Traffic Generator
PISA Protocol-Independent Switching Architecture
POF Packet Ordering Function
PSA Portable Switch Architecture
RDMA Remote Direct Memory Access
SDN Software-Defined Networking
TNA Tofino Native Architecture
VNF Virtual Network Function
VM Virtual Machine

Bibliography

[1] Intel®. Intel® Intelligent fabric processors. Oct. 2023. URL: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html (visited on 07/13/2024).

[2] Intel®. $P4_{16}$ Intel® Tofino™ Native Architecture – Public Version. Technical report, Apr. 2021. URL: https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf.

[3] G. Ara, L. Abeni, T. Cucinotta, and C. Vitucci. On the Use of Kernel Bypass Mechanisms for High-Performance Inter-container Communications. In M. Weiland, G. Juckeland, S. Alam, and H. Jagode, editors, *High Performance Computing*, pages 1–12, Cham. Springer International Publishing, 2019.

[4] G. Ara, T. Cucinotta, L. Abeni, and C. Vitucci. Comparative Evaluation of Kernel Bypass Mechanisms for High-performance Inter-container Communications. In *International Conference on Cloud Computing and Services Science - CLOSER*, pages 44–55, 2020.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[6] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng, J. Benitez, U. Michel, H. Damker, K. Ogaki, T. Matsuzaki, M. Fukui, K. Shimano, D. Delisle, Q. Loudier, C. Kolias, I. Guardini, E. Demaria, R. Minerva, A. Manzalini, D. López, F. J. R. Salguero, F. Ruhl, and P. Sen. Network Functions Virtualisation. An Introduction, Benefits, Enablers, Challenges & Call for Action. Technical report, Oct. 2012. URL: https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[7] ETSI. NFVwiki. NFV FAQ. Apr. 2022. URL: https://nfvwiki.etsi.org/index.php?title=NFV_FAQ (visited on 07/25/2024).

[8] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN: an intellectual History of programmable Networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, Apr. 2014.

[9] S. Gilbert and N. Lynch. Perspectives on the CAP Theorem. *Computer*, 45(2):30–36, 2012.

[10] L. Gorrie, M. Rottenkolber, A. Wingo, and D. Pino. Snabb. June 2024. URL: https://github.com/snabbco/snabb (visited on 07/26/2024).

[11] S. Hassas Yeganeh and Y. Ganjali. Kandoo: a Framework for efficient and scalable Offloading of control Applications. In *Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, Helsinki, Finland. Association for Computing Machinery, 2012.

[12] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and applied Research. *Journal of Network and Computer Applications*, 212:103561, 2023.

[13] S. Hemminger. tc-netem(8) — Linux manual page. Technical report, Nov. 2011. URL: `https://man7.org/linux/man-pages/man8/tc-netem.8.html` (visited on 08/14/2024).

[14] Y. Jarraya, T. Madi, and M. Debbabi. A Survey and a Layered Taxonomy of Software-Defined Networking. *IEEE Communications Surveys & Tutorials*, 16(4):1955–1980, 2014.

[15] S. Kianpisheh and T. Taleb. A Survey on In-Network Computing: Programmable Data Plane and Technology Specific Applications. *IEEE Communications Surveys & Tutorials*, 25(1):701–761, 2023.

[16] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 90–106. Association for Computing Machinery, 2020.

[17] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal. Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration. In *Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 74–78, Nov. 2015.

[18] J. Langlet, R. Ben Basat, G. Oliaro, M. Mitzenmacher, M. Yu, and G. Antichi. Direct Telemetry Access. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, pages 832–849. Association for Computing Machinery, 2023.

[19] S. Linder and F. Ihle. P4TG. Mar. 24. URL: `https://github.com/uni-tue-kn/P4TG` (visited on 07/28/2024).

[20] S. Lindner, M. Häberle, and M. Menth. P4TG: 1 Tb/s Traffic Generation for Ethernet/IP Networks. *IEEE Access*, 11:17525–17535, 2023.

[21] S. Lindner, D. Merling, M. Häberle, and M. Menth. P4-Protect: 1+1 Path Protection for P4. In *P4 Workshop in Europe*, EuroP4'20, pages 21–27, Barcelona, Spain. Association for Computing Machinery, 2020.

[22] D. F. Macedo, D. Guedes, L. F. M. Vieira, M. A. M. Vieira, and M. Nogueira. Programmable Networks—From Software-Defined Radio to Software-Defined Networking. *IEEE Communications Surveys & Tutorials*, 17(2):1102–1125, 2015.

[23] T. Mai, H. Yao, S. Guo, and Y. Liu. In-Network Computing Powered Mobile Edge: Toward High Performance Industrial IoT. *IEEE Network*, 35(1):289–295, 2021.

[24] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[25] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal. Creating Complex Network Service with eBPF: Experience and Lessons Learned. *High Performance Switching and Routing (HPSR). IEEE.*, 18:21–23, Jan. 2018.

[26] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[27] NVIDIA. ConnectX NICs. July 24. URL: `https://www.nvidia.com/en-us/networking/ethernet-adapters/` (visited on 07/27/2024).

[28] Open Networking Foundation. OpenFlow Switch Specification. Version 1.5.1 (Protocol version 0x06). Technical report, Mar. 2015. URL: `https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf`.

[29] M. Pall. LuaJIT. Aug. 23. URL: `https://luajit.org/index.html` (visited on 07/26/2024).

[30] M. Pall, L. Gorrie, A. Wingo, M. Rottenkolber, D. P. Garcia, A. Takikawa, J. Tallon, D. Pino, A. Gall, C. Apreutesei, N. Nikolaev, K. Barone-Adesi, H. Huebner, J. Guerra, M. Wiget, N. Larosa, P. Bristow, A. P. de Castro, A. P. de Castro, N. Larosa, A. Nikishaev, D. Kožar, P. Cawley, A. Altshuler, R. Sommerhalder, T. Buhrmester, P. Kazmier, D. Majumdar, L. D. Cruz, M. Nazarov, K. Larsson, J. Cormack, B. Agricola, F. Geißler, R. M. Emerson, M. G, C. Graf, C. A. L. Perez, J. Loughridge, K. Wysocki, A. Perez, W. Adams, V. Fedin, V. Maffione, T. Korcak, T. Upthegrove, T. LaBerge, S. Markovic, S. Leinen, R. Hartlage, K. Kielhofner, J. Olsson, J. Liu, J. Fenton, J. Cunningham, H. Xiang, G. Nusshall, F. Bonk, E. Hope-Morley, D. Bacon, A. Makkar, A. Chong, A. Kordic, A. Kostrikov, and A. Spyridakis. Snabb Reference Manual. Nov. 2019. URL: `https://snabbco.github.io` (visited on 07/26/2024).

[31] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho. SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In *Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 86–92, Nov. 2015.

[32] F. Parola, R. Procopio, R. Querio, and F. Risso. Comparing User Space and In-Kernel Packet Processing for Edge Data Centers. *SIGCOMM Comput. Commun. Rev.*, 53(1):14–29, Apr. 2023.

[33] R. J. Recio, P. R. Culley, D. Garcia, B. Metzler, and J. Hilland. A Remote Direct Memory Access Protocol Specification. Technical report 5040, Oct. 2007. 66 pages.

[34] M. Rottenkolber. ConnectX: Review ConnectX-5 transmit/receive/forwarding performance on PCI-e Gen4. Feb. 2022. URL: `https://github.com/snabbco/snabb/issues/1471` (visited on 07/29/2024).

[35] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa. A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1237–1255, Apr. 2023.

[36] D. Scholz, H. Stubbe, S. Gallenmüller, and G. Carle. Key Properties of Programmable Data Plane Targets. In *International Teletraffic Congress*, pages 114–122, 2020.

[37] The Linux Foundation. NVIDIA Mellanox NICs Performance Report with DPDK 22.07. Technical report, Feb. 2023. URL: `http://fast.dpdk.org/doc/perf/DPDK_22_07_NVIDIA_Mellanox_NIC_performance_report.pdf` (visited on 07/30/2024).

[38] The Linux Foundation. P4 Language and Related Specifications. URL: `https://p4.org/specs/` (visited on 07/22/2024).

[39] The P4 Language Consortium. $P4_{16}$ Language Specification. Technical report, May 2023. URL: `https://p4.org/p4-spec/docs/P4-16-v1.2.4.html`.

[40] The P4.org Architecture Working Group. $P4_{16}$ Portable Switch Architecture (PSA). Technical report, Apr. 2021. URL: `https://p4.org/p4-spec/docs/PSA.html`.

[41] N. V. Tu, J.-H. Yoo, and J. W.-K. Hong. Building Hybrid Virtual Network Functions with eXpress Data Path. In *International Conference on Network and Service Management (CNSM)*, pages 1–9, 2019.

[42] B. Varga, J. Farkas, S. Kehrer, and T. Heer. Deterministic Networking (DetNet): Packet Ordering Function. Technical report 9550, Mar. 2024. 11 pages.

[43] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1345–1358. USENIX Association, Apr. 2022.

[44] C. Zhang, H. P. Joshi, G. F. Riley, and S. A. Wright. Towards a Virtual Network Function Research Agenda: A systematic Literature Review of VNF Design Considerations. *Journal of Network and Computer Applications*, 146:102417, 2019.

[45] T. Zhang, L. Linguaglossa, P. Giaccone, L. Iannone, and J. Roberts. Performance benchmarking of state-of-the-art software switches for NFV. *Computer Networks*, 188:107861, 2021.

List of Figures

List of Listings

Appendix

A. Diagrams of the Evaluation

The following pages contain the diagrams used for the evaluation of the experiments conducted. The throughput evaluation is depicted in green, the delay evaluation in blue, and the jitter evaluation in red. The diagrams are included in the appendix for completeness. In chapter 5 "Evaluation" the major parts of the diagrams are described. The diagrams are in the order of the experiments conducted. In the following, the pages of the diagrams for each experiment are listed:

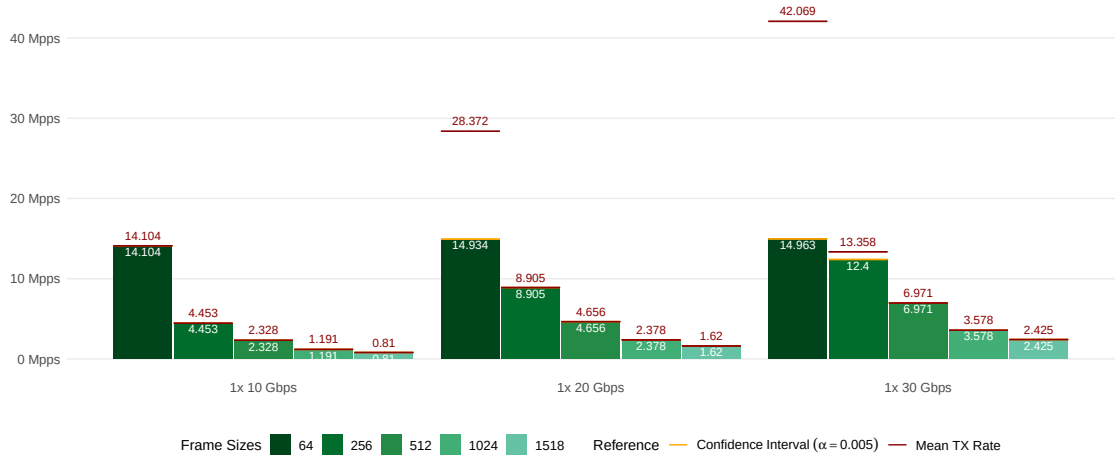

Figure A.1.: Experiment: Baseline single NIC; Flows: Single; Metric: Throughput.

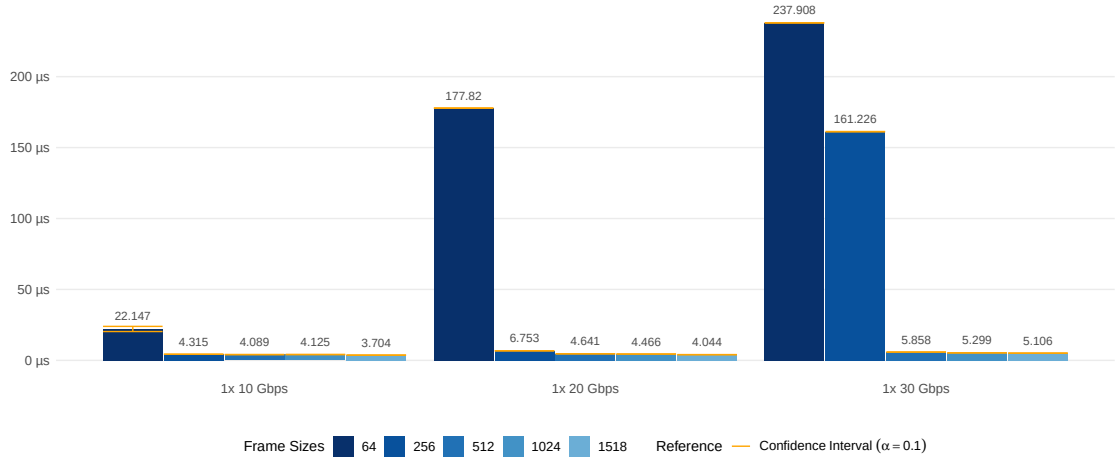

Figure A.2.: Experiment: Baseline single NIC; Flows: Single; Metric: Delay.

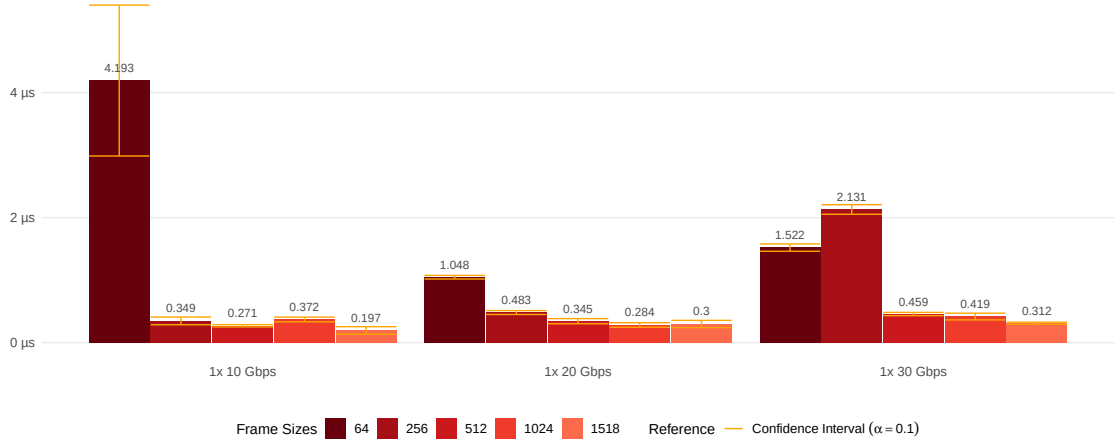

Figure A.3.: Experiment: Baseline single NIC; Flows: Single; Metric: Jitter.

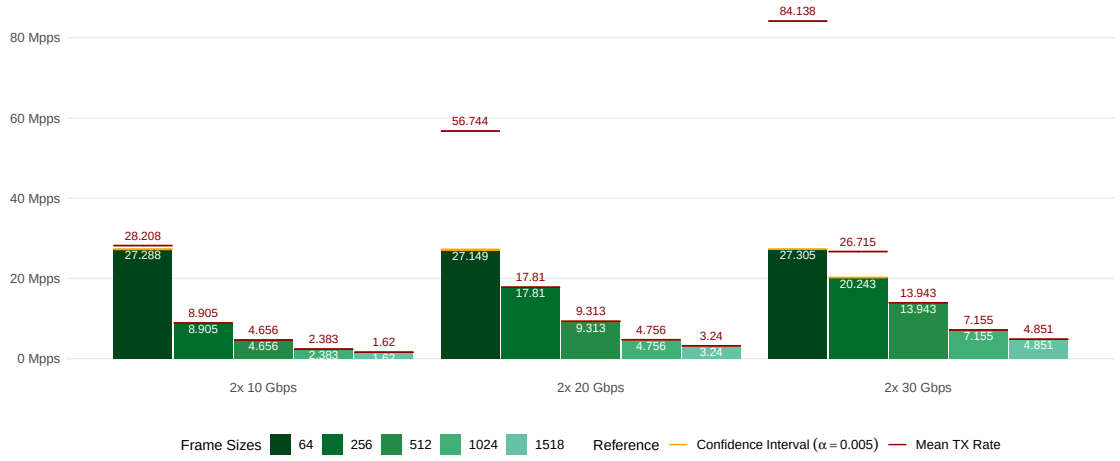

Figure A.4.: Experiment: Baseline single NIC; Flows: Multiple; Metric: Throughput.

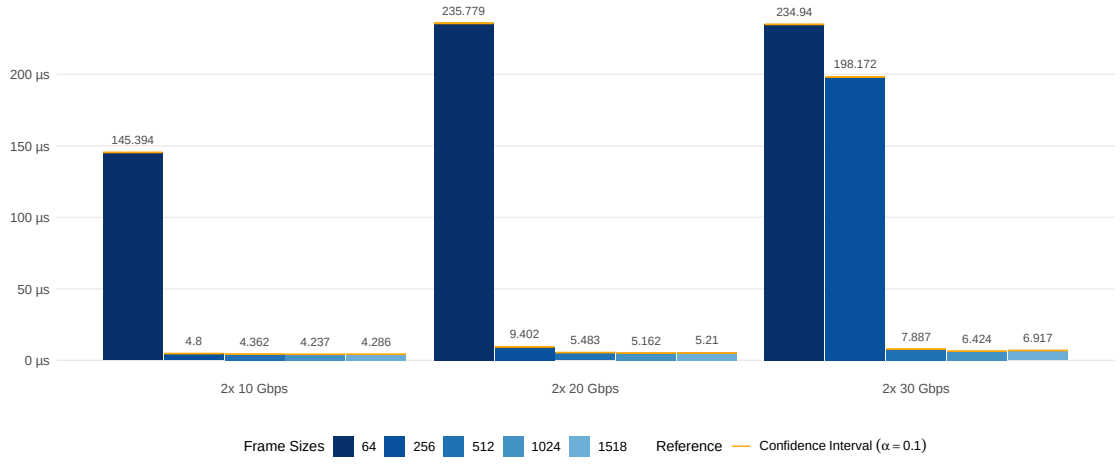

Figure A.5.: Experiment: Baseline single NIC; Flows: Multiple; Metric: Delay.

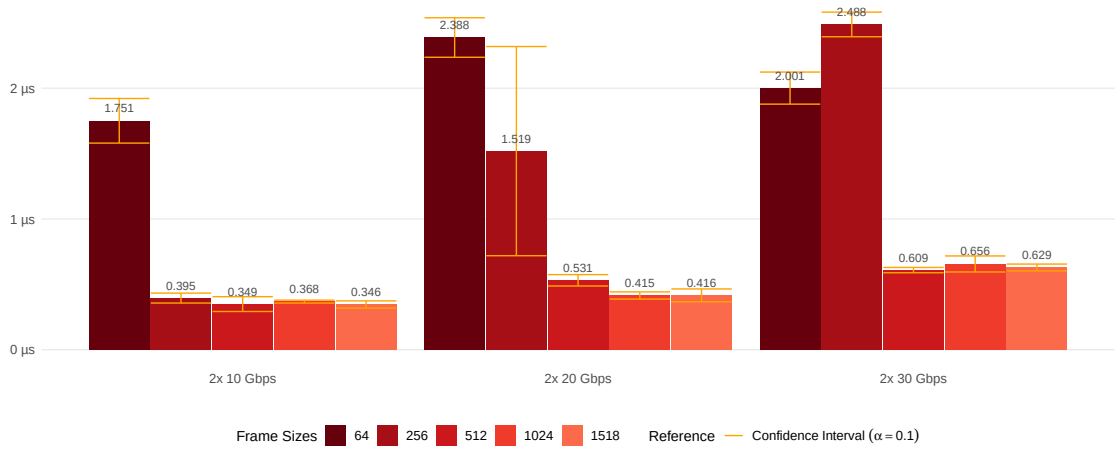

Figure A.6.: Experiment: Baseline single NIC; Flows: Multiple; Metric: Jitter.

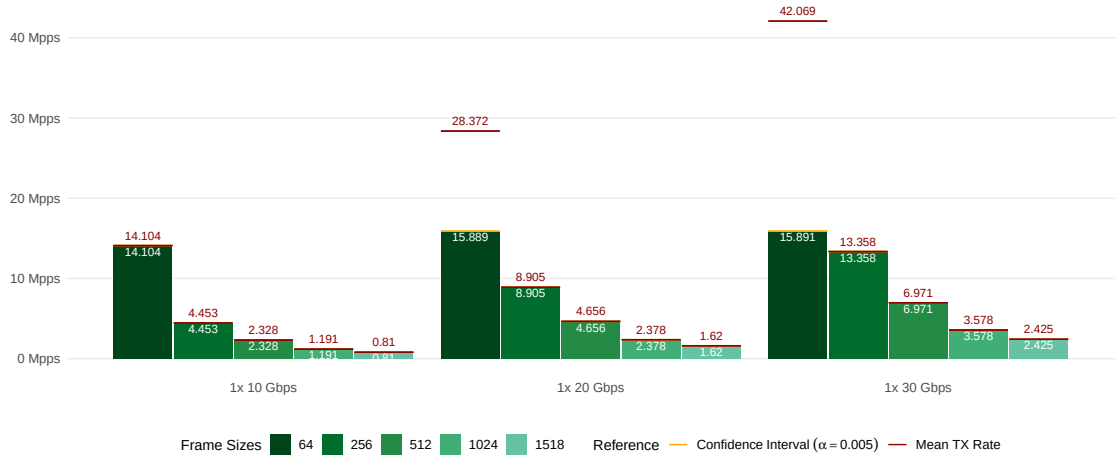

Figure A.7.: Experiment: Baseline multiple NICs; Flows: Single; Metric: Throughput.

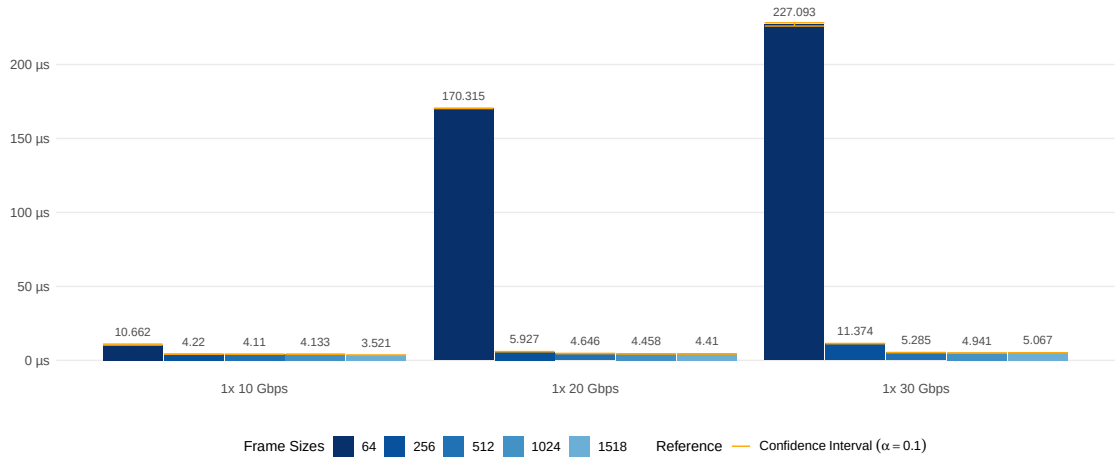

Figure A.8.: Experiment: Baseline multiple NICs; Flows: Single; Metric: Delay.

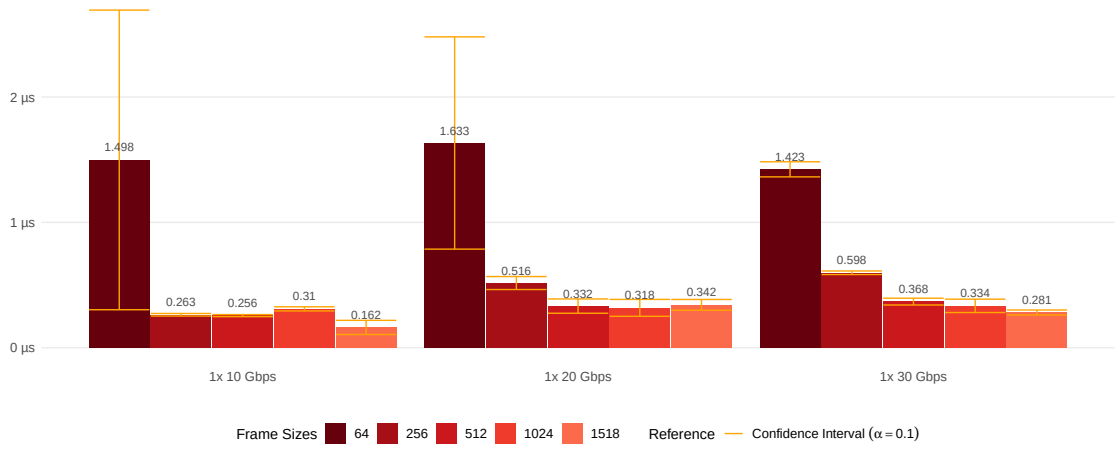

Figure A.9.: Experiment: Baseline multiple NICs; Flows: Single; Metric: Jitter.

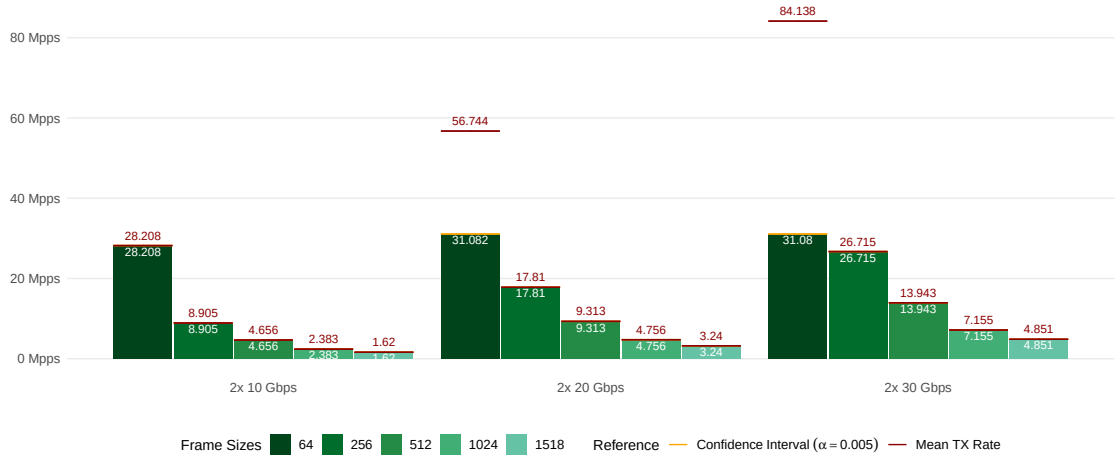

Figure A.10.: Experiment: Baseline multiple NICs; Flows: Multiple; Metric: Throughput.

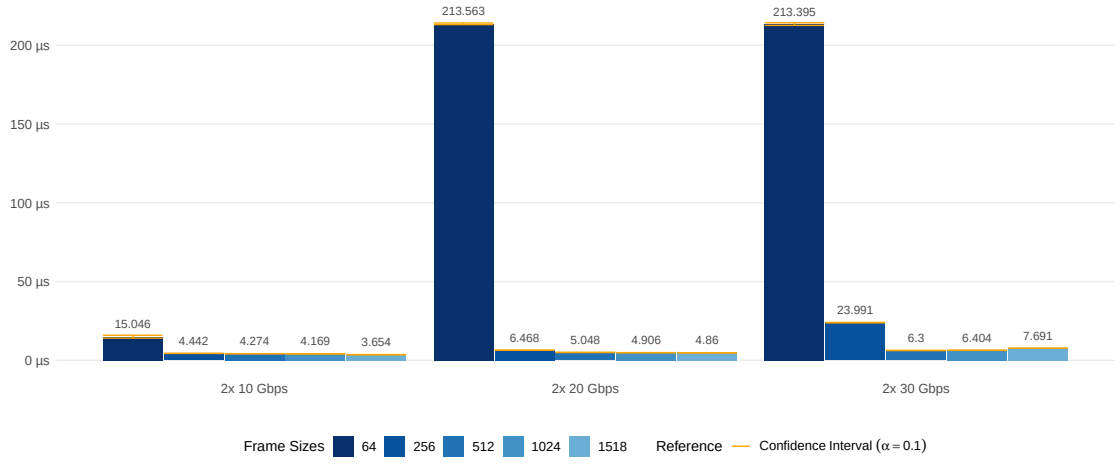

Figure A.11.: Experiment: Baseline multiple NICs; Flows: Multiple; Metric: Delay.

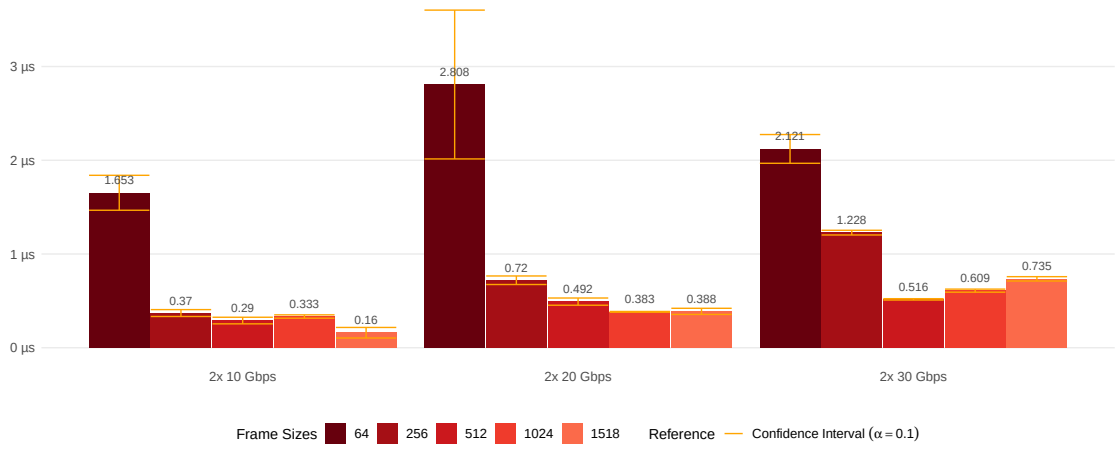

Figure A.12.: Experiment: Baseline multiple NICs; Flows: Multiple; Metric: Jitter.

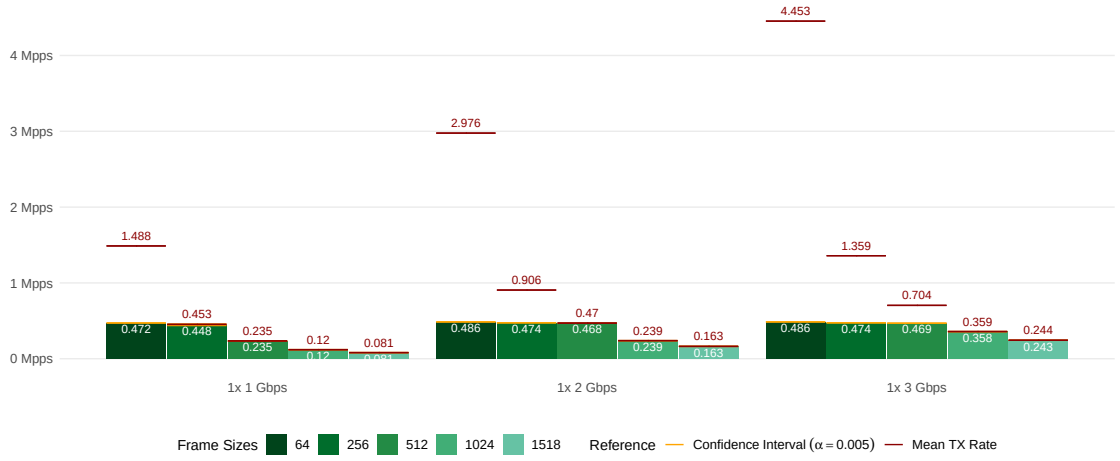

Figure A.13.: Experiment: NetEm (5ms/0.1ms); Flows: Single; Metric: Throughput.

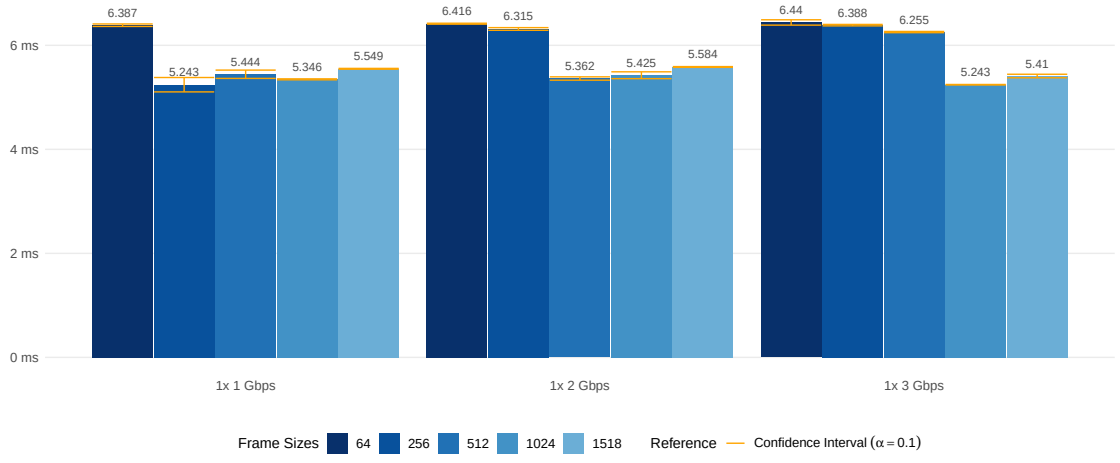

Figure A.14.: Experiment: NetEm (5ms/0.1ms); Flows: Single; Metric: Delay.

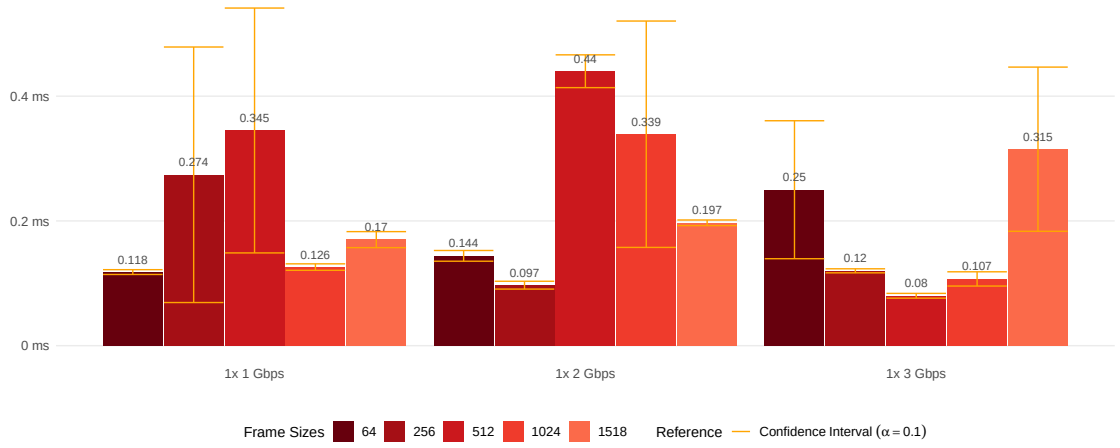

Figure A.15.: Experiment: NetEm (5ms/0.1ms); Flows: Single; Metric: Jitter.

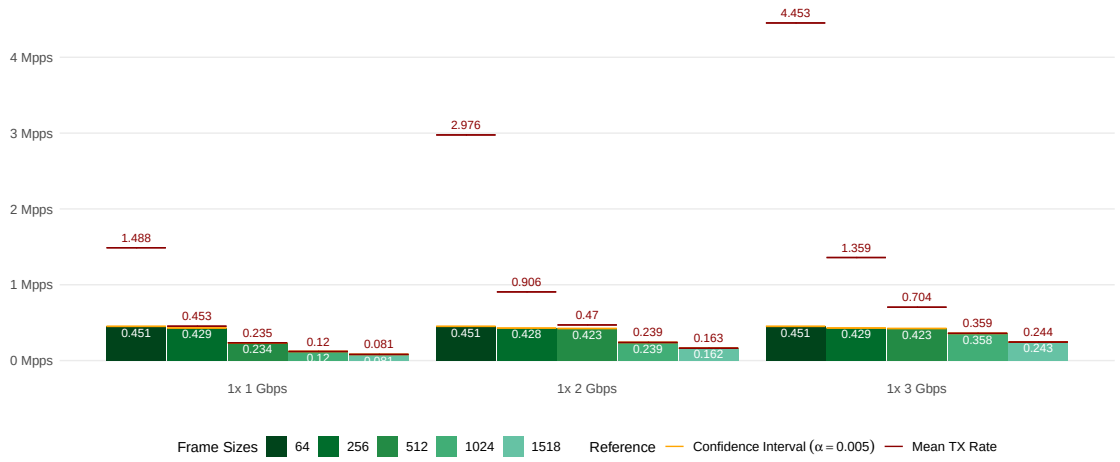

Figure A.16.: Experiment: NetEm (50ms/1ms); Flows: Single; Metric: Throughput.

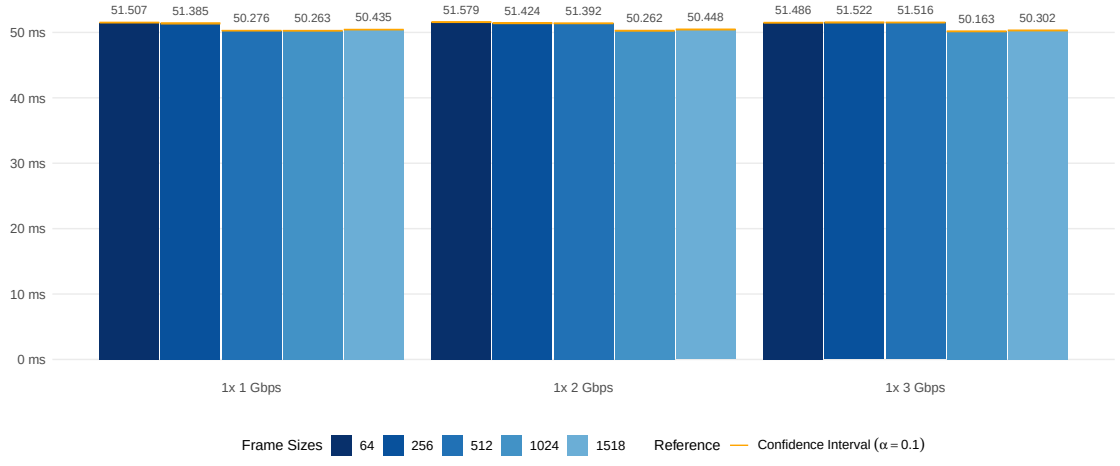

Figure A.17.: Experiment: NetEm (50ms/1ms); Flows: Single; Metric: Delay.

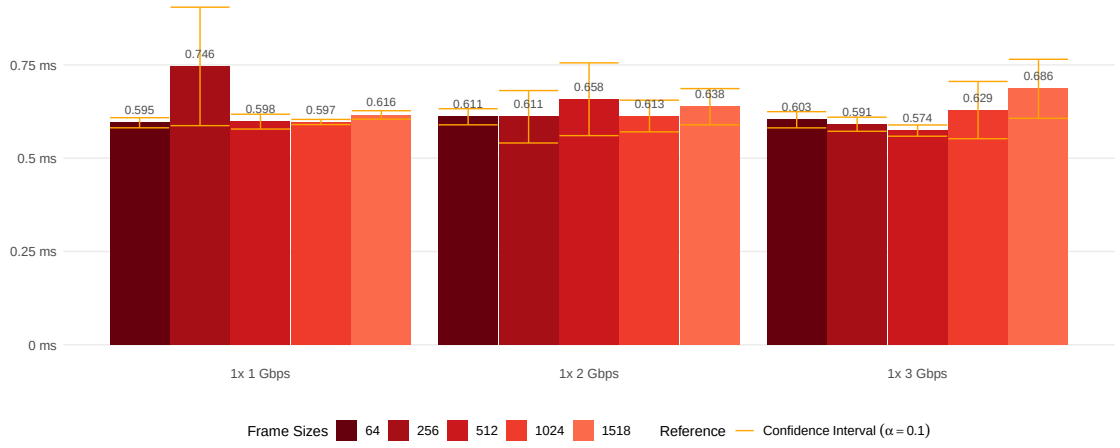

Figure A.18.: Experiment: NetEm (50ms/1ms); Flows: Single; Metric: Jitter.

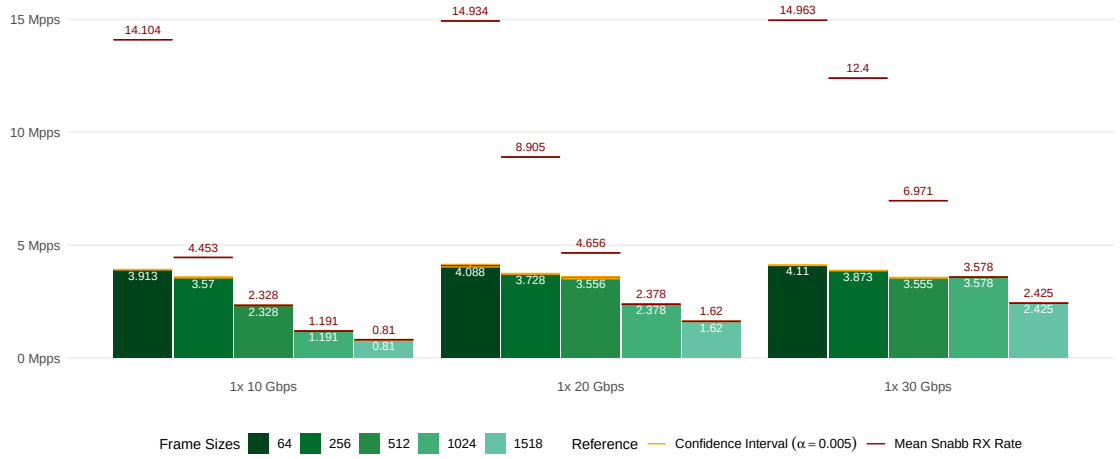

Figure A.19.: Experiment: Delayer (noTimeDrop/5ms/0.1ms); Flows: Single; Metric: Throughput.

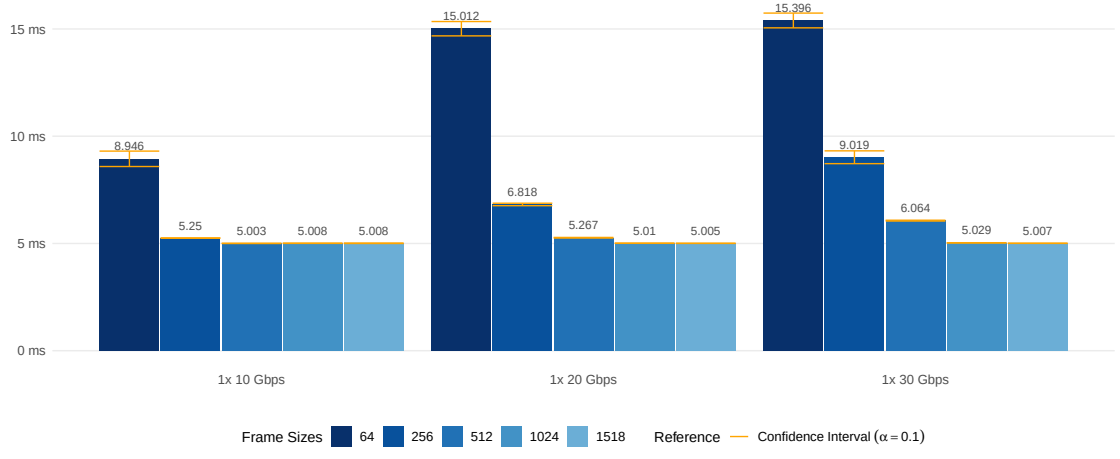

Figure A.20.: Experiment: Delayer (noTimeDrop/5ms/0.1ms); Flows: Single; Metric: Delay.

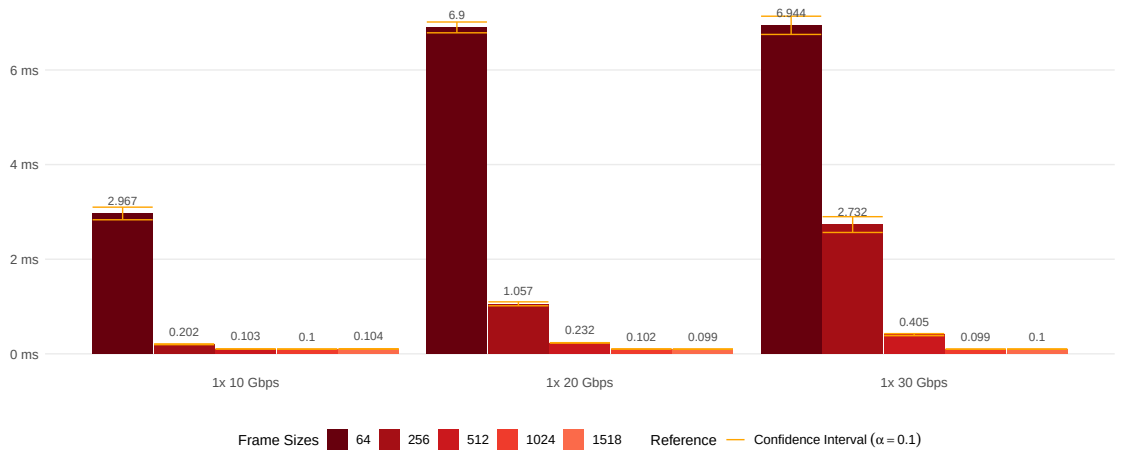

Figure A.21.: Experiment: Delayer (noTimeDrop/5ms/0.1ms); Flows: Single; Metric: Jitter.

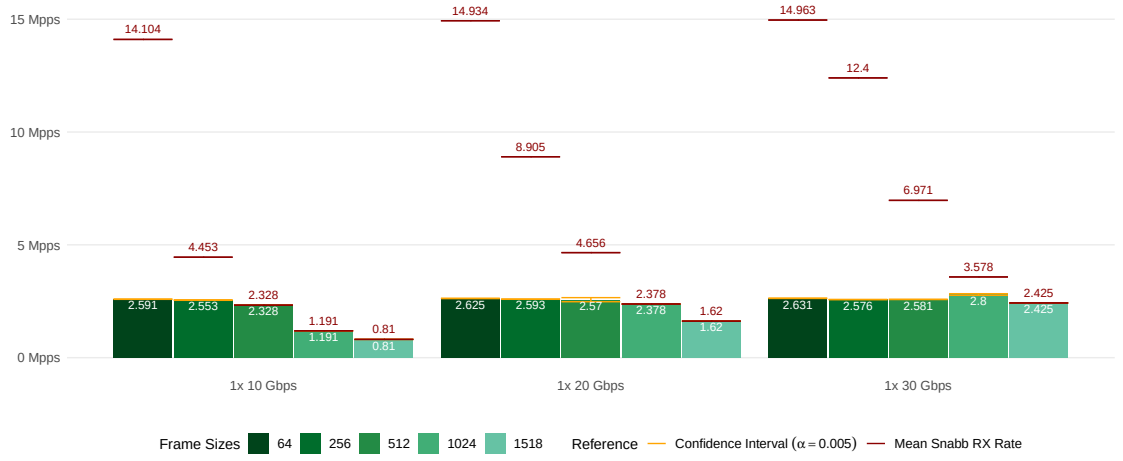

Figure A.22.: Experiment: Delayer (noTimeDrop/50ms/1ms); Flows: Single; Metric: Throughput.

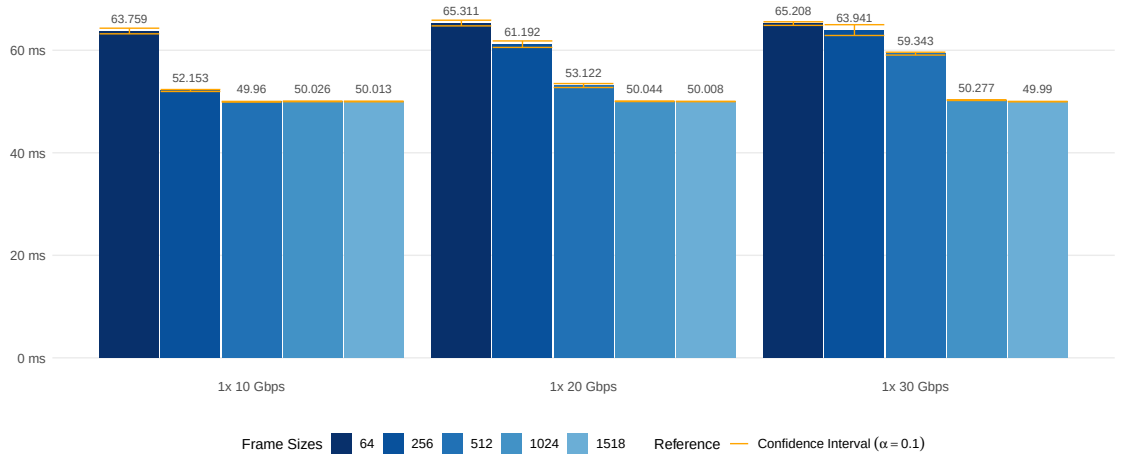

Figure A.23.: Experiment: Delayer (noTimeDrop/50ms/1ms); Flows: Single; Metric: Delay.

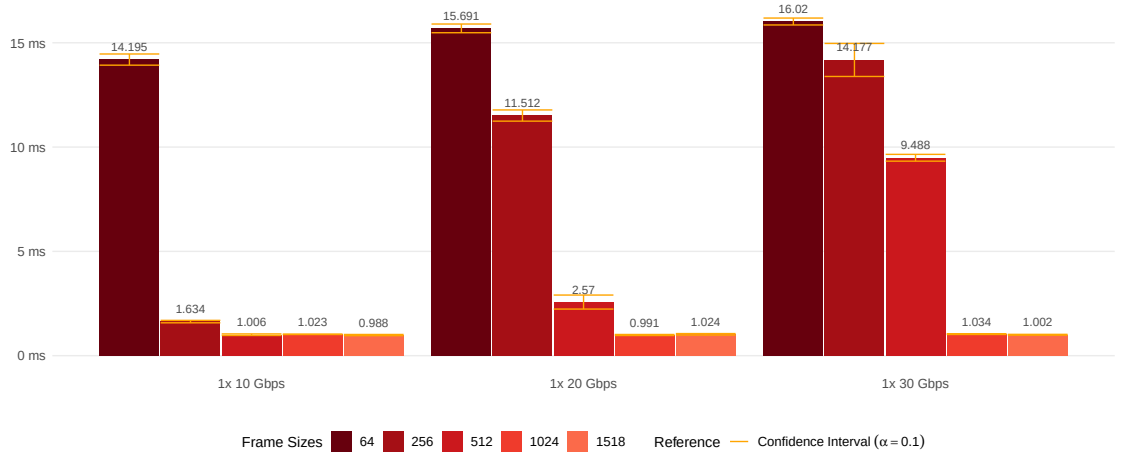

Figure A.24.: Experiment: Delayer (noTimeDrop/50ms/1ms); Flows: Single; Metric: Jitter.

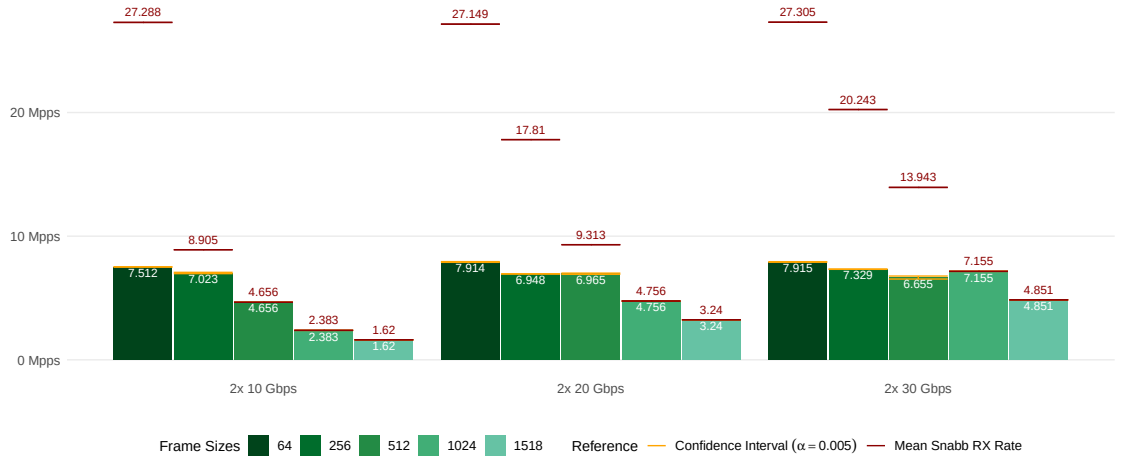

Figure A.25.: Experiment: Delayer (noTimeDrop/5ms/0.1ms); Flows: Multiple; Metric: Throughput.

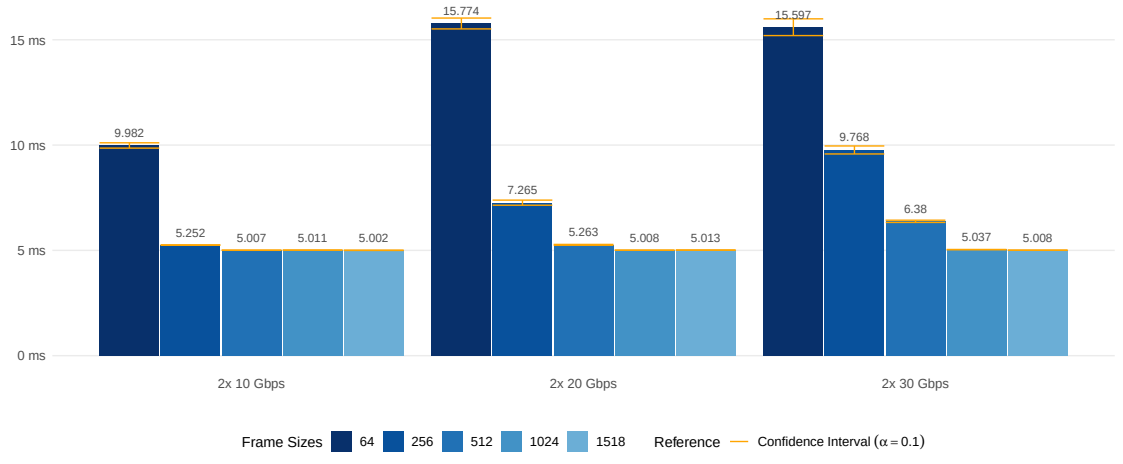

Figure A.26.: Experiment: Delayer (noTimeDrop/5ms/0.1ms); Flows: Multiple; Metric: Delay.

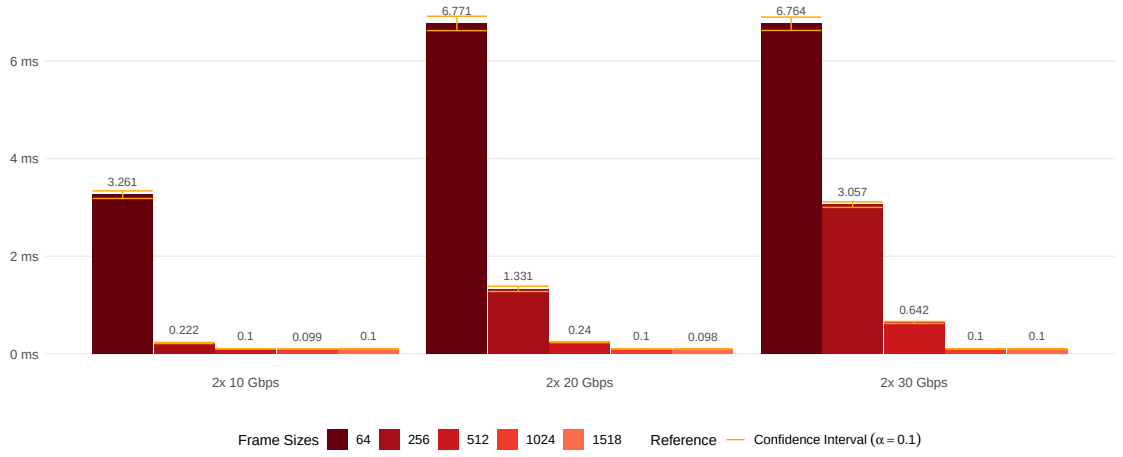

Figure A.27.: Experiment: Delayer (noTimeDrop/5ms/0.1ms); Flows: Multiple; Metric: Jitter.

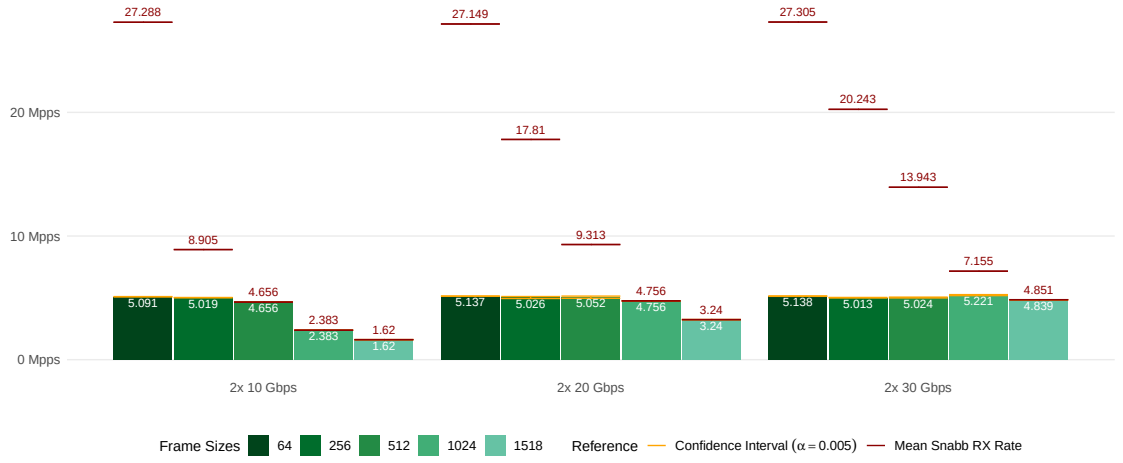

Figure A.28.: Experiment: Delayer (noTimeDrop/50ms/1ms); Flows: Multiple; Metric: Throughput.

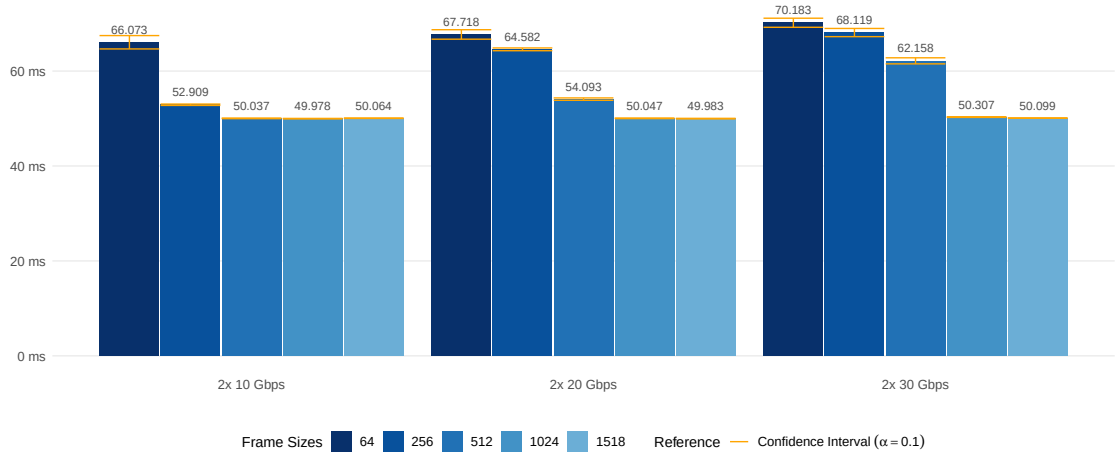

Figure A.29.: Experiment: Delayer (noTimeDrop/50ms/1ms); Flows: Multiple; Metric: Delay.

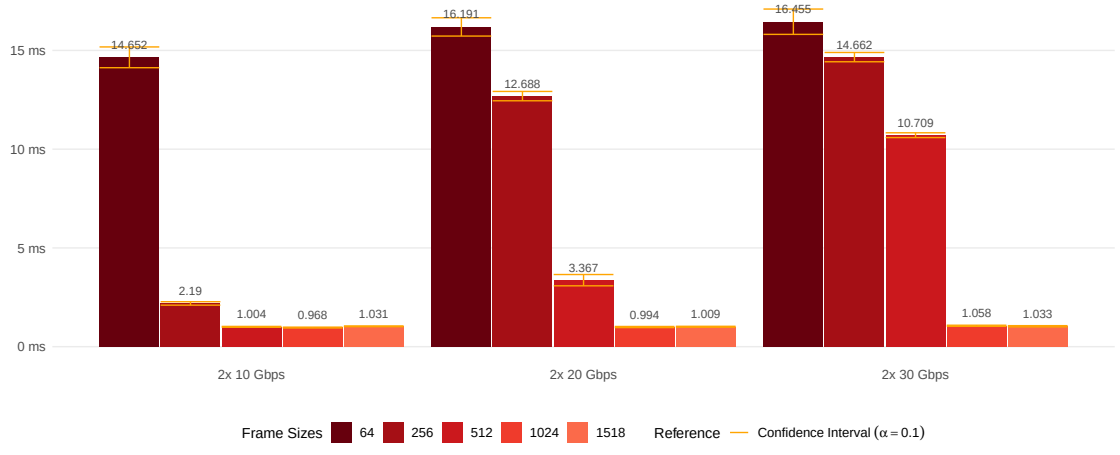

Figure A.30.: Experiment: Delayer (noTimeDrop/50ms/1ms); Flows: Multiple; Metric: Jitter.

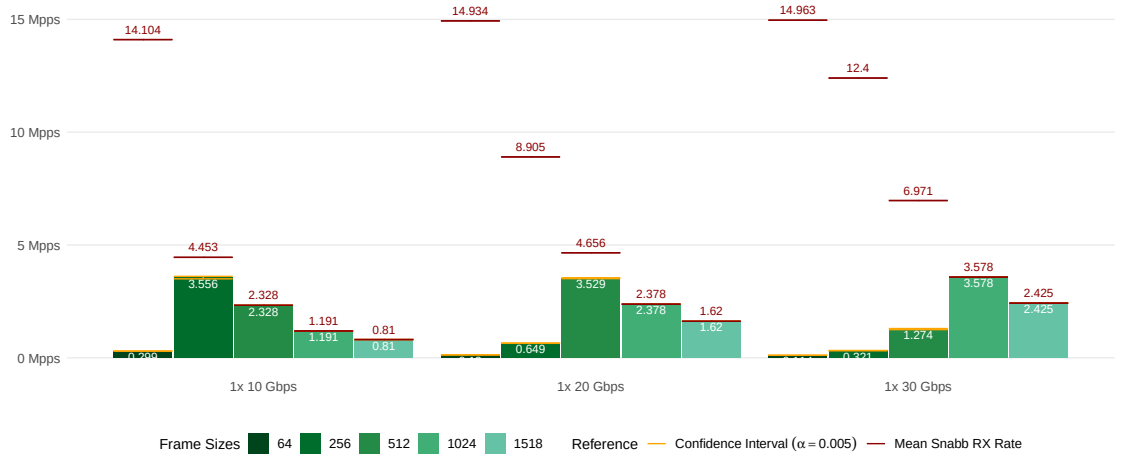

Figure A.31.: Experiment: Delayer(TimeDrop/5ms/0.1ms); Flows: Single; Metric: Throughput.

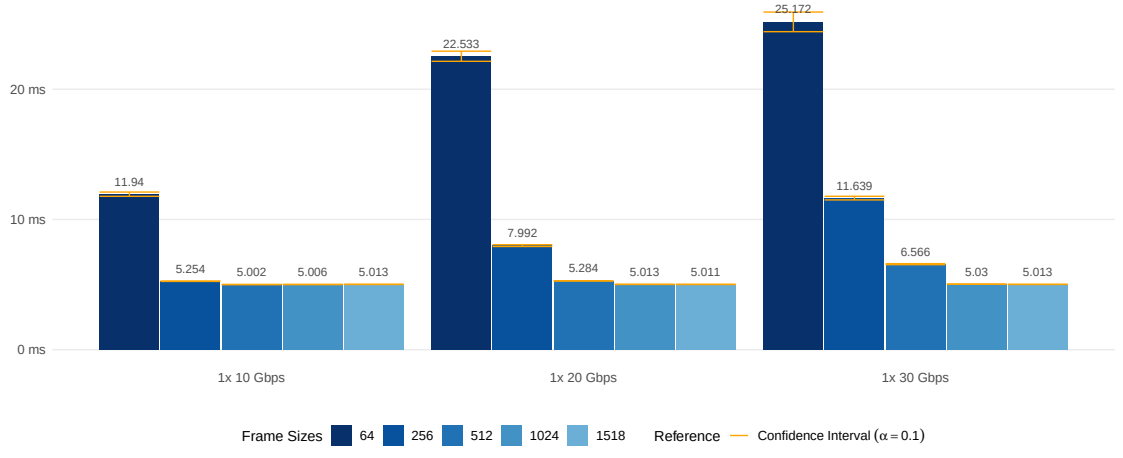

Figure A.32.: Experiment: Delayer(TimeDrop/5ms/0.1ms); Flows: Single; Metric: Delay.

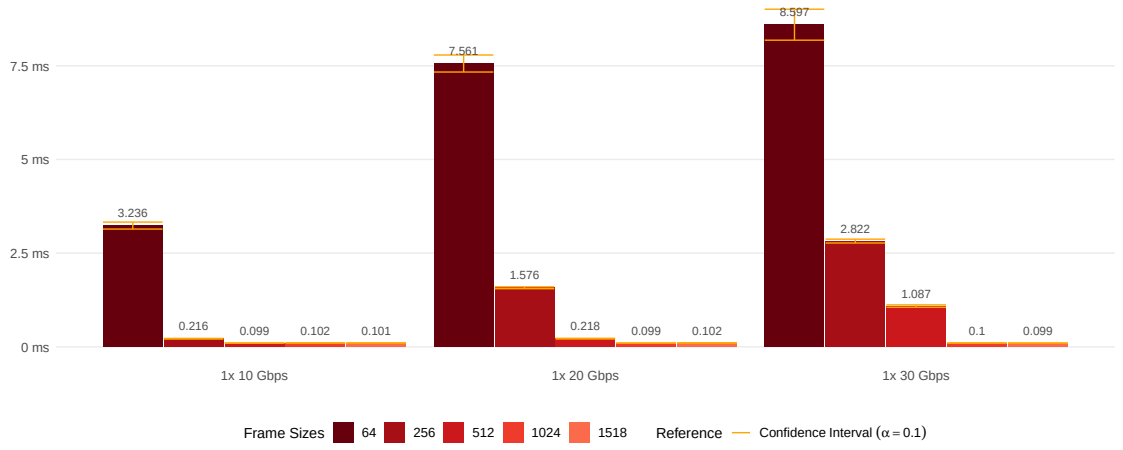

Figure A.33.: Experiment: Delayer(TimeDrop/5ms/0.1ms); Flows: Single; Metric: Jitter.

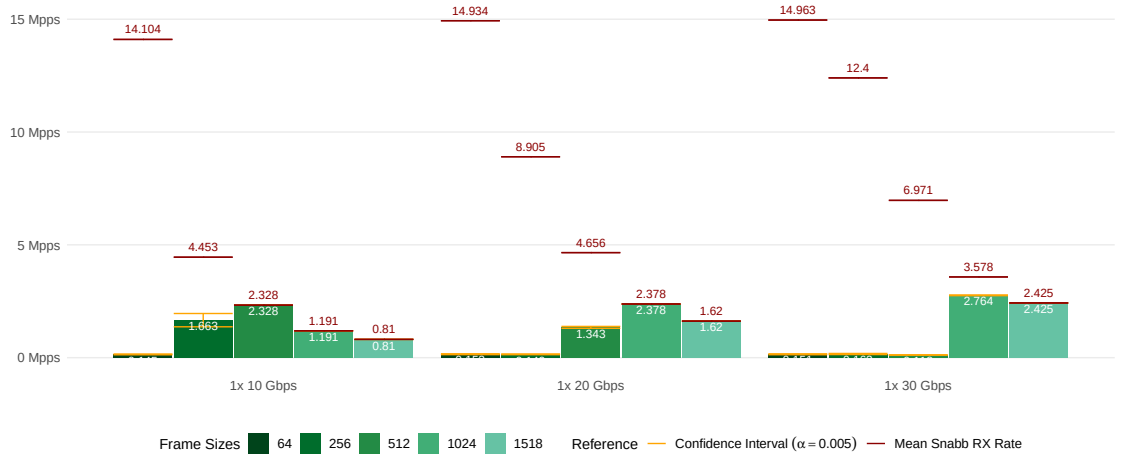

Figure A.34.: Experiment: Delayer (TimeDrop/50ms/1ms); Flows: Single; Metric: Throughput.

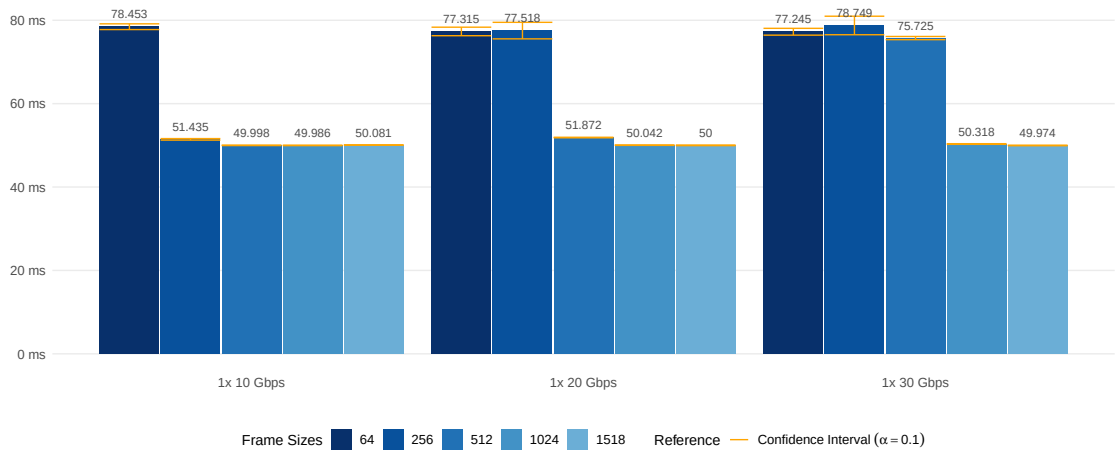

Figure A.35.: Experiment: Delayer (TimeDrop/50ms/1ms); Flows: Single; Metric: Delay.

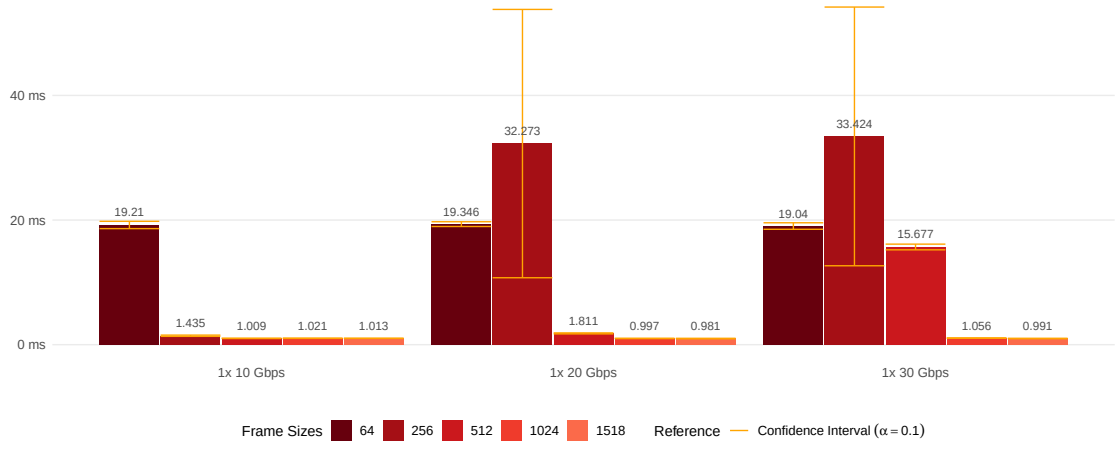

Figure A.36.: Experiment: Delayer (TimeDrop/50ms/1ms); Flows: Single; Metric: Jitter.

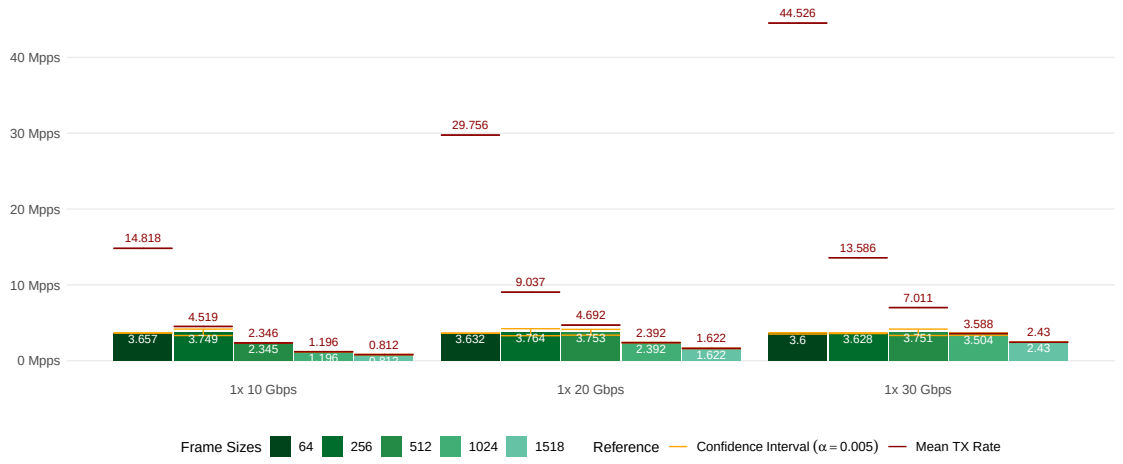

Figure A.37.: Experiment: Reorder (50µs Max Delay); Flows: Single; Metric: Throughput.

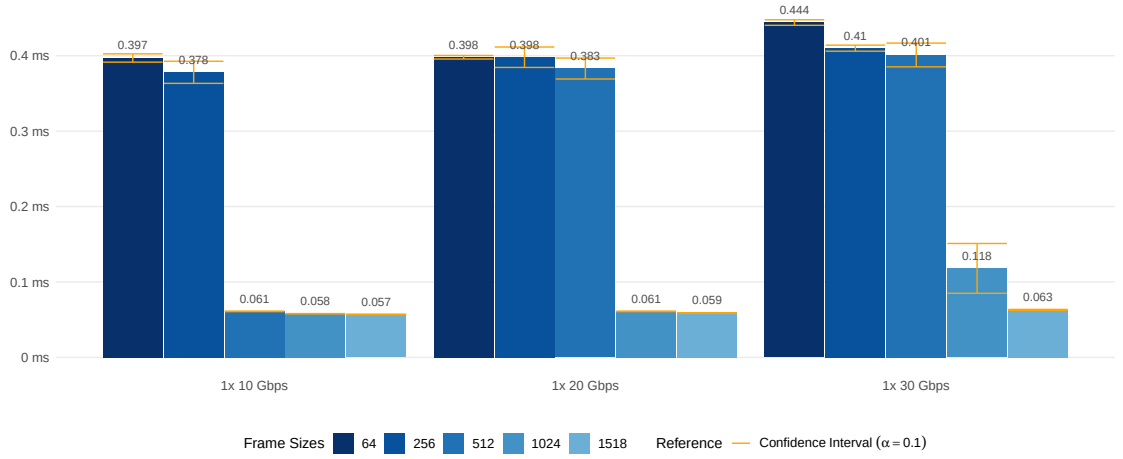

Figure A.38.: Experiment: Reorder (50µs Max Delay); Flows: Single; Metric: Delay.

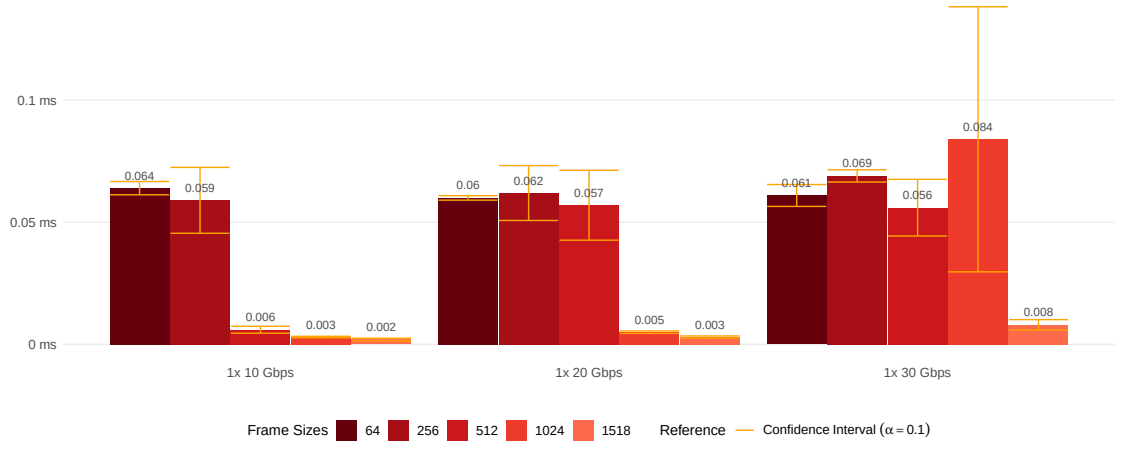

Figure A.39.: Experiment: Reorder (50µs Max Delay); Flows: Single; Metric: Jitter.