Exploring high-level transport programming through eXpress Data Path

by

Pedro Vitor Valença Mizuno

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Pedro Vitor Valença Mizuno 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The transport layer is a layer from the network stack responsible for providing several essential services, such as congestion control, and lost packet detection and retransmission. Moreover, this layer is constantly changing to satisfy the new demands present in the network. This includes the increase in traffic, the leveraging of new hardware, and the release of new workloads.

However, implementing these protocols is a considerably challenging task due to several reasons. First, the available documentation about these algorithms is written in natural language and can be long, which may result in misinterpretations, and consequently, incorrect implementations. Second, while developing transport algorithms, programmers must deeply study the environment where the protocol will run to find the best-suiting built-in helper functions and data structures available for their implementations. For this reason, the protocols are usually implemented as large sections of optimized code that are challenging to navigate, read, and modify. Third, since the transport layer is located between two other layers, namely application and network, transport algorithms must handle the details of these interactions, which can depend on the execution environment.

In this thesis, we introduce Modular Transport Programming (MTP). This event-driven framework allows the declaration of transport algorithms with a high-level, yet, precise language. MTP abstracts low-level details by providing a set of high-level built-in functions and constructs focused solely on transport programming. It also abstracts the interactions between the transport layer and its neighbours by having interfaces responsible for such communication. Moreover, it improves the readability of MTP code by having a modularized design that clearly shows how each module interacts with the other and how the events are processed.

To explore the feasibility and effectiveness of MTP, we chose to implement transport protocols in the Linux Kernel. The Linux Kernel and its transport layer implementations are some of the most notable examples that highlight the challenges stated above. These implementations can require thousands of lines of code and hundreds of functions, which are difficult to navigate due to the low-level C used to write their codes. Additionally, transport protocol implementations in the Kernel are tightly linked with the neighbouring layers of the network stack, utilizing complex data structures for packet processing and socket interface. Thus, implementing protocols from the start or modifying already existing ones can be a considerably difficult task.

Nevertheless, instead of directly implementing protocols to the Linux Kernel, we opted to use the eXpress Data Path (XDP) framework as our first step. This framework allows

a simple and safe loading of new user-defined code to the Linux Kernel. Moreover, several characteristics of the framework are ideal for the implementation of transport layer protocols, such as its loading point being in the Network Interface Card (NIC) driver and allowing Kernel bypass. Thus, this thesis also introduces a back-end developed in the XDP framework, to which the MTP code can be compiled using our code generator. During the development of this back-end, we explored XDP in depth to make wise design decisions. This includes utilizing a set of data structures best suited for our model, making the most of the helper functions available in its libraries, and bypassing the limitations of this framework.

Finally, we show that it is possible to specify the logic of transport protocols in a high-level language and translate it to XDP. We implemented two protocols, TCP and RoCEv2, in the MTP language and successfully translated them with our compiler to the XDP back-end. Moreover, between two servers connected by 100Gbps links, the TCP implementation presented 80Gbps of throughput with 16 cores processing packets. Meanwhile, RoCEv2 translation is functional but still needs further optimizations to reach its expected performance. Lastly, we evaluate the strengths and weaknesses of XDP for transport programming.

Acknowledgements

I would like to thank my supervisor, Mina Tahmasbi Arashloo, for all the support she gave me during my MMath program. I am grateful for all the countless and fun hours spent discussing the design of MTP, improvements to the language, new features of XDP back-end, and the many curious events that happened while working with XDP. All the knowledge that Mina presented during this program has always amazed me and I am fortunate enough to have learned a lot from her. Finally, I thank Mina for her patience in answering my (sometimes trivial) questions and for always providing all the necessary help for me and her other students.

I thank Ali Mashtizadeh and Samer Al-Kiswany for accepting being members of my committee and for providing insightful feedback for my thesis. I also thank Chengnian Sun for the excellent course he lectured and the great support he provided for my group project.

Finally, I also would like to thank a few more people: first, a special thank you to Kimiya Mohammadtaheri for joining the MTP project and doing an excellent job of re-working the MTP compiler; Crystal Zhou and Beihao Zhou, both URAs who helped to understand many concepts of XDP; Alex Lin, another URA, who gave the first steps for the TCP example implementation; the URAs Arnav Nagpal, Danny Akbarzadeh, Jiashuo Shi, Joshua Johnson, the URF student Pradhyumnan V., and the MURA student Linfan Qian for working on many more targets of MTP and expanding this project; and, finally, Chris Neely and Mario Baldi from AMD for assisting on the design of MTP.

Dedication

This thesis is dedicated to my family and friends, who make me company even so far away.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements v

Dedication vi

List of Figures x

List of Tables xii

List of Abbreviations xiii

1 Introduction 1

1.1 The changes of the transport layer 2

1.2 The challenges of implementing and understanding transport protocols . . 6

1.3 Why high-level modular transport programming? 8

1.4 eBPF/XDP as the target 10

1.5 Related Work 11

1.6 Contributions 12

2 Modular Transport Programming **14**
2.1 MTP logical model . . . 14
2.1.1 Event-related modules . . . 16
2.1.2 Incoming Interfaces . . . 17
2.1.3 Outgoing Interfaces . . . 18
2.2 MTP Language . . . 20

3 Exploring eXpress Data Path **29**
3.1 eBPF/XDP background . . . 29
3.2 XDP back-end logical model . . . 34
3.2.1 User space side . . . 34
3.2.2 XDP Side . . . 37
3.3 Choosing eBPF Maps . . . 40
3.4 Multithreading . . . 45
3.4.1 User space Threads . . . 46
3.4.2 Kernel Threads . . . 47
3.4.3 State Consistency . . . 48
3.5 Packet Mutation . . . 50

4 XDP Code Generator **53**
4.1 The basic translation . . . 54
4.2 The complexity added by XDP . . . 55
4.2.1 Void pointers . . . 55
4.2.2 Loops . . . 57
4.2.3 Unifying minor events . . . 60
4.2.4 Out-of-bounds access checks . . . 61

5 Evaluation **63**

5.1 Evaluating MTP 63

5.1.1 Expressiveness 64

5.1.2 Reducing development effort 66

5.2 Evaluating XDP back-end 67

5.2.1 TCP implementation 68

5.2.2 RoCEv2 implementation 69

5.3 XDP as a transport layer framework 71

5.3.1 Strengths of XDP 72

5.3.2 Weaknesses of XDP 73

5.3.3 Final considerations 74

6 Conclusion **76**

6.1 Summary of contributions 76

6.2 Future Work 77

References **79**

List of Figures

2.1 MTP's logical model of the transport layer. 15
2.2 Example of a Application-to-Transport interface declaration. 23
2.3 Example of a Network-to-Transport interface declaration. 24
2.4 Example of a dispatch table declaration. 25
2.5 Event processor to send packets from application request. 27
2.6 Event processor to handle the arrival of data. 28

3.1 The process to load an eBPF program. 30
3.2 Summary of XDP's architecture. 31
3.3 An overview of MTP's XDP back-end. 35
3.4 eBPF map benchmarks. Results are from read followed by write (W = Write) 41
3.5 Effect of contention on eBPF map of maps. 42
3.6 A summary of the maps utilized for application event processing, the keys and the values retrieved with them. 43
3.7 An overview of the threads utilized in the back-end. 45
3.8 Common TCP packet 51
3.9 TCP packet after getting resized and copied the metadata header. 51
3.10 TCP packet after having application and network metadata appended to it. 52

4.1 Example of *bpf_loop* call. 58
4.2 Example of nested loops. 60

4.3 Loop argument tree generated from the nest loop example. 60

5.1 The progression of TCP congestion window. 69

5.2 Total throughput with increasing number of cores. 70

List of Tables

1.1 Number of lines of code in the Linux Kernel for TCP code 8

5.1 Reduction in development effort enabled by MTP for the TCP implementation in XDP 67

5.2 Reduction in development effort enabled by MTP for the RoCEv2 implementation in XDP 67

List of Abbreviations

AF_XDP Address Family XDP 31, 33, 34, 37, 40, 44, 46, 47, 49, 50, 66, 67, 72–74, 78

eBPF extended Berkeley Packet Filter 2, 6, 10, 12, 13, 29–42, 44, 46–50, 53–55, 57, 58, 61, 62, 66–75, 77, 78

EP Event Processor 19, 21, 25, 26, 28, 57, 65–68, 70

IRQ Interrupt Request 68, 73

MTP Modular Transport Programming 1, 2, 8–26, 28–31, 34, 36–41, 44, 53–58, 60, 61, 63–67, 69, 70, 76–78

NIC Network Interface Card 2, 4, 6, 29–31, 34, 37, 39, 47, 50, 68, 71–73

PFC Priority Flow Control 4

QUIC Quick UDP Internet Connections 1, 5, 6, 19, 28

RDMA Remote Direct Memory Access 4, 65, 71

RoCEv2 RDMA over Converged Ethernet version 2 4, 6, 13, 19, 64–67, 70, 71, 77, 78

TCP Transmission Control Protocol 1, 3–5, 7, 11–13, 17, 19, 20, 22–24, 26, 27, 33, 37, 38, 42, 43, 50–52, 54, 64–69, 71, 77

WQE Work Queue Element 19, 65

XDP eXpress Data Path 2, 6, 8, 10–13, 29–50, 52–58, 61–63, 65–78

Chapter 1

Introduction

The transport layer is of great importance in the overall network architecture. The protocols and algorithms implemented in this layer handle several services that ensure reliable and efficient communication between applications. Flow and congestion control, data segmentation, and loss detection and recovery are common services in the transport layer. These services and many others are provided by an ever-expanding set of transport layer algorithms, from the classic Transmission Control Protocol (TCP) [24] to more recent examples such as Quick UDP Internet Connections (QUIC) [51].

Due to the transport layer's crucial role, several works aim to improve the performance of already existing algorithms and services [14, 28, 44, 72], or create new protocols to supply the demand for new emerging applications, hardware, and traffic patterns [27, 39, 51, 60, 65, 68]. However, developing transport layer algorithms can be a considerably demanding task due to factors such as their sheer complexity, how they interact with the layers of the network stack, and the environment they run in. Thus, this thesis explores how to reduce the effort spent in developing transport layer algorithms by allowing a simpler way to implement them. The result of this research is Modular Transport Programming (MTP).

MTP is a modular and event-driven programming framework that enables the declaration of transport protocols, focusing on enhancing abstraction and providing a more modular developing environment. MTP provides a domain-specific programming language to declare algorithms at a high-level, yet, precisely. Specifically, it provides a set of constructs focused on transport programming and requires an explicit declaration of how each of the algorithm's events must be executed. MTP also provides a set of compilers responsible for translating MTP code to different execution environments (i.e. targets), varying across a wide range of targets, from software to hardware.

We chose to explore eXpress Data Path (XDP) [43] as one of the first compilation targets of MTP. XDP is a type of program available in the extended Berkeley Packet Filter (eBPF) framework [6], which is a platform that allows the addition of user-defined code to certain points (hooks) of the Linux Kernel, simply and securely. We hypothesized that XDP can be well-suited to be one of the targets of MTP since XDP programs are attached to the hook of the Network Interface Card (NIC) driver and XDP has a packet-driven nature; both characteristics resulting in an environment that allows packet processing at an early stage. Additionally, XDP permits Kernel network stack bypass, avoiding unnecessary processing, since the packet processing is done in the XDP program itself.

Thus, this thesis presents MTP framework, including its event-driven and modular logical model, as well as the high-level language that describes it. Moreover, we created an XDP back-end that serves as the basis for MTP programs to be translated into and attached to the Linux Kernel. While exploring XDP, we made several design decisions based on the platform's characteristics and constraints, impacting its performance and functionalities. And two protocols were developed in MTP to evaluate the language's expressiveness and the performance of the target.

The rest of this chapter will present a brief summary of how the transport layer has evolved over a wide range of algorithms and how it is still in ongoing development. Additionally, this chapter will further explain the motivation behind the design of MTP and why XDP was chosen as its first target. Finally, a summary of the contributions of this thesis are presented.

1.1 The changes of the transport layer

The transport layer is responsible for providing various services that are relevant to the network, such as congestion control and loss detection and recovery. This layer is in constant change to accommodate the evolution of the network. These changes occur due to the increase in traffic [41, 44, 72, 80] and the deployment of new workloads [51], hardware [65], and network architectures [27, 68]. And new protocols and extensions of older ones are introduced to meet the new requirements caused by these changes. Therefore, this section will present an overview of how diverse the transport layer is, how it changes over time, and how this is an ongoing trend.

TCP: in 1974, Cerf and Kahn [24] designed the Transmission Control Program, later named Transmission Control Protocol (TCP). This transport layer protocol introduced several services that would be commonly applied to many other algorithms of this layer

later on. These services included port addressing, reassembly and sequencing, flow control, duplicate detection and retransmission, and others. So, due to the services provided, TCP became a staple among the transport layer protocols.

Although TCP proved to be quite effective, various modifications and additions have happened to TCP over the years. As the computer networks increased considerably in size and traffic during the 80s, TCP connections started to experience noticeable congestion problems. To remedy this problem, Jacobson and Karels [44] proposed TCP Tahoe, which expanded the protocol by including new mechanisms for congestion control. Later, a new expansion called TCP Reno [72] was proposed to introduce fast recovery from packet loss. Over the years, many other variations in TCP's congestion control, loss detection and recovery algorithms were proposed to improve for traffic with higher throughput [41, 46, 73, 80], adopt new technologies and design philosophies to optimize its performance [14, 28, 29, 62, 78], and fulfill the specific requirements of new network infrastructures [17]. Works such as TCP-RL [62] and Orca [14] (from 2019 and 2020) show that there is still much to be done in the transport layer, even in recent years.

Data center transport: new protocols have been created to fulfill the requirements of data centers. Although TCP can be employed in data center networks, its performance leaves much to be desired in such network architecture. One reason is that traditional TCP relies solely on packet loss as a metric for congestion [64]. This can cause problems in data centers because until a packet drop is detected, many low-latency flows may have already been impacted negatively. To avoid this situation, Alizadeh et al. propose DCTCP [17], a new congestion control algorithm for TCP that utilizes ECN bit to propagate congestion notifications to endpoints as early as possible.

Data center networks have a very different workload than the Internet. These networks are mainly characterized by their shorter messages and latency-sensitive workload. However, due to queuing delay, TCP-based protocols perform badly when there is a high rate of small messages in their workload [60], including DCTCP and others [34, 42]. To amend this problem, Montazeri et al. present Homa [60]. Homa is a very different protocol to TCP. It is a connectionless transport layer protocol with receiver-driven rate control, as opposed to the sender-driven TCP. Additionally, this protocol contains a priority system for received packets, prioritizing the shorter messages to avoid high delay for them.

RoCEv2: with the success of TCP, naturally, this protocol was originally used in different network architectures, such as data center networks [25]. Nevertheless, TCP presents major problems when operating in data center networks [17, 39, 60, 79, 86]. The network stack is typically implemented in the operating system, being executed by the CPU. However, executing the stack can cause significant overhead at high packet processing

speeds, which is common in data centers [39, 55]. Moreover, TCP proved to be inefficient for latency-sensitive workloads [33, 39, 50, 79], a traffic pattern commonly found in data center networks. Therefore, a new technology and protocol were proposed to address these problems.

Remote Direct Memory Access (RDMA) is a technology that allows direct access to memory on a remote system with no involvement from the CPU on the remote end, reducing overhead and latency. As such, it was a promising candidate to use for high-throughput low-latency data transfers in data centers. However, the suitable transport protocol for RDMA is very different from TCP. First, since the CPU must not be involved in this process, the network stack runs in the NIC. As such, protocols for RDMA should not be complex so that the stack can be implemented in. Moreover, the InfiniBand [65] protocol stack, where RDMA was first introduced, was designed such that its network is lossless to maintain a simple design and good performance [39]. This means that packets can still be (but rarely) dropped and it utilizes a flow control at the link layer, executed before data transmission [38, 74].

The Infiniband transport protocol needed further change to run in data center networks. This is because data center networks use the Internet protocol stack, having Ethernet and IP in the link and network layers, respectively. For this reason, RDMA over Converged Ethernet version 2 (RoCEv2) was created to allow RDMA over the Ethernet network, which uses the Internet protocol stack. Moreover, unlike Infiniband networks, the Internet protocol stack's network layer and below does not guarantee a lossless network. Originally, Priority Flow Control (PFC), a per-hop flow control mechanism, was used to add lossless to Ethernet networks. However, as highlighted by Guo et al. [86] PFC presents several problems when applied in data center networks, leading to bad performance. Thus, due to the limitations of PFC, Guo et al. [86] propose an end-to-end congestion control protocol, called Datacenter QCN (DCQCN). This protocol changes the transmission rate via congestion notifications. Similarly, Mittal et al. [58] present another congestion control protocol TIMELY, that can also be utilized in RDMA connections and decides the rate via the RTT. Meanwhile, Yuanwei et al. [54] also explore multipath in RoCEv2. Thus, all these examples show how a new network architecture might need new transport protocols; and how new protocols might need new expansions over time.

QUIC: latency-sensitive network services are becoming more popular, such as video streaming services. New extensions for older protocols, such as TCP, present possible solutions to decrease the latency in such connections [33, 50]. However, as pointed out by Langley et al. [51], the dependency on TCP can bring several difficulties in optimizing traffic for latency-sensitive applications, due to delays originating from the protocol's handshake and head-of-line blocking. Additionally, as TCP is implemented in the system's

Kernel, deploying new changes and fixes requires significant effort and long periods of time to be submitted; similarly, due to the many middleboxes present in today's network and how intrinsic they are to TCP, any change to the protocol requires updates across the middleboxes.

This motivated the creation of QUIC [51], an entirely new transport protocol that is built on top of UDP. QUIC encrypts its header to avoid middlebox interference, moves a part of its functionality to the user space (as opposed to the Kernel), and aims at decreasing the delays found in TCP. The latter is achieved by employing new handshake techniques decreasing the RTT of connection establishment, multiplexing the traffic of one connection in multiple streams without blocking and having a flow control for each of these streams, and other benefits. Finally, although QUIC is a fairly recent protocol, many works already propose new extensions and changes to it, such as multipath exploration [27, 77, 84], congestion control extension [48], and performance enhancements for specific traffic patterns [35].

Machine learning (ML) in the transport layer: the TCP algorithm and its following expansions follow a rule-based pattern. In this algorithm design, pre-defined rules dictate how the protocol will react and handle the many situations it may face. For instance, TCP's basic congestion control contains pre-defined policies to decrease the congestion window size as duplicate acknowledgements arrive or packets are dropped, and increase gradually as the expected acknowledgements arrive [18]. However, this fixed set of rules can potentially make the protocol's performance unoptimized [83]. This is because the rules for reacting to events are fixed during protocol implementation for the network environment they are designed for; thus, they cannot adapt if the environment changes significantly, after deployed. These problems lead to a fixed algorithm that may be optimized for certain situations, but not for others.

Thus, it was necessary to have congestion control algorithms capable of adapting to its surrounding network. This led several works to experiment with using machine learning. Older works tried implementing machine learning techniques to such algorithms [81, 36]. However, as machine learning and, more specifically, deep learning became quite popular over the years, many more recent works presented new congestion control algorithms involving these techniques [53, 62, 82, 83].

Takeaways: from the examples presented above, we can conclude that the transport layer is changing and, as more recent examples imply, this is a continuous trend. The motivation behind these modifications can vary considerably, highlighting the diversity of the transport layer. TCP [30] and all its extensions [41, 46, 68, 72, 73, 80] show how older protocols are constantly evaluated and updated, to better results in new scenarios,

as well as utilize new and helpful technologies [14, 28, 29, 78]. QUIC [51] highlights how new protocols are developed to fulfill demands that cannot be easily satisfied by older ones; additionally, other works show that even a recent protocol such as QUIC can be extended [27, 48, 77, 84]. RoCEv2 [20] exemplify that new protocols are created to support great changes in the network structure, such as moving the network stack to the NIC and going from a lossless to a lossy network. Data center protocols provide insight into why new protocols must be developed for new architecture [17, 60]. Finally, the recent (and quite popular) usage of machine learning in congestion control algorithms shows how the transport layer is subject to changes as new algorithm paradigms appear [53, 62, 82, 83].

1.2 The challenges of implementing and understanding transport protocols

As discussed before, the transport layer is responsible for providing numerous essential services to the network applications. These services include segmentation and reassembly, congestion and flow control, loss detection, among others. However, in addition to the sheer complexity of their operations, other factors make it challenging to implement and understand transport protocols. Thus, this section will present each of these points.

Understanding low-level details of targets: While developing a transport layer protocol, the developer must understand the target on which the algorithm will run, specially many of the low-level characteristics of the target. These low-level details can include: special built-in functions, that can execute significant operations for transport programming; data structures that are optimized for various kinds of operations; and the computational limitations that each target can have. Nevertheless, identifying these low-level details and how to make the most of them can take considerable effort and time from programmers. For instance, XDP presents unique helper functions that facilitate packet processing and a special set of stateful data structures (eBPF maps); however, as their usage may add overhead, the programmer needs to understand the trade-offs in using them in depth to be able to properly implement their desired transport protocol.

Interactions with neighbouring layers: In the network stack, the transport layer is surrounded by the application (above) and the network (below) layers. Transport protocols and algorithms are deployed in a wide range of network environments, some containing specific ways to interact with their neighbouring layers. For instance, in the Linux network stack, the sender and receiver must utilize *send* and *read* syscalls for the application to interact with the socket and transport layer [23]; meanwhile, the Infiniband architecture

[65] specifies a set of verbs used by the application to communicate with the transport layer's Queue Pair (QP). Moreover, when developing a transport layer algorithm, the developer must handle how the application requests data to be transmitted, how data is transmitted back to the application, buffer management, data copy, and other operations. Therefore, when implementing transport protocols, developers must spend a non-negligible time handling these interactions.

Understanding the protocol operations: Transport layer protocols are implemented as large pieces of optimized code that are highly difficult to navigate. As such, understanding what each of its operations does can be a challenging task. This leads to an even higher challenge in implementing new protocols or changing older ones. Currently, there are two main options available for understanding these algorithms: reading its documentation or its implementation.

Regarding the first option, when talking about transport algorithms (and the Internet as a whole) RFCs are usually the best source of information for the algorithm's implementation. They are documents written in natural language (more specifically English) and, sometimes, considerably long. Due to these characteristics, however, RFCs are susceptible to misinterpretations due to ambiguities and the vast complexity of some of these documents. For example, TCP's RFC [30] is 98 pages long, while two of its supplementary documentation, being TCP's congestion control [18] and retransmission timer computation [63], are 18 and 11 pages long, respectively. In total, to implement TCP just with its basic congestion control algorithms, the developer might need to read around 127 pages. Thus, although documentation is essential for understanding how transport protocols must be implemented, it is not the ideal solution.

Meanwhile, the second option (i.e. reading the protocol's implementation) may sound natural, as it shows exactly all the steps necessary to create a protocol and how each is implemented. However, these implementations can take thousands of lines of code and hundreds of functions. Additionally, they can be even more complex to navigate since they can be written in low-level language. For instance, Table 1.1 presents the number of lines of code taken to write some of the files that implement TCP in the Linux Kernel, written in low-level C. In addition to the number of lines, which adds up to over 21 thousand, we can highlight the intricate data structures for packet processing and socket interface, for communication with other layers of the network stack.

Name	Function	Number of loc
tcp.c	Main TCP file	4910
tcp_input.c	Input packets	7258
tcp_output.c	Output packets	3619
tcp_ipv4.c	IPv4 related functions	3666
tcp_timer.c	Timeout operations	858
tcp_cong.c	Congestion control	527

Table 1.1: Number of lines of code in the Linux Kernel for TCP code

1.3 Why high-level modular transport programming?

Section 1.1 presented the motivation behind the various changes that happened and are still ongoing in the transport layer. Meanwhile, Section 1.2 showed the challenges of working with the transport layer, both in terms of implementation and understandability. Now, we present how transport layer programming can be improved by leveraging the common characteristics of the layer, having a target-agnostic representation, and simplifying its challenging aspects. Additionally, we briefly explore the framework proposed by this thesis, MTP, that aims at reaching all these improvements.

Common characteristics: Transport layer protocols and algorithms present similar characteristics regarding their organization and functionalities. These protocols are commonly asynchronous and have an event-driven nature. Additionally, as the several examples from Section 1.1 show, the services provided by transport protocols are similar, usually revolving around congestion and flow control, segmentation, among others. These common services also may utilize similar tools, such as timers and sliding windows. Therefore, transport programming can be improved by providing abstractions for elements commonly found in this layer.

Target-agnostic representation: When developing a transport protocol, the target on which the code is executed directly impacts how the code is represented. For instance, XDP prohibits dynamic memory allocation for data structures and unbounded loops and limits its programs due to its small call stack. If a protocol needs to be deployed in different targets, this algorithm must be (usually) adapted to match the syntax and constructs present in each target. This significantly reduces the portability and reusability of code. Thus, transport programming can be enhanced by allowing the declaration of algorithms in a single representation format, that can be executed in multiple targets.

How abstraction can help: The points brought up by Section 1.2 can be split into two groups based on how to handle them. The first two challenges, i.e. low-level target

details and neighbouring layer interaction, can be in the same group. This is because they can be simplified by enhancing the abstraction of transport programming. With the target-agnostic representation proposed above, we can have built-in functions and constructs that abstract the unique low-level details of the targets. Similarly, neighbouring layer interaction can be simplified by providing interfaces whose internals are abstracted away. These solutions allow the developer to avoid spending effort finding the target's optimal constructs and handling interactions with other layers; thus, they focus solely on the transport protocol implementation itself.

The third challenge of Section 1.2, i.e. analyzing transport protocols, can be simplified by enhancing the modularization of transport programming. As already mentioned, the transport layer is event-driven. As such, the different types of events must be processed differently from each other and each type must follow a pre-defined execution path. Moreover, if different event types have similar steps during their processing, the section of code that defines these steps can be reused between their execution paths. Thus, modularization can enhance transport programming in two ways. First, by allowing developers to precisely define the execution path of events; simplifying the process of analyzing transport protocols, since each event has a clear set of steps (i.e. functions) to process it. Second, by fostering code reusability, since processing steps can be shared between different events.

Modular Transport Programming (MTP): Given the points presented above, we can introduce MTP. MTP is a programming framework that provides a high-level specification of the transport layer; while still allowing a precise declaration of its services and functionalities. The imperative domain-specific language in MTP is equipped with a set of constructs aiming to abstract this layer's complexity, such as data structures, events, timers, and sliding windows, all valuable constructs for transport protocol specification. MTP's language is event-driven and modular, in which events are dispatched to a set of modular event processors that can be reutilized and rearranged as needed. Additionally, the language requires an explicit declaration of the chain of event processors to which an event should be dispatched, leading to easier navigation and evolution of the code.

Moreover, MTP includes a set of interfaces, which handle the interaction with the neighbouring layers. To simplify programming, those interfaces have their internals abstracted out. Similarly, MTP contains configurable abstract scheduling modules, used in contention points, e.g. defining the order in which events should be processed or packets to sent out.

Finally, the MTP framework provides a compiler from which MTP programs can be translated to various targets. Thus, rendering MTP programs target-agnostic. The first target of MTP, which is the focus of this thesis, will be presented in the next section.

1.4 eBPF/XDP as the target

To explore the feasibility and benefits of MTP, we decided to use it to implement transport protocols on a specific target. For that, it was necessary to decide the target to which the MTP programs would be compiled. At first, the Linux Kernel seemed to be a good option, due to the network stack being usually implemented in it. Nevertheless, due to the complexity already pointed out in Section 1.2, prototyping in the Kernel network stack is challenging. So, as our first step to explore the feasibility of MTP, we decided to use a target that allowed us to extend the functionality of the network stack, without directly changing Kernel code. Thus, we chose to use eBPF [6] as our first target.

eBPF is a tool in the Linux Kernel that allows user-defined programs (eBPF programs) to run in the Kernel context. These programs are attached to "hooks" within the Kernel, including syscalls, the network stack, and other locations. So, this implies that the program can run in privileged mode, while the developer does not modify the Kernel code. eBPF also provides a verifier capable of identifying whether the execution of an eBPF can lead to any unsafe scenarios. This includes accessing out-of-bound memory, utilizing unbounded loops, or trespassing the complexity limit of eBPF (the stack). Therefore, eBPF allows users to write programs that can be easily executed in Kernel space, while still guaranteeing the system's safety.

An eBPF program can be one of several types, each targeting a different objective and being attached to a distinct hook. Among these types, eXpress Data Path (XDP) [43] is well-suited for this work for multiple reasons. First, programs of this type are attached to the network device driver and called when an ingress packet arrives on the device. This is an ideal place for us, since it allows the transport algorithm to act over the newly received packet. Second, XDP allows Kernel bypass. Since our goal is for the user-defined eBPF program to replace the transport layer from the Kernel's network stack, bypassing the Kernel stack is essential. Third, Høiland-Jørgensen et al. [43], shows that XDP does not fall much behind in terms of performance when compared to DPDK [32] (a popular toolkit for packet processing in user space solely with Kernel bypassing). Moreover, other works [57, 15, 67, 85, 37, 47] use XDP for accelerating packet processing. So, this tool is becoming increasingly relevant as a packet processing data path; thus, having a transport layer there can be useful.

The XDP back-end for MTP: With XDP as the chosen target to compile MTP programs into, we developed an XDP back-end that represents an XDP code that can execute MTP programs. Moreover, it follows the modularized organization and event-driven nature of MTP. This back-end code comprises of two programs: a user space code written

in C language and an XDP program. The former is responsible for loading the XDP program, initializing data structures, implementing some of the interfaces with neighbouring layers, creating and sending packets out, and triggering the execution of idle flows. Meanwhile, the XDP program is responsible for parsing incoming packets, processing events, and redirecting the received packet to user space.

While developing the back-end and exploring the tools available in XDP, we made several decisions based on XDP's characteristics and elements. This includes understanding the usage and applicability of the built-in functions and data structures available in XDP, making optimizations for the back-end's performance, and handling some of its limitations. For example, while XDP has stateful data structures called maps, there are many kinds of them and each has its own uses and limitations. During the development of XDP back-end we had to take that into account to decide which one would suit each situation better.

Compiler from MTP to the XDP back-end: The XDP back-end described above provides an infrastructure for high-speed processing. To execute MTP programs on it, a compiler needs to plug in event descriptions and processing functions and integrate them to the XDP framework. Our research group developed a compiler based on Antlr [3] to perform lexing and parsing over programs written in MTP's language. This compiler generates an Abstract Syntax Tree (AST) and visitors to traverse its nodes. Moreover, through these node visitors, we created a code generator that translates MTP code into code that can be plugged into and run as. The translation part of the XDP back-end from one language to another is considerably simple since the user-defined parts of the code are written in a C-like language, at both MTP and XDP sides. However, certain details were considerably more complex to implement, such as loops and out-of-bounds memory access checks.

1.5 Related Work

Many network programming frameworks have been proposed and released in recent years. Additionally, few works have enhanced the programmability of XDP and extended TCP in XDP.

Stateless packet processing: Numerous works implement high-level languages for stateless packet processing [31, 59, 70]. Their languages can represent services such as packet-forwarding policies, and classification and aggregation of traffic of SDN architectures. These languages focus on lower levels of the network stack (below the transport layer) and are packet-driven. Meanwhile, MTP focuses on the transport layer programma-

bility; whose protocols have considerably different characteristics, being event-driven and stateful.

Stateful packet processing: Other works provides programming abstractions for stateful packet processing [19, 21, 66, 56]. These works were also designed for programming lower levels of the stack on programmable switches. Moreover, the hardware they execute on contains several limitations, such as reading and writing to memory and the computation between these operations. Meanwhile, since transport protocols usually execute in the end-points of a connection, they must not handle these restrictions. Due to this reason, MTP supports constructs and abstractions that are unusual for these frameworks, such as loops, timers, data segmentation, and sliding windows.

Programmability in the transport layer: Congestion Control Plane (CCP) [61] consists of an off-datapath module that separates the congestion control from the transport layer. The main goals of CCP are the reusability of the same algorithm across different transport protocols and providing higher abstraction for congestion control programming. Therefore, while CCP focuses solely on congestion control algorithms, MTP enables the declaration of the entire transport layer.

Simplifying eBPF programming: Although it is written in C language, programming in eBPF can be a challenging task. This originates from the several processes required to load the eBPF program, the special declaration and usage of eBPF maps, its limitations, and others. However, many works facilitate eBPF programming by providing a high level of abstraction [9, 11]. BCC [5] is a toolkit and library that abstracts complex processes (e.g. loading eBPF program and interacting with maps) through high-level programs written in Python and Lua. However, BCC main focus is Kernel tracing. Meanwhile, the XDP back-end aims at simplifying transport layer programming in eBPF/XDP.

Extending TCP in eBPF: A few works implement extensions to the TCP protocol of the Linux Kernel network stack [22, 76]. TPC [45] extends TCP by analyzing the round-trip-time and making rerouting decisions as failures are detected on flows. Moreover, instead of simply using eBPF/XDP as a way to extend TCP in the Linux Kernel, the XDP back-end presented in this thesis allows the declaration of an entire transport protocol in XDP.

1.6 Contributions

This thesis aims to explore the programmability of the transport layer in XDP. To address the points discussed in Section 1.3, this work focused on enhancing the abstraction and

modularity of transport programming. We chose to utilize XDP because it allows a simpler and safer declaration of programs in the Kernel. Moreover, as we explored XDP, we faced several unique characteristics of eBPF/XDP and handled many of the constraints of this tool.

Modular Transport Programming (Chapter 2): MTP is a programming framework which targets simplifying the programmability of the transport layer. First, it provides an abstract view of the transport layer, making the developer oblivious to low-level details. These details include boundary interactions, packet generation, and scheduling. Moreover, it enhances abstraction by bestowing a set of constructs designed specifically for the construction of transport protocols. Second, it enforces the modularization of transport layer algorithms. It does so by allowing the declaration event processing modules that can be rearranged and reutilized for different events. Two simplifications of transport protocols, TCP and RoCEv2, were implemented in MTP's language to evaluate the expressiveness of MTP when covering protocols with significantly different characteristics.

XDP Back-End (Chapter 3): We explored the transport layer programming by creating an XDP back-end that follows MTP's model. This back-end implementation contains two main programs. First, a user space program is responsible for loading the XDP program, triggering the execution of idle flows, and performing most of the interactions with other layers. Second, an eBPF/XDP program that parses network packets, processes events, and redirects the received packets to user space. During the back-end's development, we made numerous design decisions to optimize its performance and handle the limitations of XDP.

XDP Code Generator (Chapter 4): A code generator was developed for this thesis using the AST visitors generated by our research group's compiler. This code generator creates a pair of user space and XDP code based on the inputted MTP program. Moreover, we were able to compile the TCP and RoCEv2 MTP implementations into the XDP back-end.

Chapter 2

Modular Transport Programming

Working with the transport layer of the network can be a challenging task. As presented in Section 1.2, handling the interaction with neighbouring layers and low-level target details complicates its programmability. Additionally, understanding and analyzing transport implementations can be difficult in code implementation and documentation. Meanwhile, Section 1.3 presents how transport layer programming can be improved, by enhancing abstraction and modularization. Thus, to simplify the development of the transport layer's algorithms, this work presents Modular Transport Programming (MTP). This programming framework provides a higher level of abstraction and enhanced modularization. These two characteristics aim to abstract low-level details, boundary interactions, and other difficulties from the developers. Moreover, it also allows the declaration of more readable code, based on modularization.

The MTP programming framework was developed based on a logical model developed by our research group. This model comprises a set of linked blocks that vary in levels of programmability. This model includes interfaces that communicate with the neighbouring layers and modules responsible for processing events. To represent the interfaces and modules, MTP provides a language that allows the declaration of the programmable blocks. Thus, this chapter will explore MTP by presenting the logical model that inspired it, followed by how MTP's language represents it.

2.1 MTP logical model

The transport layer is surrounded by other two layers, application and network. As such, it is responsible for moderating their communication between them. On one endpoint of

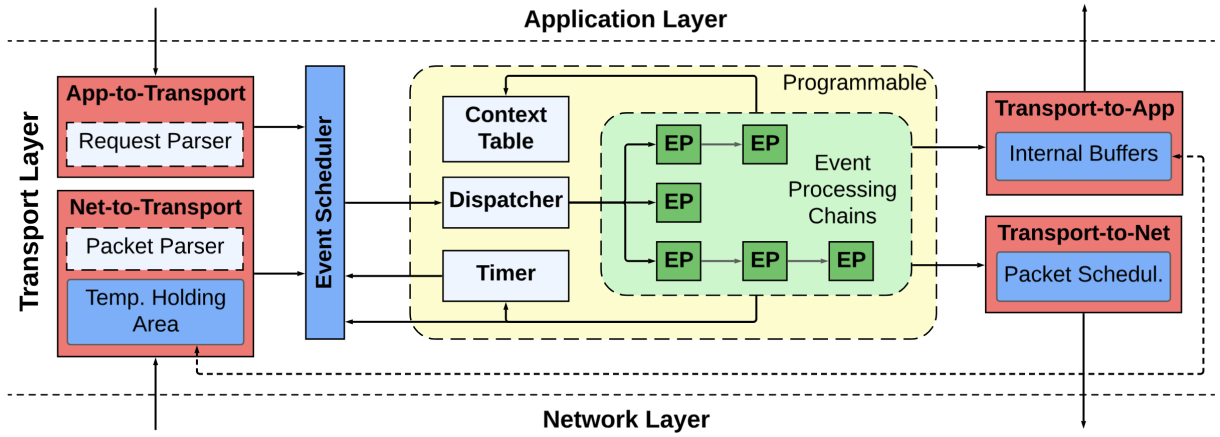


Figure 2.1: MTP's logical model of the transport layer.

a connection, the application can submit requests for the transport layer, which leads to the creation of packets. During this, the transport algorithms decide how the packets are created (how much data to send and the transport header values) and how many must be generated, based on feedback from the other endpoint. Then, these packets are dispatched to the network layer for transmission to the other endpoint of the connection. Meanwhile, the other endpoint receives the transmitted packet. On this side, the transport layer protocol can perform services such as reassembling the arriving data, as well as sending feedback.

MTP represents the transport layer through a logical model. This model contains interfaces that handle communication with the neighbouring layers and other modules for event processing. Due to MTP's event-driven nature, both application requests and network packets are parsed into events. To process these events, MTP's logical model provides modules responsible for several steps of event processing. Figure 2.1 presents the logical model of MTP. In this figure, the interfaces are represented by the left and right-most red blocks, while the event processing-related modules are found between the interfaces.

Before delving into further details of each module, we must summarize how MTP's logical model operates. An event can be parsed from an application request or a packet from the network layer. Then, this event is enqueued and waits for the scheduling policy to decide the event to be processed. Moreover, the chosen one is sent to the dispatcher, which dictates the chain of event processors that should execute the event. During its processing, new events can be generated, timers can be initialized, and the context table (containing stateful information) can be accessed. Finally, the event's processing can result

in transmitting received data to the application and/or sending new packets to the network layer.

Thus, this section will present the modules that compose the MTP model.

2.1.1 Event-related modules

Events are the main concept of MTP's model. They define the many possible operations that a transport protocol implementation will perform. Event types, also called minor types, can be declared by the developer. Moreover, MTP classifies minor events into four major types. Depending on their origin, they can be of major type: *application event*, being parsed from application requests or a notification to send data to the application; *network event*, being parsed from packets from the network, new packets to be sent out, or to change the sending rate; *timer event*, generated when timers (set during event processing) reach their timeout; and *program event*, those generated immediately during event processing. Moreover, application and network events can be classified as *incoming* or *outgoing*. This depends on whether the event originates from the incoming interface or should be sent to the outgoing interface.

Incoming events of these four major types are received from their starting points and are passed across a series of modules for processing:

Event scheduler: Scheduling policies may contribute to an even distribution of the system's computational resources, avoiding contention and starvation. Additionally, having control over these policies can be desirable to prioritize certain characteristics of the scheduled environment. For example, prioritizing some types of events over others. Since a transport protocol must handle several flows concurrently, MTP's event scheduler module allows the definition of scheduling policies in two levels: flow and major event type. After a new event is generated, it is enqueued to a specific queue based on its flow and major type. Then, it is only dequeued after the scheduling policy decides it should be processed next. However, the event scheduler may have single-level scheduling in certain MTP targets (flow or major event type), due to the limitations of each target.

Dispatcher: After the scheduling policy decides the next event to be processed, the event is transferred to the dispatcher. The dispatcher's main responsibility is to map minor event types to the modules (event processors) that execute them. When declaring each mapping, MTP allows the chaining of event processors. This means that, after one processor finishes executing the event, the next in the chain continues processing it. Thus, the dispatcher requires an explicit declaration of the steps taken to execute each type

of event. This promotes modularization and reuse of event processors across different mappings, improving code readability.

Event Processor (EP): This module consists of a programmable block that describes how an event is executed. An event processor takes as parameters: the event and its flow's context table, and a piece of stateless information (scratchpad) passed across chained event processors. While it executes, an event processor can generate new events (program events) and initialize timers, which can lead to the creation of timer events after their timeout. Additionally, it may produce *outgoing* events, which are handled by the outgoing interfaces. This includes events representing packets to be sent out, a new sending rate, among others. After the module finishes executing, both the context table and scratchpad are updated.

Context table: Transport layer protocols may require access to stateful memory to read and write information necessary to its services. For example, in TCP an endpoint must keep track of the next sequence number to be sent, the last unacknowledged byte, and other values; moreover, all these values can be stored in a context table. Thus, the context table serves this purpose, allowing the event processors to access and update their stateful memory. Additionally, it is worth noting that each flow has a separate instance of the context table.

2.1.2 Incoming Interfaces

MTP has a pair of interfaces that handle incoming data from the neighbouring layers. Requests are received from the application, while packets are received from the network layer. Both received data must be parsed into events, which are later processed by the event-related modules. These two interfaces are represented as the left-most red boxes in Figure 2.1.

Application-to-Transport: This interface receives application requests, representing operations an application requires the transport layer to perform. This includes sending data to another endpoint, requesting data, among others. MTP allows the declaration of requests (i.e. the fields that compose it) and the application request parser. This module receives requests and parses them to create application events. The minor type of the event is defined through one of the fields of the request, similar to an opcode. Finally, the events are transmitted to the event scheduler, where they are enqueued to the application queue of their flow.

Network-to-Transport: The network layer passes packets to the transport layer. Through MTP, it is possible to declare the packet parser. This parser extracts the packets' transport headers, parses them and creates network events. This approach follows a similar

design to already existing packet processing languages such as P4 [10]. Then, the newly created events are transmitted to the event scheduler, where they are enqueued to the network queue of their flow.

Moreover, arriving packets may carry a payload, which must be transmitted to the application. Since transport layer operations work with information about the payload (i.e. length, offset, address) but not the payload itself, carrying it across the various modules that process the packet's network event is unnecessary. To store the payload, the Network-to-Transport interface has access to a *temporary holding area*. This is a space of memory where the payload carried by the arriving packet is temporarily stored, while the event is processed.

While the Network-to-Transport interface writes to the holding area, the Transport-to-Application (Section 2.1.3) reads from it and stores the data in its *internal buffers*. However, since the internal buffers of the Transport-to-Application interface can become full, the Network-to-Transport monitors the internal buffers' available space. Moreover, the Network-to-Transport can pass the available space to events, which can be utilized for congestion control decisions. If the internal buffers are full, the interface can also drop events.

MTP has a built-in method (called *extract*) that stores the payload in the temporary holding area, among the other operations it does. This method also retrieves the address where the payload is stored. Moreover, this address can be passed to events so the Transport-to-Application interface can use it to access the payload. Going back to one of the two major goals of MTP (enhancing abstraction), this method allows the developer to interact with the temporary storage while abstracting how the memory is organized and works.

2.1.3 Outgoing Interfaces

After an event is processed, two main outcomes are possible: transmitting received data to the application or sending packets to the network layer. As such, MTP's model contains two interfaces that interact with the neighbouring layer for these purposes. The two interfaces are represented as the right-most red blocks in Figure 2.1.

Transport-to-Application: As explained in Section 2.1.2, packets can carry a payload, which must be transmitted to the application. This data is stored in a temporary holding area and, at later stages of MTP's model, is accessed by the Transport-to-Application interface. This interface reads the holding area at a given address, reassembles it with other previously read data, and, possibly, transmits the result to the application.

Instead of transmitting the received data immediately after reading from the holding area, protocols can define rules for when the data should be "flushed" to the application. For example, TCP utilizes *socket.read* to access the socket and write its data to a buffer, and RoCEv2 waits for Work Queue Element (WQE) to arrive in the Completion Queue (CQ). Moreover, the Transport-to-Application interface contains a set of *internal buffers* to store and incrementally reassemble the data. These buffers accumulate the data read from the temporary holding area for individual traffic streams (as in TCP and QUIC) or WQE (as in RoCEv2). Finally, the data is flushed to the application only when the transport protocol implementation notifies so.

During event processing, an Event Processor (EP) can instantiate *DATA_EVENT outgoing* application events. This special type of event signals the Transport-to-Application interface to read from the temporary holding area. Additionally, it specifies how the interface should store the data in its internal buffers. Events of this type specify: the address at the holding area that contains the data; the total length of the data; the buffer ID, which is a unique identifier for each of the internal buffers; and an offset, to define where the retrieved data is written in the chosen buffer. Therefore, how the internal buffers are managed, the data is transferred between the holding area and buffers, and other details are target-specific and abstracted from the programmers.

Although the target usually decides when the data must be transferred to the application, MTP provides a way for the protocol to define when the transfer can happen. When the condition to flush the data to the application is met, EPs can instantiate a special *outgoing* application event, called *APP_NOTIF_EVENT*, signalling the data can be transferred. This event defines a range of data that should be flushed to the application, as well as potentially signalling the application that the request for data was completed.

On the other hand, while the condition is not met, the internal buffers keep accumulating data until they reach their maximum size. When that happens, the Transport-to-Application signalizes the Network-to-Transport interface to drop packets until the buffer has space. This drastic design decision is motivated by the context table. During event processing, the flow's context table can be updated. However, if the interface's internal buffers are full, the data cannot be retrieved from the holding area; meanwhile, the context table has already changed. Because rolling back changes can be challenging, we chose to drop packets to avoid this possibility.

Transport-to-Network: While events are processed, EPs may decide that packets should be sent out. For that, MTP provides an *outgoing* network event type, called *PKT_EVENT*. This event specifies the information necessary to create a packet, such as header fields, the address and the offset to retrieve the payload, among other values. Addi-

tionally, if requested by the event processor, this interface can perform packet segmentation.

In addition to creating packets, the Transport-to-Network interface may allow the developer to control how the packets are scheduled and the rate they are transmitted. To achieve a more balanced transmission of packets, MTP may provide a *packet scheduler* for flow scheduling policy, depending on the target. Additionally, event processors may generate a special *outgoing* network event, called *RATE_EVENT*. This event provides feedback for the interface to update the per-flow packet transmission rates.

2.2 MTP Language

MTP provides a language to represent the logical model presented in Section 2.1. This language was designed to closely resemble a restricted C++, bringing its syntax and some of its usual constructs, such as conditions, for loops, function declarations, and structures. MTP's language allows the declaration of the logical model's programmable modules as separate blocks. These blocks are only linked during the compilation phase, according to the model's specification. Thus, this section presents how the main constructs of MTP are declared through an example of a TCP implementation.

One of the first steps in writing an MTP program is to define the protocol-specific types that the transport protocol will utilize. To do so, MTP provides a set of special structures that can describe the format of those types. In addition to the default C++ structures, they include events (the major concept of MTP's event-driven design), the context table, and others. Although they all share a style of declaration inherited from C++, each presents some unique characteristics and objectives that are important to highlight.

Event: MTP allows the declaration of protocol-specific event types through the keyword *event*, followed by a unique identifier. These types are considered "minor" since MTP classifies them into major groups depending on their origins: APP_EVENT if it derives from application requests; NET_EVENT, originating from network packets; TIMER_EVENT if it is created from a timer; and PROG_EVENT if it is created during event processing. Additionally, application and network events can be categorized as *incoming* or *outgoing*, depending on the interface they interact with. In the example below, the declaration of an event named ACK represents acknowledgement packets arriving from the network, expressed through $INCOMING :: NET_EVENT$.

```
event ACK : INCOMING::NET_EVENT {
    uint32 ack_seq;
    uint32 rwnd_size;
```

```
}
```

Other two details regarding events must be pointed out. First, the built-in type *event_t* represents all minor events. This type is used in many blocks to refer to events in a generic way. Second, all event types contain, by default, a flow ID field, which uniquely identifies the event's flow. This value is filled during application request and packet parsing.

Context: Event processors can access the event's flow context table entry. This module holds the stateful information used in the transport algorithms. To declare the context table, MTP provides the keyword *context*, followed by its identifier and fields, as shown in the example below. Note that the context is where instances of some of MTP's special built-in types, such as timers, sliding windows, and buffer identifiers, are declared.

```
context tcp_context {
    uint32 last_ack = 429496729;
    uint32 send_una = 0;
    uint32 send_next = 0;
    uint8 duplicate_acks = 0;
    ...
    uint32 recv_next = 0;
    timer_t ack_timeout;
    sliding_wnd meta_rwnd;
    buffer_id_t bid;
    ...
}
```

Scratchpad: During the execution of an event, it might be beneficial to have a way to share information only between event processors of the same chain. This could potentially be achieved through the context table. However, since its values can be accessed by all EPs, the developer would have to control where each context field is accessed. To simplify this aspect of MTP programming, a structure named scratchpad was introduced. This construct declares stateless metadata shared only across EPs of the same chain. Moreover, the keyword *scratchpad_t* is utilized to define this structure, following a C++ structure-like declaration.

Packet blueprint: This structure type describes the exact and entire configuration of a packet. It includes transport header values and even payload. As this section will present later, fields of certain special built-in types (e.g. *checksum16_t* and *data_t*) cannot have values assigned during event processing. Instead, MTP provides built-in helper functions that abstract these processes. These functions allow the developer to specify how the Transport-to-Network should these special fields when creating packets.

```
pkt_bp TCPBP{
    bit<16> srcport;
    bit<16> dstport;
    bit<32> seq_no;
    bit<32> ack_no;
    bit<16> urgent_ptr;
    ...
    checksum16_t checksum;
    data_t data;
}
```

After the protocol-specific types are defined in MTP, the next step is to declare the parsing interfaces and event processing-related modules that use the defined types. While developing MTP, we chose to provide different levels of programmability for each of these constructs to allow a wide range of possibilities on where it matters the most (i.e. event processing). Meanwhile, MTP provides a higher level of abstraction on parts where having a complex syntax would not be as significant. For example, the dispatcher follows a strict (but high-level and effective) syntax; while other modules, such as event processors, are fully programmable.

Application request parser: When the application submits a request, the Application-to-Transport interface receives it. This interface's main task is to parse the request into an application event. As such, MTP contains a construct that declares the interface through the keyword *app_to_transp*. This construct contains the declaration of the request's header and the parser function declarations, presented in Figure 2.2. Although headers are usually elements found in packets, MTP specifies that application requests are to be interpreted in this header format. Moreover, this special header type is declared through the keyword *app_header* and follows the same syntax as other structure-like declarations.

Meanwhile, a parser function declaration can be seen in line 5. This function receives as an argument the application request and returns a list of events created after parsing it. Moreover, the function starts by instantiating the application header and extracting it from the request (lines 7 and 8). The next step is creating the events' flow identifier. This step requires the new ID to be instantiated with the values that must be hashed (line 9). Since this example consists of a TCP implementation, the 4-tuple is hashed. Moreover, a field from the requests specifies the expected minor type of event to be created from it. This value is compared in a series of conditions (in this TCP example, only one, since there is a single minor application type). Finally, inside the minor type condition, the event is instantiated, filled with its values, including the flow ID (line 13), and added to the list of outputted events.

```
1  app_to_transp att_module {
2      app_header APPheader {
3          ...
4      }
5      list<event_t> parse_app_request(app_req_t req) {
6          list<event_t> out;
7          APPheader r;
8          req.extract(r);
9          flow_id id(r.src_addr, r.dst_addr, r.src_port, r.dest_port);
10         if(req.type == SEND) {
11             SEND event;
12             event.data_size = r.data_size;
13             set_fid(event, id);
14             out.add(ev);
15         }
16         return out;
17     }
18 }
```

Figure 2.2: Example of a Application-to-Transport interface declaration.

Network packet parser: As packets arrive from the network, the Network-to-Transport interface is responsible for intercepting and parsing each into network events. To do so, this interface is declared through the keyword *net_to_transp*, as shown in Figure 2.3. The interface contains the declaration of the network header (in this case, IP) and the packet parser. The network header is declared through the keyword *net_header* and follows the same syntax as other structure-like declarations.

Moreover, the parser function can be seen declared at line 5. This parser takes the packet and the network header as arguments and returns the list of new events. This function behaves similarly to the one presented before, where a header (in this case, the TCP header) is extracted from the packet, followed by the creation of its flow ID based on the 4-tuple. Instead of having a field specifying the event type that should be generated, this parser utilized values from the packet to make this decision. For example, a *DATA_EVENT* is created in case the packet carries a payload (lines 10-17) and an *ACK_EVENT* is generated if the arriving packet is an acknowledgement (lines 18-23).

As mentioned in Section 2.1.2, if a packet carries a payload, the Network-to-Transport interface must store this data in a temporary holding area. To represent this concept in MTP, the packet's information is extracted into a TCP packet blueprint, shown in the line 8. When this blueprint is extracted, its data portion is automatically stored in the temporary holding area. Moreover, after assigning the blueprint to the newly created event field, both the payload's length and the holding address can be retrieved from the packet

```
1  net_to_transp ntt_module {
2      net_header IPheader {
3          ...
4      }
5      list<event_t> parse_net_packet(pkt_t p, IPBP i) {
6          list<event_t> out;
7          TCPBP pkt_bp;
8          p.extract(pkt_bp);
9          flow_id id(i.src_addr, i.dst_addr, p.src_port, p.dest_port);
10         if(len(pkt_bp.data) > 0) {
11             DATA event;
12             ...
13             event.hold_addr = pkt_bp.data.addr;
14             event.data_len = len(pkt_bp.data);
15             set_flow_id(event, id);
16             out.add(event);
17         }
18         if(t.ack == 1) {
19             ACK event;
20             ...
21             set_flow_id(event, id);
22             events.add(ack_ev);
23         }
24         return out;
25     }
26 }
```

Figure 2.3: Example of a Network-to-Transport interface declaration.

blueprint's data. Finally, these values can be copied to some of the event's fields (lines 13 and 14), to be later used by the Transport-to-Application interface.

Dispatcher: During the development of MTP's language, we had to analyze how complex the dispatcher module should be. Having imperative language constructs, such as conditions, loops, and assignments, would be a natural choice. This design presented benefits since it allowed the developer to specify a more sophisticated decision algorithm to dispatch events. However, since the objective of MTP is to simplify transport programming, this approach would not provide the desired abstraction.

As examples of transport layer algorithms were developed in MTP, we noticed that the steps of event processing could be represented as disjointed services. For example, TCP may handle arriving acknowledgments by updating RTO, performing fast retransmission and recovery, executing slow start and congestion control, and retransmitting packets or sending new ones as the window allows. Each of these tasks has its own logic and most

```
dispatch disp_table {
    SEND      -> {send_ep};
    ACK       -> {rto_ep, fast_retr_rec_ep, slows_congc_ep, ack_net_ep};
    DATA      -> {data_net_ep, send_ack, app_feedback_ep};
    MISS_ACK  -> {ack_timeout_ep};
}
```

Figure 2.4: Example of a dispatch table declaration.

of their control flow is typically within the boundaries of the task. This means that they can be disjointed, but the order of execution across tasks must still be preserved. Thus, following a simple, yet, precise definition of the dispatcher was the chosen format.

The way the dispatcher is represented is one of the good novelties of MTP to improve and simplify the readability and analysis of transport layer implementations. Although it is simple, the declaration of this module explicitly exposes how the transport algorithm should behave. This is done through a mapping between event types and the chain of event processors that executes each. Additionally, this design allows the developer to reuse and chain event processors without requiring direct changes to the EPs. Figure 2.4 shows the declaration of a dispatcher table through the keyword *dispatch*, followed by its identifier and the mapping between events and their chained processors. For example, when an ACK event arrives, it is dispatched to a chain of EPs that executes the steps for acknowledgement processing.

Event processor: This type of module is where the main logic of the transport algorithm is written. As such, the event processor is the module with the highest degree of programmability across MTP's framework. This module includes the most commonly found constructs from C++; such as variable declarations, assignments, logical and mathematical operations, conditions, and bounded loops. Additionally, MTP provides built-in helper functions that perform a plethora of useful operations focused on transport layer programming, while abstracting its complexities for the developer. For example, initializing and cancelling timers, operating over a sliding window, segmenting data, etc.

The declaration of an event processor follows a syntax similar to a C++ function declaration. This module may return two types: *void*, in case the EP does not create any new events; or a list of events, if one or more events are generated. The declaration continues with the EP's unique identifier, utilized by the dispatcher to map event types with their chain of processors. Then, the EP must receive three arguments: the event passed to be processed, whose type must match what is described in the dispatcher; the context entry for the event's flow; and the scratchpad passed across the chained processors. Moreover, if an EP is utilized to process more than one event type, MTP allows its declaration to be in a format similar to a C++ function template.

Two examples of EPs will demonstrate the declaration of an event processor and some of the useful tools available in MTP. The first example, presented in Figure 2.5 is an event processor responsible for sending the first packets of an application request. After performing a series of operations that decide the number of bytes that should be sent, line 11 instantiates a packet blueprint. Moreover, the values of this blueprint, such as its ports and sequence number, are filled accordingly in the next few lines. On line 16 the event processor utilizes a predefined checksum object. This object specifies the checksum technique and the fields used during its computation. Although this object is specified in the event processor, the actual checksum calculation is only performed in the Transport-to-Network interface, when the packet is created.

The last field of the packet blueprint that must be filled is its payload, an object of type *data_t*. Objects of this type can avoid or require segmentation. The first refers to data already segmented (or does not require), i.e. the instantiated blueprint is expected to create a single packet. *seg_data* is used to instantiate such an object. Objects of this type must be initialized with the address from which the payload should be retrieved and the number of bytes (length).

Meanwhile, the second type of object refers to currently unsegmented data that must be segmented when the packets are created in the Transport-to-Network interface. *unseg_data* is used to instantiate this object type. In addition to the two values previously mentioned (address and length), the initialization of unsegmented data objects must be provided with: the maximum segment size (MSS); the header field that changes for each packet; and how much the header field changes for each packet. The line 18 shows an example of unsegmented data. In this example, the payload to send starts from an address *addr* and is *b2s* bytes long. Additionally, each packet carries *MSS* bytes and varies its sequence number, increasing by the payload's length. Finally, a *PKT_EVENT* is instantiated and filled, later transmitted to the Transport-to-Network interface.

Transport layer algorithms commonly utilize timers to perform various tasks, such as retransmitting packets and changing transmitting rates. Due to this reason, MTP allows the declaration of timers and provides a set of methods to operate with them. Lines 23 to 27 demonstrate a common use of timers and their operations. In case *ack_timeout* is currently inactive, the timer is initialized by setting its total duration (*set_duration*). Then, a new timer event is instantiated and added to the timer, being enqueued after the timer's timeout.

Shown in Figure 2.6, the second example of an event processor presents the point-of-view of the other endpoint of a TCP connection, specifically, how it handles the receipt of data. This example highlights two relevant tools provided by MTP: the sliding window and

```
list<event_t> send_ep (SEND ev, tcp_context ctx, tcp_scratch s) {
    list<event_t> events;
    ctx.data_end = ctx.data_end + ev.data_size;
    int32 unsent = ctx.data_end - ctx.send_next;
    int32 window = min(ctx.cwnd_size, ctx.last_rwnd_size);
    int32 b2s = ctx.send_una + window - ctx.send_next;
    b2s = min(unsent, b2s);
    if(b2s < 0)
        return events;
    // Instantiating and filling packet bp
    TCPBP bp;
    bp.srcport = ctx.srcport;
    bp.dstport = ctx.dstport;
    bp.seq_no = ctx.send_next;
    ...
    bp.checksum = CRC16_t([...]);
    addr_t addr = ctx.addr + ctx.send_next - ctx.init_seq;
    bp.data = seg_data(addr, b2s, MSS, TCPBP::seq_no, len(TCPBP::data))
        ;
    PKT_EVENT pkt_ev;
    pkt_ev.bp = bp;
    events.add(pkt_ev);
    // Timer operations
    if(!ctx.ack_timeout.is_active()) {
        MISS_ACK time_ev;
        time_ev.seq_num = ctx.send_una;
        ctx.ack_timeout.set_duration(nanosecond(ctx.RTO));
        ctx.ack_timeout.start(time_ev);
    }
    return events;
}
```

Figure 2.5: Event processor to send packets from application request.

the notification of the Transport-to-Application module. The sliding window consists of a set of boolean flags mapped to a sequence of numbers. Moreover, in this implementation of TCP, this window is utilized to record the sequences of bytes received; so the event processor can know which bytes are missing in case they are dropped or arrive out-of-order. In line 7, method *set* sets as true the boolean mapped from the arriving sequence number until the payload's length, i.e. sets that section of bytes as received. Meanwhile, method *first_unset* (line 8) returns the number mapped to the first *false* boolean of the window, i.e. the next sequence number that should be received. Additionally, the window can be initialized from a certain number and slide as data is flushed to the application.

```
1  list<event_t> data_net_ep (DATA ev, tcp_context ctx, tcp_scratch s) {
2      list<event_t> events;
3      uint32 data_end = ev.seq_num + ev.data_len;
4      if((ctx.rwnd_size == 0 && ev.data_len > 0)  (ev.seq_num > ctx.
           recv_next + ctx.rwnd_size)  (data_end < ctx.recv_next))
5          return;
6      // Updating sliding window
7      ctx.meta_rwnd.set(ev.seq_num, data_end);
8      ctx.recv_next = ctx.meta_rwnd.first_unset();
9      // Notifying Transport-to-Application
10     DATA_EVENT ev;
11     ev.read_addr = ev.hold_addr;
12     ev.size = ev.data_len;
13     ev.buffer_id = ctx.bid;
14     ev.offset = ev.seq - ctx.init_seq;
15     events.add(ev);
16 }
```

Figure 2.6: Event processor to handle the arrival of data.

When a packet carrying a payload arrives, the event processor that handles its event must alert the Transport-to-Application interface regarding the arrival of data. As such, MTP provides a special event type, named *DATA_EVENT*, that carries the necessary information to notify where the data should be retrieved and where to store it. The fields of this event are: the temporary holding area address, where the data is initially stored; the payload's length; the identifier of the internal buffer that will store the data after retrieving from the holding area; and the offset to where the data is stored in the buffer. Lines 10 to 15 demonstrate the instantiation of the built-in event and the filling of its fields.

Although not shown in the examples, event processors can utilize other two relevant operations regarding the Transport-to-Application interface. First, when a new flow is established (or in QUIC's case, a new stream), an event is created and processed by an EP that calls a built-in helper function *create_buffer*. This function requests the creation of a new Transport-to-Network internal buffer, whose ID is stored in the context. Second, as mentioned in Section 2.1.3, MTP provides a special event *APP_NOTIF_EVENT*. This event specifies when data must be flushed and the application notified. These events contain data information (such as offset and length) and a condition. The latter determines which part of the buffered data must already be in the application-accessible memory before the notification is sent to the application.

Chapter 3

Exploring eXpress Data Path

After developing MTP's model and its language, we had to test whether MTP framework was valid and feasible. To accomplish that, we implemented the model in eXpress Data Path (XDP) [43], the first target of MTP. XDP is a tool that enables a simple attachment of user-defined code to the NIC driver. During this process, we explored many aspects of XDP, taking advantage of the instruments provided, understanding XDP's limitations and finding ways to approach them. This exploration resulted in a back-end based on MTP's model that can perform transport layer services and send and receive packets.

3.1 eBPF/XDP background

As previously mentioned in Section 1.4, deploying code to the Linux Kernel can be challenging and, possibly, lead to security problems. To simplify this process, we chose to utilize the extended Berkeley Packet Filter (eBPF) platform (more specifically, XDP) as the first target of MTP. eBPF introduces the concept of "hooks", which are certain points of the Kernel to which developers can attach eBPF programs. Moreover, it does not require direct changes to the system's code and runs only when an application thread or Kernel passes through a hook. These hooks include system services such as the network stack, system calls, I/O operations, among others. Meanwhile, the safety of the Kernel is assured through the eBPF verifier, a tool that analyzes whether the code meets certain criteria before attaching it to the Kernel space.

eBPF programs are represented in a limited C language. The limitations include restricting the usage of float numbers, unbounded loops, common C libraries, and other

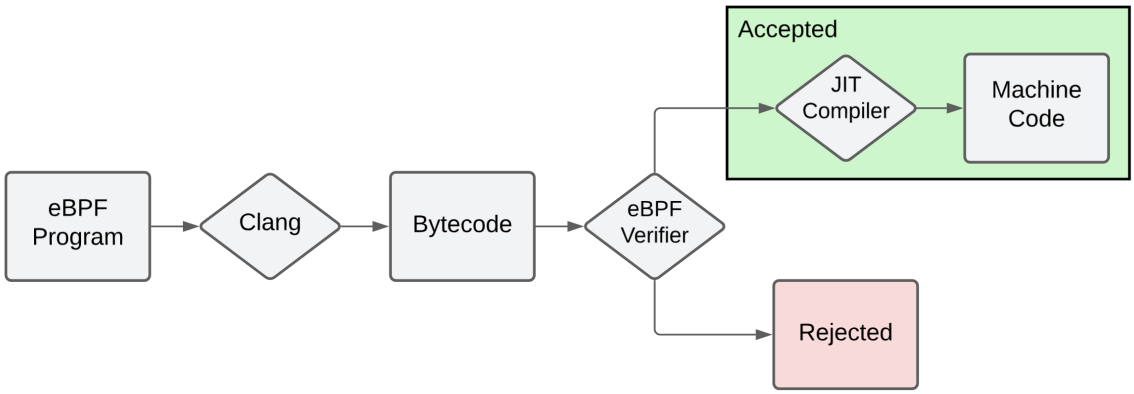


Figure 3.1: The process to load an eBPF program.

limitations. Additionally, these programs contain macros to define the type of eBPF program and to declare special data structures (eBPF maps, disclosed in Section 3.3). They also contain helper functions that provide some essential tools for eBPF programming. The Linux Kernel [75] provides the *bpf* library that has these elements.

An eBPF program undergoes several stages between its development and its attachment to the Linux Kernel. First, the Clang compiler from LLVM [52] converts the restricted C code into eBPF bytecode. Then, through a BPF system call, a user space process triggers the eBPF verifier to analyze whether the program meets all safety requirements. These requirements include in-bound memory access, bounded loops, and the prohibition of any set instructions that may cause crashes or potential harm to the system [8]. If all requirements are met, the bytecode is translated to machine-specific code using a Just-in-Time (JIT) compiler, and attached to whatever hook the eBPF program type utilizes. Meanwhile, if the requirements are unsatisfied, the program is rejected and cannot be attached to the hook. Figure 3.1 summarizes the stages an eBPF program takes before it is loaded to a Kernel hook.

eBPF has several different types of programs that can be developed with it. The most important difference between them is the location of their hooks, i.e. where they are loaded. Among the many options of eBPF programs available, eXpress Data Path (XDP) [43] was shown to be the best option for this project. Programs of type XDP are attached to a hook localized in the NIC driver. This means that the XDP code is executed for each incoming packet from the network (packet-driven nature). Additionally, XDP programs can redirect packets directly to the user space, bypassing the Kernel network stack. These two characteristics were decisive for this work. Because our objective with MTP is to implement the transport layer, the localization of XDP's hook is perfect for processing newly arrived

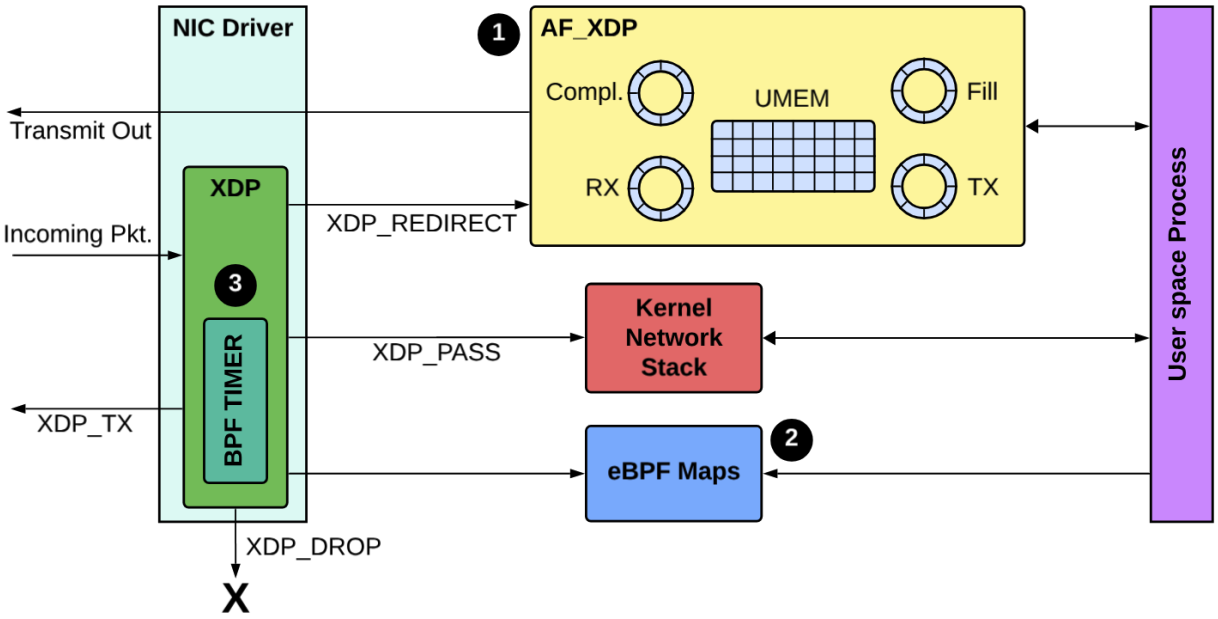


Figure 3.2: Summary of XDP's architecture.

packets. Moreover, bypassing the network stack avoids the unnecessary overhead that the stack would generate, as the code generated from MTP already implements the transport layer.

Figure 3.2 represents a summary of the XDP architecture described in this section. When a packet arrives at the NIC's driver, the XDP program is triggered to execute in a Kernel thread. XDP can operate over the packet in various ways. For example, it can parse and mutate the packet, retrieve and store information to stateful data structures (eBPF maps), and execute user-defined functions to process the packet's information. XDP also contains a set of return codes. These return codes specify the packet's destiny after the program finishes its execution. They can be: **XDP_DROP**, the packet is dropped, i.e. no further processing is done; **XDP_PASS**, the packet is passed to the Linux Kernel network stack; **XDP_TX**, it transmits the packet out through the same network interface it came from (one packet in, one packet out design); **XDP_REDIRECT**, the packet can be redirected to either another NIC (including virtual), another CPU, or user space. The latter (XDP_REDIRECT) is the most important return code for this work since it allows kernel network stack bypassing through a special socket, Address Family XDP (AF_XDP) [49]. This socket redirects packets directly to the user space.

Bypassing the kernel network stack requires the usage of AF_XDP [49] (also known

as XSK), a special user space socket (Number **1** in Figure 3.2). This socket utilizes five structures to store and manage the redirected packets. **UMEM** is a memory area allocated by the user space process, consisting of a set of chunks of equal size. Each of these chunks may contain the redirected packets as they flow between Kernel and user space. The other four structures are all ring buffers, being: **FILL**, its entries point to the UMEM chunks that the user space process specifies that Kernel can take control and fill as packets are redirected; **RX**, references the UMEM chunks that the Kernel has filled with packets and the user space can read; **TX**, in case the user space process wants to send packets out, this ring's entries point to the chunks that should have their packets transmitted by the Kernel; **COMPLETION**, its entries points to chunks that had their packets transmitted out, so the user space process can set it free and reutilize it.

Another essential element of eBPF/XDP is eBPF maps. They are data structures used to store and retrieve data across multiple executions of the XDP program (**2**). Since eBPF does not allow memory allocation, these maps are the only option for stateful data storage. They also can be accessed from both Kernel and user space, being one of the main forms of communication between them. However, accessing from user space is considerably costlier in terms of performance. eBPF maps have several types to provide a wide range of data structure formats and functionalities. And, in general, they almost follow the same declaration format. Usually, the declaration requires the eBPF map type, size (maximum number of entries), entry type, and key. For the sake of brevity, only the map types that were used in this thesis will be described below:

- **Array:** Consists of an array of elements mapped to keys. These keys are unsigned 32-bit integers that increase linearly (by one) for each element. Kernel and user space can read and write to this map. However, when reading, the Kernel receives a pointer to the element in the map, while the user space receives a copy of the element's data. This map type also contains a per-CPU variant, in which each CPU has a different instance of the array map;

- **Hash:** This map hashes over the keys to access its elements. The keys of this map can be of various types and sizes, including user-defined structures. It follows the same idea as array maps in terms of reading and writing from Kernel and user space. Additionally, it also has a per-CPU variant;

- **Queue:** The elements are enqueued and dequeued through push and pop functions only, respectively. For this reason, this map does not contain keys. Kernel and user space can push and pop elements;

- **Array of maps:** Consists in an array that each element is an eBPF map (not being another array or hash of maps). Its keys are unsigned 32-bit integers that increase linearly (by one) for each element. The Kernel cannot create new entries in this array of maps, requiring either a static instantiation of all entries in the map's declaration or the user space to create them dynamically. Moreover, the Kernel can access elements of an array of maps to retrieve the pointer to the inner maps. In turn, this pointer can be utilized to access the elements of the inner map;
- **Hash of maps:** Similar to the previous map type, each element of this map is a map (not being another array or hash of maps). However, it accesses its elements by hashing a key. As opposed to hash maps, which can have keys of any reasonable size, hash of maps can only have 32-bit long keys. Lastly, it follows the same idea as an array of maps in terms of creating and accessing entries;
- **XSK map:** This data structure maps AF_XDP sockets (XSK) to certain keys. It is utilized in the XDP_REDIRECT return code to specify to which XSK a packet should be redirected.

A relevant characteristic of both array and hash maps is the multiple ways their entries can be updated. The first form uses the *bpf* library helper function, called *bpf_map_update_elem*. This function is mainly useful when the entire entry must be updated atomically, instead of a single field, if the entry is a structure. However, this function can add a high overhead. The second form is to update the entry using the pointer returned by the lookup function, used to read from these maps. This form adds the lowest overhead but does not update the entire entry atomically. Finally, the third way of updating is to utilize atomic operations. These operations add low overhead but may only update one field of an entry if it is a structure. Additionally, they are limited to the set of atomic operations available (add, subtract, logical operations, among others).

Transport layer algorithms sometimes set timers to execute certain operations when a timeout happens. For example, in TCP [30], missing acknowledgement timeouts result in packet retransmissions. Thankfully, eBPF provides a proper tool for this purpose, called bpf_timer (**3**). These timers must be stored in eBPF maps. This means that XDP programs retrieve the timers in the same way they do for other maps. Moreover, XDP programs must follow certain steps to set up a bpf_timer (through helper functions). First, they initialize the timer, setting the amount of time (in nanoseconds) for its timeout. Second, they set the callback function, i.e. the function executed after the timeout occurs. Finally, it can start the timer. eBPF also provides a helper function to cancel timers, which stops them.

Finally, we can highlight two significant libraries for eBPF and XDP programming. libbpf [9] is a library that simplifies the processes of loading, verifying, and attaching the eBPF programs. It also provides several functions that abstract the interaction with the eBPF maps in user space. On the other hand, libxdp [11] is a library whose main focus is XDP programming. Similarly to the other library, libxdp simplifies the interactions of the elements used by XDP programs. For example, setting up and handling the AF_XDP socket and its ring buffers.

3.2 XDP back-end logical model

When working with XDP, usually two programs must be created: the user space program that loads the XDP code and the XDP program attached to the NIC driver. This same design is followed by the back-end presented in this thesis. These two programs perform complex tasks, requiring an intricate design and many lines of code to represent each. Thus, this section will present an overview of what these programs are responsible for performing and how they accomplish these tasks. Figure 3.3 summarizes the XDP back-end.

Note that this section will present the functionalities of each side of the back-end model at a higher level of abstraction. This means that concepts such as multithreading, the eBPF maps utilized, and packet mutation will be described in the latter sections. So, when this section describes the responsibilities of the user space side, they are not all executed by a single thread.

3.2.1 User space side

The user space side of the back-end is written in plain C. This means that it has access to all tools available in the C language and the helper functions provided by libbpf [9] and libxdp [11]. This implies that this side can be considerably versatile in the tasks it can perform. Across the many tasks the user space performs, we can highlight that it also handles many of the interfaces of MTP's model. Because the applications are localized in the user space, it is natural that the user space side receives the requests and transmits the data from/to the application. Additionally, because AF_XDP is used to transmit packets out, the creation of packets and transmission to the network is also done in the user space. But, as this section will show, the user space is responsible for providing several other services.

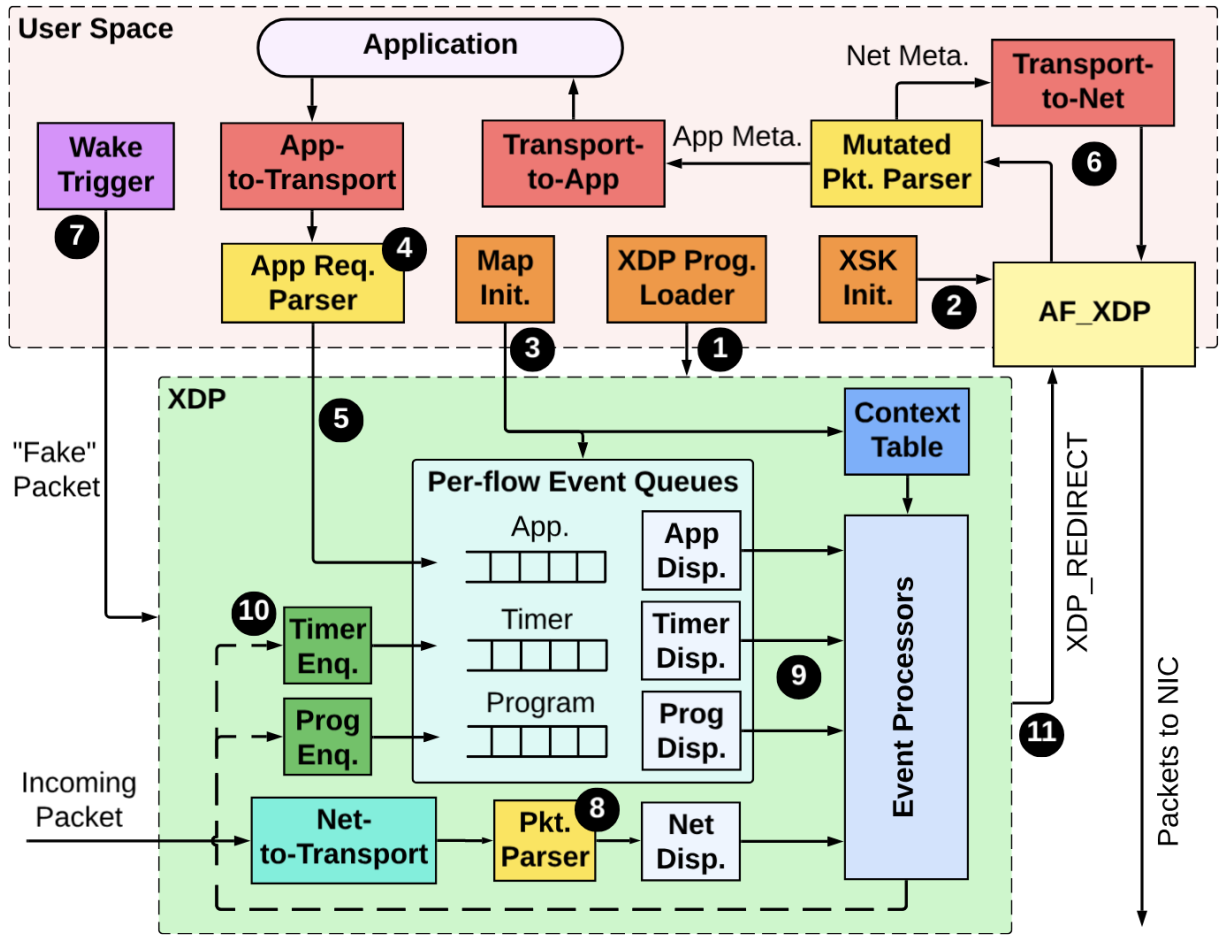


Figure 3.3: An overview of MTP's XDP back-end.

Before processing events, a few services must be executed to set the execution environment. This includes when the back-end is first launched and when a new flow is established. The services are:

Load XDP program: When the user space main thread starts running, the first major task is to load the XDP program in the Kernel hook. To do so, the library libbpf [9] provides helper functions that wrap the complex eBPF system calls, simplifying the process of loading the XDP program. Moreover, before effectively loading the code, the eBPF verifier is called to analyze if the XDP program meets all safety criteria. This is followed by the JIT compiler, which compiles it to machine code. Finally, the resulting core is attached to the hook point (**1** from Figure 3.3).

Initialize XSKs (AF_XDP sockets): To redirect packets from XDP to user space, XSK must be instantiated and have its structures initialized (**2**). For that, the XDP back-end utilizes the helper functions from libxdp [11]. During this process, the user space process allocates memory area for the UMEM. Then, it configures the socket by assigning the allocated area to it and by initializing each one of the ring buffers. As its final step, the process reserves some chunks of the UMEM for the fill ring. These chunks are then submitted to the Kernel to be filled with redirected packets later. Note that one XSK is initialized for each core utilized in this back-end's execution.

Initialize eBPF map entries: XDP back-end utilizes eBPF maps to represent the data structures from the MTP logical model, e.g. context table, event queues, and others. As such, new entries to these maps must be initialized when a new flow is established (**3**). For example, the XDP back-end instantiates a context table, fills its fields with initial values (based on the MTP program declaration), and updates a map entry with this new context table. See Section 3.3 for more information about the maps in the back-end.

After the initialization steps are finished, the XDP back-end can start executing the following services:

Parse application requests: Currently, we consider that the application requests are submitted and written to a shared buffer. The user space process reads the request and parses it into an application event (**4**). This service represents the Application-to-Transport interface from MTP's model.

Enqueue application events: After the request is parsed into an application event, the user space program needs to enqueue it to the application queue of the event's flow (**5**). This process requires a set of atomic operations and signals to ensure that state consistency is maintained. See Section 3.4.3 for more details.

Receive, parse, and send packets through AF_XDP: When entries of RX ring buffer point to UMEM chunks, it means that XDP retransmitted packets to the user space. If that happens, the user space process must read these chunks, parse each, and remove the references at the RX ring. In our model, a chunk contains a *mutated packet*. As Section 3.5 presents in detail, these packets carry two kinds of information. First, these packets contain information regarding how to send its data to the application (*application metadata*), i.e. where Transport-to-Application should write the data (based on address and offset). Second, they also contain information about how the new packets should be generated (*network metadata*), i.e. how Transport-to-Network creates the packets.

In case a packet is generated, a few steps are required to send it out. At the start, a new UMEM chunk is allocated. Then, the packet is copied to the chunk. And, finally, the process makes an entry from the TX ring buffer reference the chunk. This gives the Kernel

control over the newly allocated chunk, leading to its packet being transmitted through the NIC. Finally, via the Completion ring buffer, the user space process checks if the packet was sent out, freeing the chunk if so (**6**). This process repeats itself more times depending on how many packets the *mutated packet* asks for.

Segmentation: As presented in Section 2.2, an MTP program can specify whether the packet blueprint's data should be segmented or not. Additionally, it can specify which of its header fields vary for each segmented packet. After *mutated packet* is redirected via AF_XDP and parsed, the user space process identifies whether the information to create new packets (*network metadata*) requests segmentation. In case it does not, each network metadata is parsed into a single packet. Meanwhile, if segmentation is asked, the user space process creates multiple packets for each network metadata (varying the specified values).

Wake flows: A flow is considered "idle" if its event queues contain events but are unchanged after some time (100 milliseconds), i.e. no events were dequeued/processed during this interval. This phenomenon can happen for various reasons. For example, in TCP, consider the case where the first packet of a flow or its acknowledgement is lost. This leads to no packets arriving at the NIC driver. In turn, the XDP program cannot be triggered to execute, thus, leading to no events being processed. To avoid such occurrences, the user space may forcefully execute the XDP program if a flow is idle (**7**).

The user space process inspects if any flow is idle every 100 milliseconds. In case there is an idle flow, the process calls *xdp_program__test_run*. This is a libxdp helper function that wraps a *bpf_prog_run* syscall. In turn, the syscall runs a simulation of the XDP program at user space, by triggering it with a "fake" packet (a packet with no meaning for the flow, only used to trigger the XDP program). This results in some of the enqueued events being processed, packets being sent out and (hopefully) receiving packets back, thus, leading to a normal execution. However, *xdp_program__test_run* is a considerably expensive operation when compared to a regular XDP program execution. Therefore, idle flows must be waken up rarely.

3.2.2 XDP Side

The XDP side of the back-end is written in a restricted C. This version of the language contains the usual constructs from C, such as function declaration, structures, conditions, loops, logical and mathematical operations, etc. Meanwhile, the restrictions originate from the lack of commonly utilized C libraries and the limitations set by the eBPF verifier.

However, thankfully, the *bpf* library from the Linux Kernel [75] provides a set of helper functions that provide useful tools for XDP programming.

In addition to its helper functions, *bpf* library contains macros that enable the declaration of the constructs necessary for XDP programming. Among these macros, there are two that are important for this work. The first is a macro that declares the eBPF program's type. By knowing the program's type, the loader can know to which hook point the program should be attached. Additionally, it defines the set of helper functions and macros from *bpf* library that the program can use, since each type has different operations. Meanwhile, the second indicates that a declared structure should be interpreted as an eBPF map declaration, since each map type has a predefined set of fields.

The design of the back-end's XDP side tries to closely follow MTP's logical model. Nevertheless, due to the characteristics and limitations of eBPF and XDP, a few adaptations were necessary to make MTP model suit XDP. The adaptations are:

Network event priority: In MTP's logical model, network events are enqueued and dequeued from an event queue, similarly to other types. However, because XDP programs are triggered to run when a packet arrives (or if they are idle), it is always necessary to process the network event first. For example, in a TCP connection, acknowledgements are received in response to previously sent packets. Moreover, consider the case where the network event generated from this acknowledgement was enqueued and other events were processed first. In this case, we may unnecessarily retransmit packets the network event could have already acknowledged. Therefore, network events have the highest priority among all event types and are always processed first, causing the XDP back-end to not have a network event queue.

No flow scheduling: As mentioned in Section 2.1, targets may allow different levels of control over event scheduling. In the case of XDP, only event-type scheduling can be performed, not flow. Due to the nature of XDP programs being executed only when a packet arrives, it is important to ensure that events of the arriving packet's flow are processed. For example, if the flow scheduling policy is not set up properly, flows with lower priority may starve. This can cause the flow to be considered idle and the user space to trigger the execution of the flow. However, executing XDP programs from user space (through *xdp_program__test_run*) is expensive and should be avoided. Therefore, processing the events from the same flow as the arriving packet is simpler and safer.

Unified structure for events: In MTP, the user-declared event types (minor) are split between four major event types (application, network, timer, and program) and are defined in a struct-like format. Moreover, each of the major types contains an event queue, so different minor event types can share the same queue. Since queues must be declared

as maps and all elements of an eBPF map must have the same type, it is impossible to declare the minor event types as separate structures. Thus, the minor events that compose the same major type must be unified in a single structure when translated to XDP code.

Call-stack limitations: One of the main difficulties while designing the XDP back-end was handling the small call-stack size available in eBPF, only 512 bytes. Since most of the stack should be utilized in the event processors (where the logic of the user-declared transport algorithm is declared), we followed a few principles, such as avoiding deep-nested function calls as possible and passing pointers to structures, instead of their instances.

Void pointer issues: During the development of the XDP back-end, we noticed numerous problems related to using void pointers. This originates from the restrictions the eBPF verifier sets regarding out-of-bounds memory access. For example, consider the case where a void pointer is utilized to reference a structure instance. Even after casting it to a specific type, if one of its fields is accessed, the eBPF verifier (sometimes) announces that the value accessed is out-of-bounds. This sporadic error motivated us to avoid using void pointers in the XDP back-end. As Section 4.2.1 shows in detail, we had to separate the dispatcher module in multiple instances, each receiving an event of a different major type. Similarly, event processors used for more than one major event type had to have multiple copies written to the translated XDP back-end.

Now that the adaptations required in the XDP back-end have been explained, we can focus on the services this side of the back-end provides:

Parse packets: XDP programs are loaded into the NIC driver, meaning they are triggered when packets arrive at the network interface. When that happens, the execution of the XDP program starts from its initial function, defined in the user space loading process. This function receives a pointer to a structure of type xdp_md as an argument. xdp_md contains information about the arriving packet, such as the address of where it is stored, the address of its last byte, and indexes of the RX queue and the interface it came from.

The argument xdp_md is passed to a parser function (**8**), which represents Net-to-Transport module from MTP's model. This function retrieves the headers from the packet and creates the network event based on their values. During this process, the event is classified as one of the minor event types, later used by the dispatcher of the event's major type. At the end of parsing, the function enlarges the packet, so application and network metadata can be appended (see Section 3.5 for more details).

Dispatch and process events: Parsing is followed by dispatching the network event into the series of event processors that handle its minor type. Meanwhile, if the received packet is "fake" (from xdp_program__test_run), the network event is not generated, skipping

its processing. After the network event terminates its processing, the XDP back-end may also dequeue events of other types from their queues, according to the scheduling algorithm. Then, based on their types, these events are dispatched to the chain of event processors responsible for their execution (**9**).

Create new events: The XDP back-end allows the creation of new timer and program events as an event is processed. For timer events, the back-end provides a set of helper functions that abstract the process of initializing, starting, cancelling, and restarting a bpf_timer. When a timeout happens, a new thread is created to enqueue the timer event. During enqueuing, the XDP back-end runs functions that utilize atomic operations and signals to ensure that state consistency is maintained (see Section 3.4.3 for more details). Meanwhile, program events can be instantiated in event processors and enqueued (through a back-end's helper function) with no consistency problems (**10**).

Mutate and redirect packets: During the processing of events, it may be decided that the payload of an arriving packet or new packets should be sent out. Since the user space is responsible for both tasks, the XDP back-end must transmit those requests (metadata) to it. To do so, the back-end appends the metadata at the tail of the incoming packet. Thus, in the end, the packet may have several instances of metadata to be parsed by the user space (see Section 3.5 for more details). Finally, the packet is redirected (using the return code XDP_REDIRECT) to the user space side via AF_XDP (**11**).

3.3 Choosing eBPF Maps

As described in Section 3.1, eBPF and XDP programs are written in a restricted C. Among these restrictions, we can highlight that it is impossible to allocate memory to create complex user-defined data structures (at least not until recent Kernel updates). However, eBPF contains a collection of stateful built-in data structures, called eBPF maps, that allow storing and retrieving information while maintaining its state during multiple executions. Maps can appear in many shapes and forms (described in Section 3.1). This gives XDP developers some low degree of freedom, but it is still limited to the available formats. Additionally, they are the principal means of communication between user space and eBPF programs. These characteristics made eBPF maps relevant elements of the XDP back-end's design, being utilized to represent numerous structures of MTP's model.

Because eBPF maps are widely used in MTP's back-end, choosing the optimal map for each situation was decisive for both the back-end's functionality and performance. Before deciding which map types would be utilized we benchmarked the latency of reading and

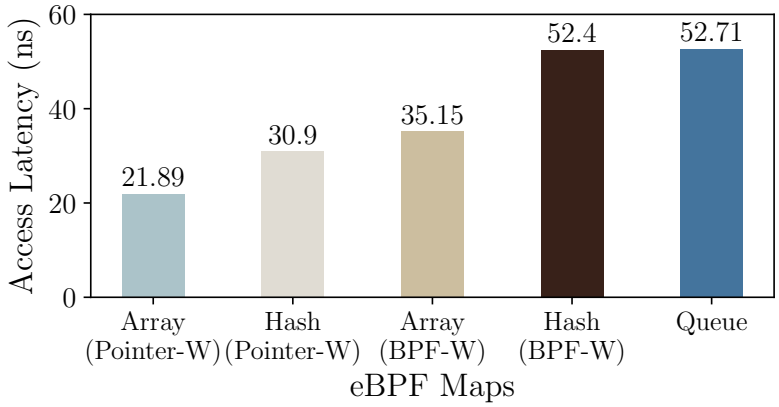


Figure 3.4: eBPF map benchmarks. Results are from read followed by write (W = Write)

writing to three of them, array, hash, and queue (Section 3.1). These types were chosen because they are the simplest maps available that can represent tables and queues, the kinds of data structure used in MTP. Specifically for array and hash maps, we compared different ways to write to a map entry, i.e. via the pointer returned from map lookup or through a helper eBPF map update function (*bpf_map_update_elem*). Meanwhile, queues can only be pushed and popped through helper functions, so we only measured it once.

The benchmark's setup consisted of two servers connected by an Ethernet switch with 100Gbps links. In this setup, one sends 100 million packets while the other server, with an XDP program loaded, reads and writes (or pops and pushes) from a map. Moreover, all packets are processed by a single core. We measured the latency (in nanoseconds) taken to access the map once for each arriving packet.

From the results, presented in Figure 3.4, it is possible to see that queue is considerably more expensive than most of the other maps. This result motivated us to avoid using queue maps, even to represent the event queues. Moreover, arrays present a significantly lower latency when compared to hashes, both using a pointer or the helper function. So, if arrays and hashes can be used interchangeably, we prioritize using arrays. Finally, it is noticeable that pointers generate lower latency than the helper function. Even so, the helper function is useful due to how it can atomically update the entire entry of a map.

Array and hash eBPF maps have an alternative version where each core contains a different map instance, called per-CPU. This variation is relevant because it avoids contention when multiple cores access the same map, since contention increases latency per map access. Figure 3.5 presents the impact in latency as an increasing number of cores concurrently access the same map of maps (array and hash of maps). Thus, it would be

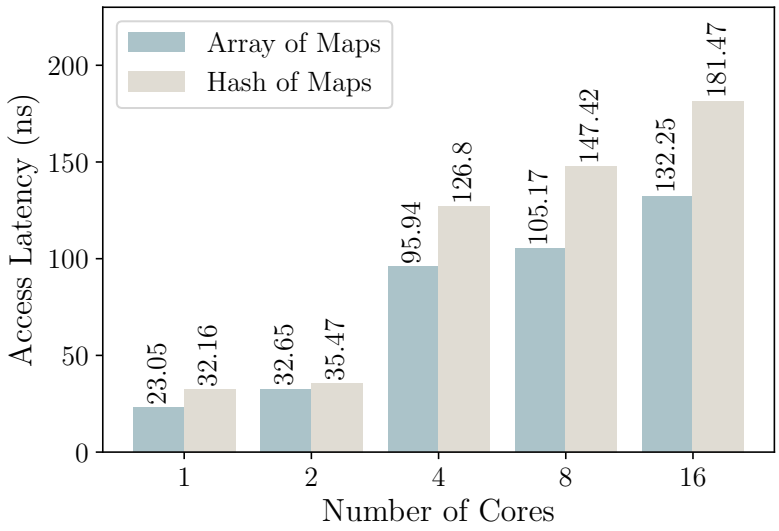


Figure 3.5: Effect of contention on eBPF map of maps.

desirable to use per-CPU maps as much as possible.

However, per-CPU maps cannot be widely utilized in XDP's back-end. This is caused by a few reasons. The XDP back-end must handle several flows simultaneously. Additionally, each flow has its own set of event queues, one for each major event type (except network event). To represent this concept in XDP, we must utilize a map of maps. This is because each of its entries represents a different flow and, since this is a map of maps, each entry will be another map, i.e. the event queue. However, there are no per-CPU variations of map of maps in eBPF. Thus, it is currently inevitable to avoid contention in the XDP back-end with the current version of eBPF.

During the design of the XDP back-end, we had to decide to use either an array of maps or a hash of maps to represent the event queues of all flows. At first, a hash of maps appeared to be the best option, because you can use the flow identifier as the map's key, instead of an integer. However, both types of map of maps prohibit keys longer than 32 bits. This is a severely limiting factor since flow identifiers tend to be much longer than that. For example, the 4-tuple utilized in TCP takes 96 bits (if using IPv4). To avoid this limitation, we chose to have an additional hash map. The purpose of this map is to hash the flow identifier into a unique integer (starting from 0 and increasing by 1 for each new flow). And, instead of using a hash of maps, we use an array of maps that takes this integer as its key. Thus, retrieving the event queue for the flow.

Now that we explained the reasoning behind choosing certain eBPF map types, Figure

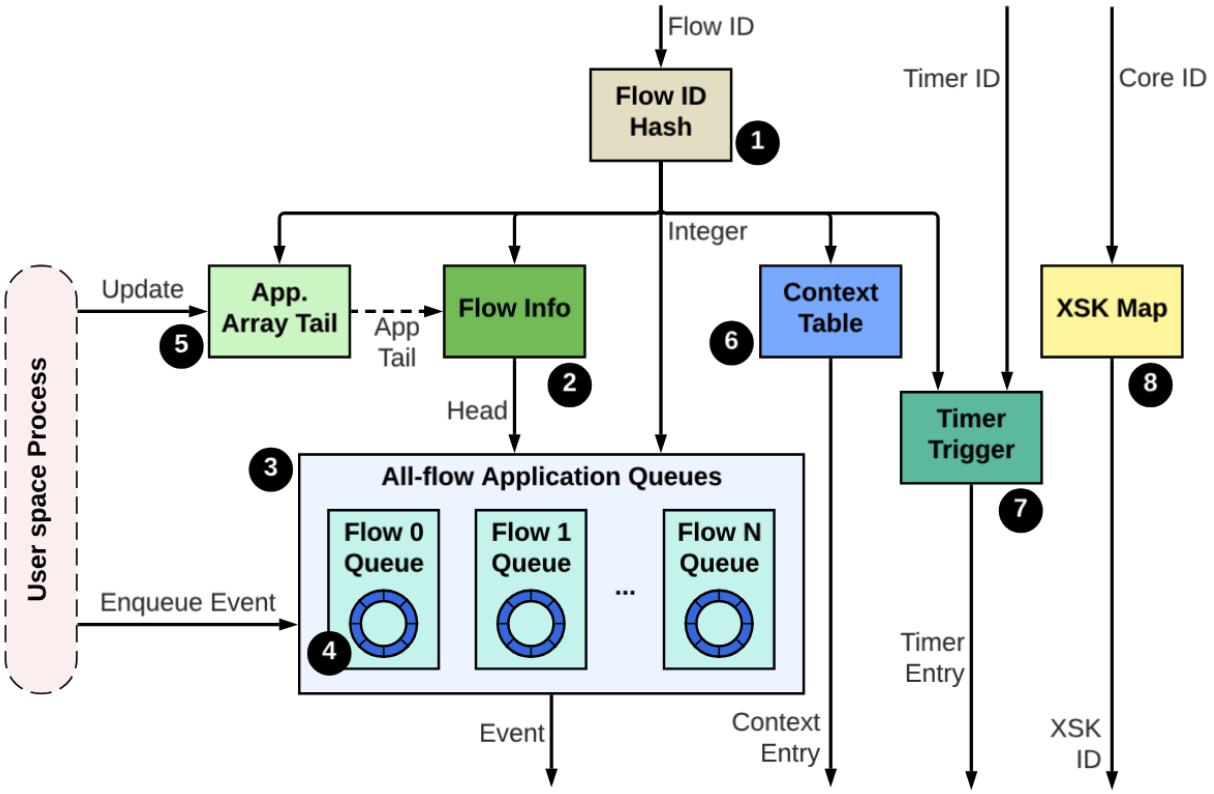


Figure 3.6: A summary of the maps utilized for application event processing, the keys and the values retrieved with them.

3.6 presents all maps used, their keys and values, and their relationships. Note that this figure presents maps used for application event processing. This means that the application array tail map is not used for other types of events. Additionally, there is an all-flow event queue for each major type. Furthermore, now we disclose in detail the maps employed in the XDP back-end:

- **Flow id hash (1):** This hash map takes the flow identifier used by the implemented protocol (e.g. 4-tuple in TCP) and hashes it to an integer. A pointer to this integer, called *flow-id*, is later utilized as the key of some of the other maps;
- **Flow info map (2):** This map contains information for event scheduling. The entries consist of information about each of the flows' queues. Since the queues are actually implemented as circular buffers, the entries of this map include the head,

tail, and length of each queue. Finally, it takes *flow-id* as the key and returns the pointer to flow information;

- **All-flow event queues (3):** Each event type must have a map of maps representing the queues for all flows, i.e. application, timer, and program events each have their map of maps. Then, this collection of queues for the flows is represented by an array of maps. This map of maps takes the *flow-id* returned from the flow identifier hash as its key and returns a pointer to the event queue of that flow;

- **Event queue (4):** Due to the low performance of the queue eBPF map type, event queues are implemented as circular buffers using array maps. This map takes as key the current head of the circular buffer (from flow info map) and returns a pointer to the next event to be processed;

- **Application array tail (5):** The user space side is responsible for enqueuing new application events to the application queue. It does that by appending at the tail of the circular buffer (if empty) and moving the tail forward. To move the tail forward the user space process would need to update the flow info map directly. However, we chose not to have this direct update to avoid state inconsistency issues and the overhead its treatment adds. Instead, the user space process writes the new tail to the application array tail map. So, after the network event is processed, the XDP program retrieves the most recent tail from this map (using *flow-id* as the key) and updates the flow info map;

- **Context table map (6)**: During the execution of events, event processors need to access stateful information to make decisions, retrieve and update parameters, etc. This all originates from the context table, one of the modules of MTP's model. The context table is represented by an array map. It uses *flow-id* as its key and returns the pointer to the context information of the event's flow;

- **Timer trigger hash (7)**: eBPF specifies that timers must be stored in maps. Each hash map's entries consist of a structure containing a *bpf_timer* and the timer event to be enqueued after its timeout. Since a flow may set many different timers simultaneously, the hash's key cannot be only the *flow-id*. Thus, the key is a structure containing *flow-id* and a timer identifier, that uniquely identifies that timer for that flow;

- **XSK map (8):** This structure maps cores with their XSK (AF_XDP socket). It utilizes the core's identifier as its key. Through this map, packets can be redirected to a UMEM of a core.

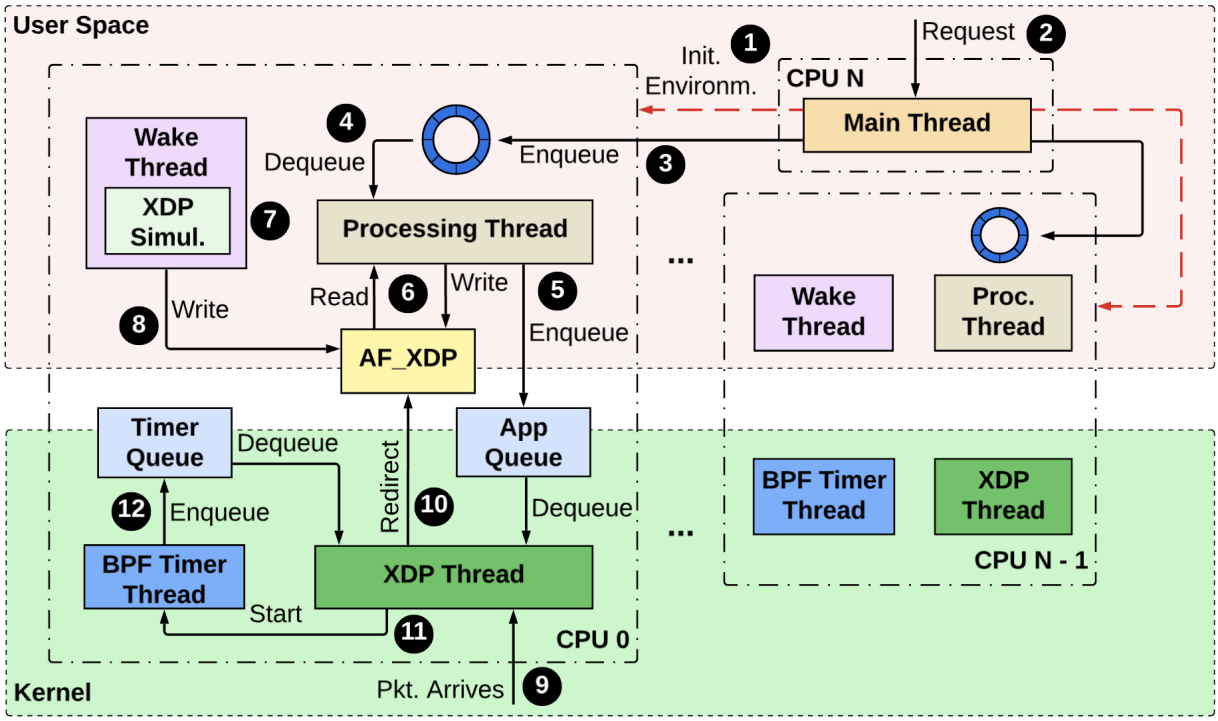


Figure 3.7: An overview of the threads utilized in the back-end.

3.4 Multithreading

As explained in Section 3.2, the XDP back-end is composed of two sides, user space and XDP. Each side is responsible for performing a set of services. To optimize the performance of this environment, the back-end was designed with multithreading in mind. However, the XDP side (Kernel) presents a restricted set of threads, due to its limitations and how the tool works. So, we explored in more depth how to apply multithreading in user space. Figure 3.7 displays the threads contained on the two sides of the back-end, in a system that has N cores.

This section will present how and why the threads were used in this context, as well as show some of the challenges to maintaining state consistency.

3.4.1 User space Threads

The user space side of the back-end performs several tasks. This includes environment initialization, receiving redirected packets and sending new ones, and triggering idle flows (Section 3.2.1). Since the user space, written in plain C, does not contain the limitations of eBPF/XDP, the multithreading design was much less restricted than the back-end's XDP side. This enabled us to implement a single producer multiple consumer multithreading design in the user space. In this design, a main thread is responsible for the initialization and dispatching of tasks. Meanwhile, worker threads receive these tasks and execute more computationally demanding steps.

Given a number N of cores available in the system, the user space process (the main thread) executes in core N. This thread executes tasks of lower demand, i.e. services that are less computationally demanding when compared to others, such as packet generation and transmission. So, the main thread is responsible for initializing the whole environment (**1** in Figure 3.7). To do so, it utilizes libbpf to load the XDP program and libxdp to create an XSK (AF_XDP socket) and its ring buffers for each of the other $N-1$ cores. The main thread must also generate a pair of worker threads for each of the $N-1$ cores, called *Processing Thread* and *Wake Thread*. Lastly, it creates entries for eBPF maps as new flows are established (event queues, context table, among others)

The multithreading design followed by the back-end consists of a single producer multiple consumer approach. In this design, the main thread dispatches tasks for the workers to perform. There are many other possible designs to be followed in this situation instead of the design currently used. However, this design was chosen since it looked natural for us to be applied in this situation. We assume that application requests are not recurrent; so, leaving their parsing and dispatching of application events to a single core allows the other cores to focus on executing more demanding tasks. Additionally, the vast majority of our efforts were spent designing the XDP side. So, other designs will be explored in future work.

The main thread must receive requests from the applications (**2**) (which are written in a shared buffer). Then, it parses them into application events, and dispatches the events to the core responsible for processing the events' flow. Each of the $N-1$ cores has a circular buffer that stores the events sent by the main thread (**3**). Finally, the events are retrieved by the worker thread (Processing Thread) (**4**).

The first of the worker threads, *Processing Thread*. It is a thread that runs in a busy loop, constantly checking if it should perform its two services. The first service being application event enqueuing (**5**). For the first service, it checks whether there are any

events in the circular buffer of its core. If there is, it retrieves the event from the data structure and uses libbpf's [9] helper function to copy the event to the application event queue of its flow. During this process, atomic operations and signals are used to achieve state consistency (see Section 3.4.3 for more information).

Processing Thread provides a second service: handling packets through AF_XDP (**6**). In this task, the thread identifies whether AF_XDP's RX ring buffer has entries referencing UMEM chunks. If there is, the packets are retrieved from the chunks and the references from the RX buffer are freed. This is followed by the packet being parsed, where application and network metadata are analyzed. In case there is an application metadata, the packet's payload is copied to the application's buffer. Meanwhile, if there are samples of network metadata, new packets are generated, copied to UMEM chunks, and referenced by the TX ring buffer. Finally, the *Processing Thread* checks through the Completion ring buffer whether the new packets were transmitted out, freeing the chunks if so.

The second user space worker thread, called *Wake Thread*, triggers the processing of idle flow events. As explained in Section 3.2.1, flows are considered idle if they have events to be processed; however, after a certain duration of time, all event queues remain unchanged, i.e. no events were dequeued and processed. To trigger the execution of the XDP program, *Wake Thread* utilizes an eBPF syscall named *xdp_program__test_run* to simulate the execution of the program on the thread (**7**), sending a "fake" packet. Moreover, since this simulation does not allow redirecting packets through AF_XDP the syscall provides an argument that contains the fake packet, mutated during the simulation's execution. Then, this mutated packet can have its metadata parsed, write new packets into UMEM (**8**), and transmit them via AF_XDP.

The decision behind splitting the worker threads in two was motivated by a characteristic of *Wake Thread*. Flows are only considered idle if they are unchanged after X ms. As such, it is unnecessary to execute any other operations during this period. Instead, we followed the design of making the thread sleep. Thus, while *Wake Thread* sleeps in intervals of X ms, *Processing Thread* keeps executing in a busy loop, constantly checking if new application events should be enqueued or new packets arrived.

3.4.2 Kernel Threads

After the XDP program attached to the NIC driver, every time a packet arrives at the NIC the XDP code must be executed. To do so, a Kernel thread called *XDP Thread* is created when a packet is received (**9**). This thread processes the event parsed from the packet and other enqueued events, and is terminated after redirecting the packet to AF_XDP

(**10**). Due to the simplistic nature and the limited set of operations of eBPF/XDP, it is impossible to have multithreaded design (i.e. multiple XDP Threads per core). So, in a system with multiple cores available to run the XDP program, the cores can process packets concurrently, but each core can only process one packet each time.

eBPF contains a structure called *bpf_timer* that must be stored in maps and is used to represent timers. During the configuration of this structure, we can set the interval of time for its timeout and the callback function. Additionally, we can start (**11**) and cancel the timer (for more information see Section 3.1). When an instance of *bpf_timer* has a timeout, eBPF creates a secondary Kernel thread (*BPF Timer Thread*). This thread executes the callback function set during the timer configuration. When the callback function finishes its execution, the thread is also terminated. Moreover, the callback function set for all *bpf_timer* operations is a function that enqueues the event to the timer event queue of its flow (**12**).

3.4.3 State Consistency

The usage of multithreading can optimize the utilization of computational resources while processes are executed. However, as a consequence of its usage, concurrent access to shared memory resources (such as eBPF maps and UMEM) may result in race condition and state inconsistency. This led us to examine ways to implement algorithms that use multithreading while applying techniques that assure state consistency. There are five main points where concurrency issues can happen:

- **Application event enqueue/dequeue:** In a core, two threads can access the application event queues: *Processing Thread* (user space) and *XDP Thread* (Kernel). If the user space thread enqueues a new event while a packet from the same flow arrives, both threads will try to access it simultaneously. This can possibly result in the same map entry being accessed by both. Since this is a single consumer and single producer situation, to avoid this issue, application events hold a boolean *occupied* field. This field dictates whether the eBPF map entry can be updated by the user space. Thus, before enqueuing the event, *Processing Thread* evaluates whether the current entry can be updated. Meanwhile, *XDP Thread* changes this field atomically when the application event is finished processing.

- **Timer event enqueue/dequeue:** Two types of threads can access the timer event queues: *XDP Thread* and *BPF Timer Thread* (both from Kernel). This situation in particular presents a multiple producer and single consumer design. Because multiple

timeouts can happen simultaneously, leading to *BPF Timer Threads* being created to enqueue a timer event. And if a packet from the same flow arrives, an *XDP Thread* may also access the same queue and event. To remediate this problem, two booleans are employed:

  - *valid_bit*: It specifies whether the eBPF map contains a valid event that can be dequeued. The *BPF Timer Thread* sets as valid when it enqueues. Meanwhile, *XDP Thread* only dequeues if it is valid, setting as invalid after it finishes processing;
  - *executing*: It is a signal used between *BPF Timer Threads* to identify whether there is a thread currently enqueuing an event. In case another thread tries to enqueue and the signal is true, it restarts its timer (for a short duration) to try enqueuing again later. Although eBPF provides spin locks, we chose to use a signal and restart for two reasons: seizing control of a lock is a computationally demanding operation; and two *BPF Timer Threads* concurrently accessing the same queue might be unusual, so, postponing one of them (for a short time) is can be justifiable.

- **Application request enqueue/dequeue:** The main thread of the user space is responsible for dispatching parsed application requests to circular arrays read by *Processing Threads*. As such, there might be issues when the main thread writes an entry while the worker thread reads it concurrently. To avoid this problem and constant lock retrieval, we followed an approach that utilizes atomic operations to lockless pop and push from the circular arrays [40].

- **AF_XDP data structures:** While receiving redirected packets and transmitting newly generated packets through AF_XDP, its data structures are accessed by two threads: *Processing Thread* and *Wake Thread*. Since both threads are in the user space, we can utilize regular *pthread* mutex locks. With it, we can lock one thread while the other executes the complex operations of AF_XDP.

- **xdp_program__test_run concurrency:** If a flow is idle, *Wake Thread* utilizes *xdp_program__test_run* system call to simulate the XDP program at user space and process some events. In a situation where the system call is running but a packet arrives at the network interface (starting a *XDP Thread*) state consistency could be affected. This is because, both threads would access the same eBPF maps and, most importantly, context table concurrently. This could lead to an inconsistent context table. However, XDP programs run with preemption disabled. This means that the

xdp_program__test_run syscall can run to completion before the *XDP Thread* starts executing, assuring state consistency.

3.5 Packet Mutation

In an ideal situation, we would want both packet creation and transmission to be done on the XDP side. This would avoid the unnecessary overhead of redirecting the packet to user space to, then, transmit packets out via AF_XDP. However, XDP has one packet in, one packet out design philosophy. In this design, only one single packet can be received, have its fields changed (e.g. IP addresses and ports) and transmitted out using the return code *XDP_TX*. This characteristic is considerably limiting for transport programming since more than one packet might need to be sent out in certain situations (e.g. TCP congestion window has space for more than one packet). Therefore, to overcome this limitation, we designed a technique we call packet mutation.

When a packet arrives at the network interface, a thread of the XDP program, loaded in the NIC driver, is initialized to handle the packet. The argument of the main XDP function consists of a reference to an *xdp_md* instance. This is a structure that contains pointers to where the packet is stored and information such as the RX queue and network interface identifiers. The two most relevant pointers it has are *data* and *data_end*, which contain the address of the packet's first (head) and last byte (tail), respectively. These two addresses are essential for XDP since they are the only form to access the packet. Moreover, with the two pointer we can parse headers, retrieve the packet's payload, and other operations.

Another operation that eBPF allows to perform on the arriving packet is changing its values, i.e. mutating it. For example, switching MAC/IP addresses and TCP ports to transmit the packet back to its source, or replacing a header with an entirely new structure. Additionally, eBPF permits modifying the packet's length by extending or retracting its head and tail. These two techniques allow eBPF/XDP developers to completely transform the packet. Although, still limited to a maximum packet length dictated by the driver and MTU [2]. With packet mutation, the XDP back-end can transmit information from XDP to the user space through AF_XDP redirection, as an alternative to eBPF maps (which is computationally costly to access at user space side).

A packet may contain several headers, as well as a payload, such as the one present in Figure 3.8 (a TCP packet). Because these headers can occupy a considerable space of the packet, we chose to maintain only the most relevant information of the headers. So,

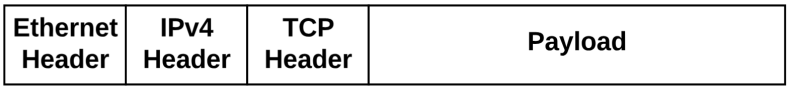


Figure 3.8: Common TCP packet

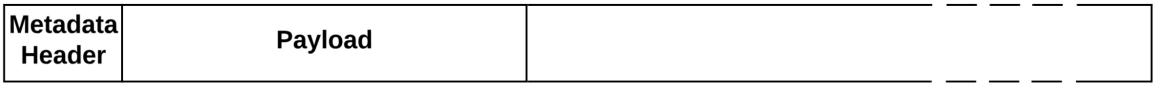


Figure 3.9: TCP packet after getting resized and copied the metadata header.

in the back-end, while the arriving packet is parsed and the headers gathered to create the network event, some of the most relevant header fields, mostly related to routing, are saved in a structure called *metadatada_hdr* (metadata header). This structure contains values that will be shared among all packets generated at user space, such as addresses and ports. So, after filling the metadata header instance, we must replace the headers with the metadata.

Replacing the packet headers starts by resizing the packet (changing its head and tail placements). Because the payload is usually much larger than the headers and its size can vary considerably, it could be more complex to change its address and move it around the packet. Due to this, all resizing is done around the payload, i.e. the payload is never mutated, but its surroundings are. This means that headers are removed, the packet can be extended, but the payload never changes.

With that presented, the first step of packet mutation is to resize the header area of the packet, so it matches the *metadatada_hdr* length. In case *metadatada_hdr* is larger than the headers combined, the head is moved to the left (extended). Meanwhile, in the opposite case, the head moves to the right (retracted). This is step followed by copying the *metadatada_hdr* to the newly changed header space. Finally, the last step is to extend its tail so new application or network metadata can be appended to its end. Figure 3.9 presents a TCP packet after it was resized from both head and tail. In this figure, we replaced the headers with metadata header (retracting) and extending its tail after the payload.

After packet parsing is done, events can start being processed. During the execution of events, it may be decided that the packet's payload should be sent to the application. To hold the information that the user space process uses to transmit the data to the application (address, offset, length, among others), the event processor instantiates an *app_metadata* (application metadata). This instance is filled with the proper values and its appended to the left-most empty space of the packet. Similarly, during event processing,



Figure 3.10: TCP packet after having application and network metadata appended to it.

it may be determined that new packets should be sent out. If that happens, a *net_metadata* (network metadata) instance is created, filled with values unique for each packet (in TCP's case, ACK, sequence number, and others), and attached to the left-most empty space. Figure 3.10 presents an example where application metadata and two instances of network metadata were appended to the packet.

During the design of the back-end's packet mutation, two approaches were considered regarding tail resizing. The first, currently utilized and described in this section, is the static design. In this design, the tail is extended once (after packet parsing) and for a fixed size. The advantage of this design is a potentially better performance since fewer tail extensions are performed. Meanwhile, the second is a dynamic design. In this approach, the packet tail is gradually extended, as events are processed and metadata are appended. This results in a packet with the exact length needed. However, it may have a higher overhead due to the many tail extensions. Currently, the static approach is being used in the XDP back-end. Nevertheless, future work can include an in-depth comparison and possible optimizations for both.

Chapter 4

XDP Code Generator

Modular Transport Programming is a programming framework that aims to simplify the process of developing transport algorithms. The framework's language achieves that by enforcing a higher level of abstraction and modularization to the code. However, another characteristic of MTP is its target-agnostic nature. This means that MTP is written in a unique language, independent of the target to which the code will be compiled. Thus, during the development of MTP, we developed a code generator for the translation from MTP to code that can be plugged into the XDP back-end, being one of the contributions of this thesis.

The compiler developed by our group is based on Antlr [3]. This tool is a parser generator that, by providing grammar, executes the lexing and parsing steps of compilation and generates an Abstract Syntax Tree (AST). Antlr also provides a simple way to create visitors that access the AST nodes, which may be utilized for semantic analysis (not yet implemented in the compiler's current) and, most importantly, code generation. Thus, a code generator for MTP specifies what each node visitor should perform to create the code for the target.

Due to the similarities in syntax between MTP and XDP, it was straightforward to translate a considerable part of the constructs. For example, dispatchers and event processors could be translated into functions. Moreover, conditions and simple statements are almost equal before and after compilation. However, due to the limitations of eBPF, a subset of constructs were considerably more complex to translate, which will be discussed in more detail in this chapter. It is worth noting that the code generator currently only implements the target-independent parts of the MTP framework, which are the core of transport algorithm implementations.

4.1 The basic translation

Because MTP is written in a C-like language and XDP is written in a restricted C, most of the translation process is considerably simple. Both languages provide the same group of mathematical and logical operators for their statements, variable declaration format, and condition constructs. For instance, the code below shows an example of an event processor declared in MTP's language that implements slow start and congestion control algorithms in TCP. Since no new events are created the event processor returns *void* and receives the event, context, and intermediate output as arguments. Moreover, this event processor performs a series of mathematical and logical operations, while accessing fields from the context table.

```
void slows_congc_ep (ACK ev, tcp_context ctx, tcp_scratch s) {
    if(s.skip_ack_eps)
        return;
    uint32 SMSS = 1440;
    if(s.change_cwnd) {
        if(ctx.cwnd_size < ctx.ssthresh) {
            ctx.cwnd_size = ctx.cwnd_size + SMSS;
        }
        else {
            uint32 add_cwnd = SMSS * SMSS / ctx.cwnd_size;
            if(add_cwnd == 0)
                add_cwnd = 1;
            ctx.cwnd_size = ctx.cwnd_size + add_cwnd;
        }
    }
}
```

Meanwhile, the next code presents the translation of the same event processor to XDP. The event processor is represented through a function declaration, which also returns *void*. However, the arguments differ from the original implementation in the MTP language. Events of minor type *ACK* are classified as network events (major type). As such, since all events of the same major type share the same event queue and eBPF maps require all elements to be of the same type, all minor event types are merged into one single structure declaration. Thus, the *ACK* minor type is translated to *net_event* (network event), which contains the fields from *ACK* and other minor network event types.

The next two arguments, *ctx* and *int_out*, represent the same elements as the MTP declaration. Nevertheless, XDP includes a fourth argument, being a reference to *xdp_md*. As explained in Section 3.5, this structure contains information regarding the arriving

packet that triggered the program's execution. *xdp_md* is passed across the functions in the XDP back-end to allow the mutation of the packet in the event processors.

```
static void slows_congc_ep(struct net_event *event, struct tcp_context *
    ctx, struct tcp_scratch *s, struct xdp_md *redirect_pkt) {
     if(s->skip_ack_eps)
         return;
     __u32 SMSS = 1440;
     if(s->change_cwnd) {
         if(ctx->cwnd_size < ctx->ssthresh) {
             ctx->cwnd_size += SMSS;
         }
         else {
             __u32 add_cwnd = SMSS * SMSS / ctx->cwnd_size;
             if(add_cwnd == 0)
                 add_cwnd = 1;
             ctx->cwnd_size += add_cwnd;
         }
     }
}
```

4.2 The complexity added by XDP

MTP and XDP are both written in variants of C and translating one language to another is relatively simple. Nevertheless, some specific characteristics of XDP brought a new layer of complexity to the code generator. These characteristics range from clear limitations of XDP, such as its lack of support for unbounded loops, to more nuanced ones, such as the eBPF verifier presenting difficulties when evaluating in-bound memory accesses when void pointers are involved. This section will provide further insight into the major difficulties faced during the creation of the code generator.

4.2.1 Void pointers

To create functions that could be utilized for different major event types, initially, *void* pointer was utilized to typify events as they were passed as arguments across functions. However, as we tested the XDP code generated, we noticed several errors announced by the eBPF verifier regarding out-of-bounds memory access to the events. Later, we discovered these errors originated from using *void* pointers. This may be because the verifier may not

know during its analysis the exact boundaries of the events when *void* pointers are used, even if a casting is applied.

This problem, regarding using void pointers to pass events, caused a few changes to the code generator. Originally, translating the dispatcher table from MTP to XDP was straightforward due to its simple concept, represented by a function containing a switch case construct. Following this idea, we used a *void* pointer to pass all event types to the dispatcher and cast the type of the event in the event processors. However, the *void* pointer problem made us reconsider how the dispatcher would be translated. Instead of having a single dispatcher for all events, the XDP back-end has a dispatcher function for each major event type. For example, the code below is one of the dispatchers generated for network events from the dispatch table declaration in Figure 2.4.

```
static void net_event_dispatcher(struct net_event * event, struct
    context *ctx, struct xdp_md *redirect_pkt) {
      struct interm_out inter_output;
      switch (event->event_type)
      {
      case ACK:
         rto_ep(event, ctx, &inter_output, redirect_pkt);
         fast_retr_rec_ep(event, ctx, &inter_output, redirect_pkt);
         slows_congc_ep(event, ctx, &inter_output, redirect_pkt);
         ack_net_ep(event, ctx, &inter_output, redirect_pkt);
         break;
      case DATA:
         data_net_ep(event, ctx, &inter_output, redirect_pkt);
         send_ack(event, ctx, &inter_output, redirect_pkt);
         app_feedback_ep(event, ctx, &inter_output, redirect_pkt);
         break;
      default:
           break;
      }
}
```

Section 2.2 mentions that MTP allows the declaration of event processors that handle more than one minor type of event, declared similarly to a function template in C++. In this situation, the void pointer would be convenient since a single function could receive events of several types. However, as explained previously, we chose to avoid *void* pointers completely. This caused another change in the code generator's design. Instead of having a single event processor shared between major events, the code generator creates copies of the same event processor, each executing a different major event. The only difference between these copies is the function's identifier and the type of the event in its parameters.

The example below shows the declaration of two copies of the same EP (*example_ep*), responsible for processing application and network events, respectively. Although not shown, the code inside the functions would be exactly the same.

```
static void app_example_ep(struct app_event *event, struct ex_context *
    ctx, struct ex_scratch *s, struct xdp_md *redirect_pkt) {
     ...
}

static void net_example_ep(struct net_event *event, struct ex_context *
    ctx, struct ex_scratch *s, struct xdp_md *redirect_pkt) {
     ...
}
```

The design decision to have multiple copies of the same event processor could potentially lead to problems with the eBPF verifier in older versions of the Kernel. This is because, until version 5.2, eBPF did not allow programs with over four thousand instructions [7] (see Verifier section). Thus, having copies of the same code could easily exceed the limit of eBPF. However, in more recent Kernel versions, this number was increased to over one million instructions, which led us to safely create programs with this design. Even so, although difficult, it is possible to surpass the new limit.

4.2.2 Loops

Although the translation between the two languages was simple for most of the constructs, some abstractions provided by MTP and the limitations of XDP made parts of the translation difficult. Among these difficulties, the most complex one was handling loops. While MTP allows the declaration of loops that iterate over a variable number of times, XDP prohibits the declaration of this kind of loop and unbounded loops. Moreover, the eBPF verifier rejects any programs that do not follow this specification. This means that when using regular iterator constructs, XDP requires that the maximum number of iterations is constant.

Because *for* and *while* iterators must have a constant number of iterations in XDP, eBPF provides a helper function called *bpf_loop* [7] (see *bpf_loop* section). This function allows the declaration of loops that the eBPF verifier assures will terminate, even if the number of iterations is variable. However, since *bpf_loop* is a helper function, it does not follow C's loop construct format. Instead, when this function is called, four arguments are passed: the number of iterations (N), the callback function that will be executed N times, the parameters passed to the callback function, and a flag with no use currently (set to 0).

```
1  static long loop_func(__u32 index, struct loop_args *arg) {
2      if(arg->bytes_to_send == 0)
3          return 1;
4      __u32 pkt_data_len = 0;
5      if(arg->bytes_to_send <= SMSS) {
6          ...
7      }
8      ...
9      arg->ctx->send_next += pkt_data_len;
10     return 0;
11 }
12
13 static void loop_ex_ep(struct app_event *event, struct context *ctx,
       struct scratch *s, struct xdp_md *redirect_pkt) {
14     ...
15     struct loop_args arg;
16     __builtin_memset(&arg, 0, sizeof(arg));
17     arg.bytes_to_send = bytes_to_send;
18     arg.ctx = ctx;
19     arg.redirect_pkt = redirect_pkt;
20     bpf_loop(num_loops, loop_func, &arg, 0);
21     ...
22 }
```

Figure 4.1: Example of *bpf_loop* call.

Figure 4.1 presents an event processor that calls *bpf_loop*. On line 15 it instantiates the argument of the callback function, followed by filling its fields, and line 20 executes the helper call. Moreover, *loop_func* (lines 1-11) declares the callback function. By default, a callback function must have two parameters, the first being the index in the loop (starting from 0), and the second consisting of the arguments passed in the *bpf_loop* call. Additionally, it must return one of two long integers: 0, in case the loop should continue in the next iteration; and 1, if the loop must be terminated, skipping the following iterations.

In a common iterator from MTP (*for* and *while*), variables declared outside of the iterator's scope can still be accessed within its inner scope. However, in eBPF, the callback function is declared in a separate scope. Thus, all variables used in the loop but declared outside of it must be passed through *bpf_loop*'s argument. For example, in Figure 4.1, because the callback function accesses context in line 9, the event processor must pass *ctx* as argument. Moreover, this situation becomes more complex when nested loops are utilized since the variables passed across the *bpf_loop* callback function arguments can be declared at many points of the nested loop.

To handle this issue, the XDP code generator creates a tree data structure, called

Loop Argument Tree (LAT). This tree classifies the variables in each loop scope, whether declared in the node's scope or an outer scope. Each node of the LAT contains a list of variables declared in the node's scope (*declared list*) and a list of variables passed from an outer scope that contains the current node (*passed list*). The first is filled while traversing the loop, where a new entry is added to the list each time a variable is declared in that scope.

Meanwhile, the second can add a new entry if a variable is utilized by the current node's scope, but not declared inside the current. When this case occurs, the nodes of LAT are retroactively accessed in search of the variable's declaration, from the current node until the parent node that declares it or the root. During this process, if a node does not declare it, the variable is added to the node's passed list and the node's parent is accessed next.

Figure 4.2 presents an example of an event processor that declares a series of nested loops. In each of these loops, they may declare variables and utilize those declared in its scope or outer scopes. Meanwhile, Figure 4.3 represents the LAT generated from the example. The construction of this tree starts from its root, the EP scope node. Its *declared list* is filled with the variables defined in it (a, b, c). Then, the node of loop 1 from the AST is accessed and the node from LAT is created, filling its *declared list* (d, e) and its *passed list* (a, b). Since a and b were declared in Loop 1's parent node (EP), no changes are made to the parent node's lists.

At loop 2, the variable f is added to its *declared list* and b to its *passed list*. Moreover, no more changes are made since its parent node (loop 1) already includes b in the *passed list* and EP declares b. At loop 3, variable g is added to its *declared list* and d to the *passed list*. No changes are made to the parent node (loop 1) because it declares d. Meanwhile, at loop 4, no variables are declared, and c, e, and g are accessed from outer scopes, filling the *passed list*. While retroactively accessing the parent nodes of the LAT in search of the declaration of these variables, c and e are added to the loop 3 passed list; and since loop 1 declares e and contains c in the passed list, and EP declares c, no further changes are made.

After a loop node from the AST is completely analyzed and its LAT node is filled, the code generator creates the callback function that represents the loop. This is accompanied by the declaration of the structure that constitutes the function's arguments, i.e. the variables used but not declared in it (the passed list). Moreover, the function whose scope contained the loop declaration must instantiate the newly created structure, initialize its fields accordingly, and perform the *bpf_loop* call, as shown in lines 15-20 of Figure 4.1.

```
1  void loop_example (LOOP_EX ev, ctx_ex ctx, scratch_ex s) {
2      uint32 a, b, c;
3      for(...) {           // Loop 1
4          uint32 d, e;
5          a = b;
6          for(...) {       // Loop 2
7              uint32 f;
8              b = f;
9          }
10         for(...) {       // Loop 3
11             uint32 g;
12             g = d;
13             for(...) {   // Loop 4
14                 c = g;
15                 g = e;
16 }   }   }   }
```

Figure 4.2: Example of nested loops.

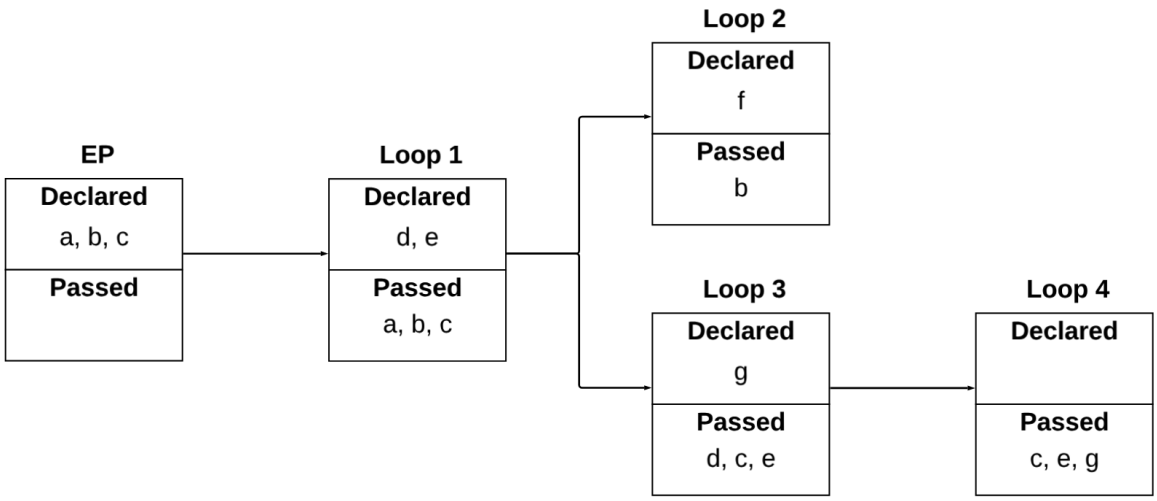


Figure 4.3: Loop argument tree generated from the nest loop example.

4.2.3 Unifying minor events

As presented in Section 2.1, in MTP's model each minor event is classified into four major types (application, network, timer, and program). Additionally, when events arrive at the event scheduler, they are enqueued to one of the event queues, based on their major type.

To represent these queues, the XDP back-end utilizes an eBPF map array; and, when declaring a map, eBPF requires all map entries to have the same type. Thus, all minor events sharing the same major type must be merged into a single declaration, so all minor events can be enqueued to the same queue.

While the AST nodes that declare minor event types are accessed, the code generator maintains lists storing all variables declared for each major type. Moreover, if a field with the same identifier and type is defined across two different minor events, a single copy is added to the list. Then, when the XDP code is written, the major types are declared in separate structures, which can be used to define the types of entries of the eBPF maps.

The two sections of code below demonstrate the unification of two minor event types into a single network type declaration. In addition to the fields of each minor type, the major declaration contains two more fields by default: *event_type*, which specifies the minor type of that event instance; and *ev_flow_id*, which represents the event's flow ID.

```
event ACK : INCOMING::NET_EVENT {
    uint32 ack_seq;
    uint32 rwnd_size;
}
event DATA : INCOMING::NET_EVENT {
    uint32 data_len;
    uint32 seq_num;
}
```

```
struct net_event {
    enum minor_event_type ev_type;
    __u32 ev_flow_id;
    __u32 ack_seq;
    __u32 rwnd_size;
    __u32 data_len;
    __u32 seq_num;
}
```

4.2.4 Out-of-bounds access checks

MTP contains an array built-in type that works similarly to arrays found in C++. As such, when declaring a variable of this type, it is necessary to specify its length. In a C++ program running in user space, if an out-of-bounds index is used to access the array the compiler may announce a warning but, even so, it will lead to undefined behaviour (UB).

This problem can hinder the program's execution, but will not cause major problems for the system. However, that is not the case when working with the Kernel, since operations such as these (e.g. accessing inappropriate memory) can lead to instability and insecurity of the system [8].

As presented in Section 3.1, one of the main motivations for creating eBPF was to allow deploying code to the Kernel simply and safely [8]. Moreover, to ensure the security of the program when executed at Kernel, the eBPF verifier analyzes it to identify any potential sections of code that may be harmful. This includes out-of-bound memory access. Thus, when an array is accessed in an eBPF/XDP program, the verifier requires that the index used to access it must be within the array's boundaries. The XDP code generator fills this requirement by including boundary checks immediately before any access to arrays.

The code below shows an example of a boundary check being performed. Although not shown, the declaration of the array (*array_ex*) specifies the number *ARRAY_LEN* as its length. Thus, the condition identifies whether the index is within the boundaries of the array, not executing the following lines if it is not.

```
static void check_ex_ep(..., struct context *ctx, struct scratch *s,
    ...) {
     if(s->index < ARRAY_LEN)
         return;
     ctx->array_ex[s->index] = 1;
     ...
}
```

Chapter 5

Evaluation

This chapter evaluates MTP's language expressiveness, the XDP back-end in terms of performance and capacity of representing different protocols, and XDP as a framework for transport layer programming. Moreover, these are the questions that this chapter will answer:

- Is MTP able to represent various transport layer algorithms? Is it expressive enough?
- Does MTP simplify the process of programming transport layer algorithms when compared to an implementation in XDP?
- Does XDP achieve good results in terms of performance when implementing transport protocols?
- Should XDP be used as a tool for transport programming?

5.1 Evaluating MTP

This section evaluates MTP in two forms. First, it will present two examples of protocols implemented in MTP and what characteristics of the language each tried to highlight. Second, this section will compare the development effort in the number of lines of code to develop the examples in MTP and the XDP back-end.

5.1.1 Expressiveness

To guarantee that MTP provides an expressive language capable of representing different transport layer protocols, we implemented two protocols, being TCP [30] and RoCEv2 [20, 69]. These protocols were chosen due to their different characteristics, such as how each handles congestion control and how the sliding windows are utilized. These characteristics made us understand in further detail the basic needs of transport layer protocol implementations, allowing us to gradually tune the language's constructs and syntax to better serve transport programmability.

TCP: Due to the relevancy of TCP in the transport layer, this protocol was the first implemented and was the basis for the simplest constructs of MTP's language and model. As such, this example implements a simplified version of TCP, following the transmitting and acknowledgement algorithms from RFC 9293 [30], the congestion control algorithms from RFC 5681 [18], and the retransmission timer computation from RFC 6298 [63].

It is worth highlighting the declaration of a few elements of the TCP example. In this implementation, a single application event type declares send requests from the application are handled; two network event types are used to represent packets arriving from the network that can be data (at the receiver side) or acknowledgements (at the sender side); and it contains one timer event that represents timeouts from missing acknowledgements. Meanwhile, the flow context has some relevant information for data transmission, congestion control, packet retransmission, and the declaration of important elements of the MTP language, such as a timer and a sliding window.

The TCP example highlights some of the most important constructs of MTP, being packet segmentation, timers, and a sliding window. Instead of requiring the user to implement loops to create packets iteratively, MTP provides functions that specify how segmentation should be executed. Specifically for TCP, the segmentation function call determines that only the sequence number of the packets will change (according to the length of the previous packet's data). The example utilizes the timer-related functions of MTP to start, cancel, and set events for the timer that keeps track of missing acknowledgements. The sliding window is used by the receiver side of TCP to monitor the sequence numbers that arrived, which would take several lines of code and considerable effort without this construct. Unfortunately, due to the simplicity of the TCP example, the reusability of event processors was not utilized in depth in this implementation.

RoCEv2: The second example implemented in MTP was RoCEv2, chosen due to its different characteristics to TCP and how it explores other aspects of MTP language. These characteristics include how congestion control is handled, and the interaction with the

transport layer through work requests. Since the documentation of RoCEv2 is not widely available, we followed the specification provided by InfiniBand Architecture Specification [69] book, which is fairly old. Even so, the core principles of the protocol should not have changed considerably and they match with concepts brought by other RoCEv2-related documents and work [20, 39].

Since this protocol provides many more operations other than simply sending data (e.g. write, read, receive, and atomic operations), RoCEv2 contains several more event types than TCP. Each RDMA operation, i.e. WQE, has an equivalent application event type, as well as each kind of received packet (e.g. data to write, read data, atomic operation request) has its own network event type. Timer events are also utilized in congestion control to represent DCQCN-related [39] events and for missing acknowledgement timeouts. Finally, this example employs program events to execute auxiliary steps of rate calculation from DCQCN.

While TCP has flows and each contains its own context, RoCEv2 separates its different connections through queue pairs (QP). The design philosophy followed in this example was to store packets generated by the application events (WQE) into a list in the QP's context. This list is accessed by an event processor reused by multiple EP chains, whose only responsibility is to transmit the newly generated packets. Moreover, this example highlights the modularity and reusability of EPs. These packets are removed from the list only when the acknowledgement (for non-ack-requesting WQEs) or the requested data arrives.

Although this design philosophy can be well implemented in MTP, as Section 5.2.2 will more detail, this cannot be well represented in XDP. This is because XDP does not allow dynamic memory allocation for data structures; which leads to the impracticability of declaring lists of variable length. Thus, this indicates that lists may not be an ideal type to have in MTP, as other targets might not allow dynamic memory allocation.

RoCEv2 also highlights some of the constructs of MTP's language. Segmentation is applied during the creation of the packets, which is especially useful because two values must change for each packet created: the PSN, which is a unique identifier for packets; and the opcode that identifies the RDMA operation and whether it is the first, middle, or last packet of the WQE. Through the segmentation function, the programmer can avoid writing several lines of code specifying the PSN and the policy that decides the packet's opcode.

Another relevant construct used for RoCEv2 implementation is timers, employed for missing acknowledgement timeouts and, more importantly, DCQCN events. While TCP's congestion control is based on the congestion window size, RoCEv2's congestion control is

dictated by DCQCN, a rate-based protocol. As such, the special rate event type comes in handy to notify the Transport-to-Network interface of the rate at which packets should be transmitted.

5.1.2 Reducing development effort

Simplifying the implementation of transport layer protocols by increasing the abstraction level and improving modularization is one of the main goals of the MTP framework. As such, evaluating the effort taken to write MTP programs is necessary to understand how useful this framework can be. For this purpose, we compare the number of lines of code taken to write the protocol implementations (TCP and RoCEv2) in MTP with the number of lines of their translation to the XDP back-end.

If users were to implement a protocol directly in XDP, they would need to handle several complex elements that the XDP back-end already covers. We can highlight some of them: handling AF_XDP, which is a considerably difficult tool to use since it requires precise control over the ring buffers it contains; packet mutation, as its complexity comes from the many operations each mutation requires, i.e. out-of-boundary memory access checks and memory copies; eBPF maps and how they are accessed; and, most importantly, how loops that iterate over a variable amount needs to be implemented as bpf_loop calls. All these difficulties result in an environment that requires deep analysis to understand how each element works. Additionally, as Section 5.3 presents in more detail, the poor documentation available for libbpf and libxdp libraries considerably increases the development effort of programming XDP.

TCP: Table 5.1 presents the number of lines of code taken for the MTP program and the XDP back-end for TCP implementation. The first row represents the MTP program, defined by the user. The second row consists of the XDP code, which is the result of MTP user-defined compilation. Lastly, the third row represents the fixed section of the XDP back-end (from both user space and XDP sides), i.e. environment configuration, AF_XDP handling, built-in functions and data structures declarations, interfaces, and other sections of code. We can highlight that MTP reduced the development effort for this implementation on XDP from 3418 to 504 lines of code.

RoCEv2: Table 5.1 presents the number of lines of code taken for the MTP program and the XDP back-end for RoCEv2 implementation, following the same pattern in its rows as TCP's table. The great difference in the number of lines between the MTP and the XDP compiled code comes mostly from the EPs shared between different chains of EPs. As explained in Section 4.2.1, since the XDP verifier presented errors while handling *void*

Program	Language	Created by	Number of loc
TCP	MTP	User	504
TCP	XDP	Automatically by the compiler	779
Target	XDP	Fixed and manually tweaked	2639

Table 5.1: Reduction in development effort enabled by MTP for the TCP implementation in XDP

Program	Language	Created by	Number of loc
RoCEv2	MTP	User	1387
RoCEv2	XDP	Automatically by the compiler	2152
Target	XDP	Fixed and manually tweaked	2740

Table 5.2: Reduction in development effort enabled by MTP for the RoCEv2 implementation in XDP

pointers, we chose to avoid utilizing them. So, instead of having a single declaration of an EP whose argument is the event represented by a *void* pointer, the code generator creates copies of the same function, each receiving an event of a different type. The same may also happen with loops used in shared EPs. This is because they are translated to bpf_loop callback functions that may access the events and, therefore, receive the events as arguments. Finally, we can highlight that MTP reduced the development effort for RoCEv2 implementation on XDP from 4892 to 1387 lines of code.

Table 5.1 and Table 5.2 differ in the number of lines of code in the fixed part (target specific). This is because the interfaces are not translated currently. Therefore, changes must be manually applied in some areas of the fixed code, especially sections dedicated to parsing the mutated packets redirected from XDP and creating the new packets sent through AF_XDP.

5.2 Evaluating XDP back-end

This section will evaluate the XDP back-end code generated for both TCP and RoCEv2 implementations. The first aspect we will examine is the generated code's capacity to pass the eBPF verifier analysis. As the examples will present, getting the approval of the verifier is a significantly difficult problem. This is due to how the verifier can detect any possibility of out-of-bound memory accesses, unbounded loops, and, most importantly, trespassing the maximum size of the eBPF call stack. Moreover, when any of these cases

(and possibly others) are detected, the program cannot be loaded to the hook point. In general, both out-of-bound memory accesses and unbounded loops are well handled by the code generator developed for this thesis, described in Chapter 4. However, the maximum eBPF call stack is still an ongoing challenge for us. The second aspect of the XDP back-end to be evaluated is its performance and how accurately it follows the expected behaviour of the implemented protocol.

The experiments presented in this section were performed using two servers connected with 100Gbps links to an Ethernet switch. Each server contains 96 Intel Xeon Gold 6342 (2.80GHz) cores across two NUMA nodes, 250GB of RAM, and an NVIDIA ConnectX-6 NIC. We also developed two scripts to set up the cores that process the packets on both sides. The first script utilizes the Intel Flow Director [26] to specify filters to redirect packets to a specific NIC RX queue based on its flow (e.g. in a TCP packet, the IP address and the ports). Meanwhile, the second script sets the affinity of the Interrupt Request (IRQ) of each NIC's RX queue. Thus, mapping one CPU to process packets from one RX queue. After running both scripts, we have an environment where we pin flows to cores. This allows us to analyze the performance of the XDP back-end in each core separately and have control over which cores are executing which flows.

5.2.1 TCP implementation

The TCP example is relatively simple to pass the eBPF verifier. This implementation only contains a few event types, having a small number of chains of EPs. There are no EPs shared between chains, which results in a code that does not need multiple copies of the same EP. Additionally, all EPs are relatively simple and do not instantiate many variables and large structures, a relevant characteristic to avoid reaching the maximum eBPF stack size. The main difficulty in passing the verifier was adding out-of-bounds memory checks and handling the sliding window. The latter required two different bpf_loops to be added into the code, each responsible for adding and removing elements to the array that represents the window.

Figure 5.1 presents the progression of the TCP congestion window. In this setup, the two communicating servers have the program loaded, where the sender side transmits packets as its congestion window allows; meanwhile, the receiver responds with acknowledgements and artificially drops packets every hundred. The congestion window follows the expected behaviour by following slow start, fast recovery and retransmit (with artificial inflation and deflation happening between receiving the third duplicate ACK and the lost segment's ACK), and the additive increase of congestion avoidance.

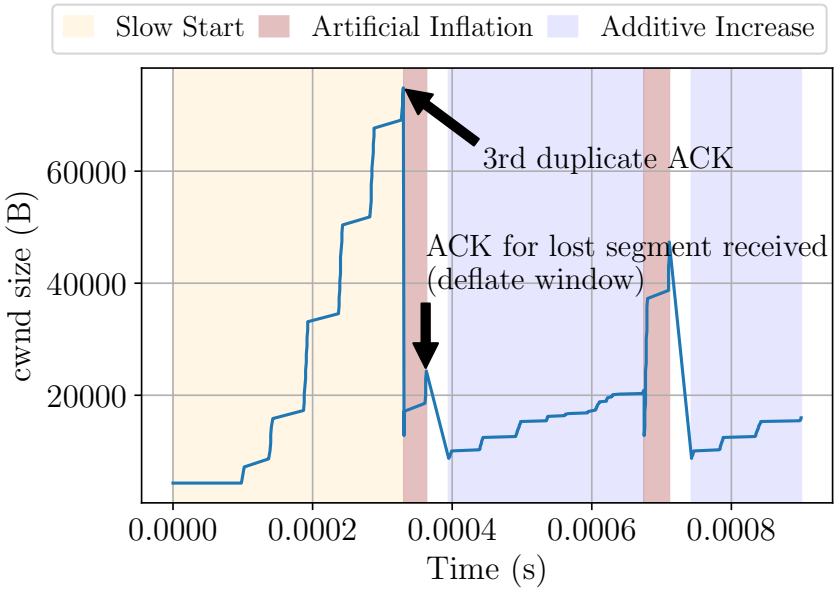


Figure 5.1: The progression of TCP congestion window.

Moreover, Figure 5.2 shows the total throughput of the TCP implementation in the XDP back-end for 1 to 16 cores. Since we have control over the flows that each core processes, we pinned the minimum amount of flows that saturate the core's event processing throughput, being 32 flows. The maximum throughput achieved was around 80Gbps with 16 cores, with an average latency of 250ns to process the received packet's event and other (at most) 10 enqueued events.

Since our flows are pinned to cores, the throughput scales well as the number of cores is increased. However, the throughput does not scale linearly due to the contention present when accessing the eBPF map of maps. As Section 3.3 shows, non-per-CPU maps have contention when multiple cores try to access the same map concurrently. This issue could be mitigated by using per-CPU maps instead of the current maps in the XDP back-end. Nevertheless, the map of maps type (used to represent the event queues of all flows) lacks a per-CPU variation, preventing us from utilizing per-CPU maps.

5.2.2 RoCEv2 implementation

MTP programs can be considerably complex. It may contain many different event types and complex event processors. Moreover, these event processors may have loops, which

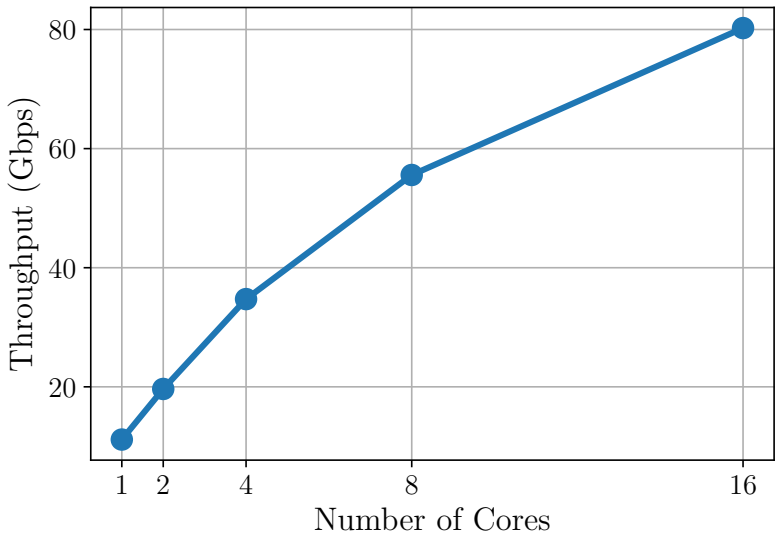


Figure 5.2: Total throughput with increasing number of cores.

are translated into bpf_loop calls. Since some of these loops must access variables declared outside of them, these variables need to be passed as arguments (as a single structure). However, if the loops access many variables, their arguments can be quite large structures. Additionally, since nested loops are allowed, nested bpf_loop calls can happen. Therefore, as large structures are instantiated and bpf_loop calls are stacked to the eBPF stack, we can easily reach the maximum limit of the stack (512 bytes).

For a long time, this was the case for RoCEv2 implementation. The code generator was able to translate the MTP example into XDP back-end code that did not have syntax and semantic errors, as well as no out-of-bounds memory accesses and unbounded loops (checked by the eBPF verifier). However, the verifier used to announce that the maximum stack size was trespassed in many of the event processors. To solve this issue, we spent considerable time analyzing the error messages returned by the verifier. These messages show an estimate of the stack size at the point the limit is reached. Nevertheless, after a while, we started noticing that the number returned was behaving strangely. For example, in some of the EPs that reached the maximum stack size, we commented out all of its internal code, and the same error was returned with the same stack size estimate.

Since the stack size returned behaved strangely for some event processors, independently of what was declared in them (commenting on them or not), we chose to investigate the earlier stages of the XDP back-end. In these earlier stages, after the network packet arrives, it is parsed, and the network event is generated and promptly processed. After this network

event finishes executing, the program enters a loop that decides the next event type to be executed, dequeues the event from the chosen type's queue, and processes it.

Originally, we intended to loop a variable amount of times (through bpf_loop). This variable number would be decided by an algorithm that sets the rate of processed events per arriving packet. However, we chose to loop over a constant number of times for the sake of simplicity and to avoid another overhead. Even so, we left bpf_loop as the form of iteration, as it did not affect the correctness and execution of the TCP implementation (the first implemented). Nevertheless, after switching the bpf_loop with a regular loop iterating over a constant, the RoCEv2 example smoothly passed the eBPF verifier. It is still unclear why this problem happened, as there is little documentation available regarding this function and how it is implemented. But we aim to investigate XDP in further detail as future work.

After finally passing the eBPF verifier, we were able to test the RoCEv2 example. But, since this is a large code, we chose to start testing and debugging solely the write RDMA operation. As mentioned in Section 5.1.1, this implementation stores the headers of the packets generated into a list and are only removed when its acknowledgement arrives. Since XDP does not allow dynamic memory allocation, XDP does not allow the declaration of a list that grows indefinitely as new elements are appended. The closest representations available in XDP are arrays or eBPF map arrays. And we chose the first, as accessing eBPF maps can add a severe overhead. Even so, since arrays have a maximum size, there is a limit as to how many packets can be sent before the queue becomes full. Thus, in the current implementation, the write operation can send a small amount of packets with low throughput. However, as future work, we want to explore solutions that do not involve saving packets into an array and debug the entire implementation.

5.3 XDP as a transport layer framework

eBPF is a tool created to simplify the process of adding user-defined code to Kernel hook points. More specifically, XDP allows new code to be added to the NIC driver. This characteristic of XDP makes it a tool that naturally could be used for packet processing/preprocessing. However, that does not mean that this framework is capable of doing all sorts of operations (including transport algorithms), due to its restrictions and other aspects. Therefore, this section will present the pros and cons of XDP and analyze whether this should be utilized to implement the transport layer.

5.3.1 Strengths of XDP

This subsection presents the pros of using XDP for transport programming:

Adding code to Kernel: Working with the Linux Kernel can be considerably difficult. As Section 1.2 showed, the code that implements the Kernel is quite complex. By just analyzing the codebase related to networking, each file contains hundreds of functions and thousands of lines. Additionally, adding new code to the Kernel can be a complex process, since wrongfully implemented sections of code may generate significant problems for the system (out-of-bound memory access, infinite loops). However, because the network stack is implemented in the Linux Kernel, any major changes to the transport layer must be implemented in this code base. Moreover, eBPF and XDP considerably facilitate the creation of transport layer code, by avoiding the direct interaction with Kernel code and doing it safely.

Packet-driven nature: XDP programs have a packet-driven nature, i.e. as packets arrive at the NIC (an event), the XDP thread is triggered to execute it. This partially meets with the nature of transport layer protocols, which are event-driven. However, the fact that the thread only exists if a packet arrives and while the packet is processed is limiting. For example, having a secondary thread to run in parallel and execute rate-limiting processing could be interesting for transport programming. We overcame such limitations by processing multiple events per arriving packet, by using timer threads and, when strictly necessary, calling xdp_program__test_run syscall.

The characteristics of eBPF maps: These data structures from XDP are very useful for transport programming. eBPF maps are stateful, i.e. they keep their state (the data stored) between executions of the same XDP program. This characteristic matches the needs of transport layer programming, as transport protocols are also stateful. Additionally, maps are useful tools for the communication between user space and the XDP side. For example, the XDP back-end used maps as the means for transmitting application requests to be processed by the XDP program. However, eBPF should be used wisely since it adds significant overhead from accessing them.

Bypassing Kernel stack: One of the objectives of this thesis is allowing transport layer programmability in XDP. As such, it is essential to bypass the Kernel stack. This is because it would add overhead to packet processing whose transport protocol can already be processed in the Kernel stack. Meanwhile, for new protocols, bypassing the Kernel stack is strictly necessary, as the Kernel stack would not be able to process it. Conveniently, XDP allows bypassing the Kernel stack through AF_XDP socket.

5.3.2 Weaknesses of XDP

This subsection presents the cons of using XDP for transport programming:

One packet in, one packet out nature: An XDP program is only executed once a packet arrives. By using the return code XDP_TX, the program can transmit the packet out via the same NIC it came from. However, only one packet can be transmitted out. This is a considerable limiting factor of XDP, since transport protocols can send multiple packets from a single event. To amend that, we had to follow the unusual path of redirecting a mutated packet to user space via the AF_XDP socket, parse it, create new packets, and send them out through AF_XDP. Although this solution functions well, it adds overhead as the packets keep transitioning between XDP and user space. Thus, it would be desirable to have a way to transmit multiple packets out from the XDP program itself.

eBPF map contention: In multi-core systems, each core handles the IRQ from a set of the NIC's RX queues. As such, a transport protocol implementation in XDP must consider that the eBPF maps can be concurrently accessed by cores. However, as mentioned in Section 3.3, there is contention when multiple processors access the same eBPF map. This can be avoided by utilizing per-CPU variation of maps, where each core contains its own separate instance of the map. Nevertheless, only a few of them have a per-CPU version. Therefore, XDP could greatly benefit if more per-CPU variations were added, mostly importantly for the map of maps type.

eBPF stack limitation: Section 5.2.2 presented how limiting the eBPF stack can be. Even if we were able to have both examples pass the eBPF verifier, the 512-byte long stack space can considerably restrict programs. Nevertheless, eBPF contains tail call [7] (see Tail calls section) that can assist in handling this issue. This tool allows eBPF functions to reutilize the stack space occupied by the caller function. However, each of the tail call functions is restricted to a stack of 256 bytes. We leave exploring tail calls in the XDP back-end as future work.

Inconsistent documentation: Although it is not a specific problem for transport programming, the available documentation on XDP and related libraries lacks full coverage and detail of the framework. New additions and updates are constantly included in XDP and its libraries. However, their documentation does not receive updates as fast as the code, both for new inclusions and for updates to older documents. For a few reasons, this makes XDP a somewhat difficult framework to maintain a codebase for the long term. First, undocumented changes can potentially harm the code, while the programmer is unaware of why. Second, new useful additions may be hidden and, therefore, unutilized. Third, without proper documentation, XDP code cannot evolve at the same pace the tool and its libraries evolve.

As an example, we can highlight how we discovered how to use xdp_program__test_run. Originally, a syscall called bpf_prog_run (or bpf_prog_test_run) was used to simulate the XDP program at user space. But, on a recent version of the Kernel, a new element called xdp_dispatcher was added to allow multiple XDP programs to be loaded to the hook point. And, when the XDP program is attached to the xdp_dispatcher, its type changes from "xdp" to "ext". However, bpf_prog_run does not support the "ext" type. After a few days spent trying to avoid loading xpd_dispatcher and, instead, loading the XDP program directly, we were able to accomplish it.

However, we found a new error message, announcing that the dispatcher was required to utilize AF_XDP, an essential tool for our back-end. After hours spent trying to find a way to allow both bpf_prog_run and AF_XDP and unsuccessfully doing so, we chose to ask a question [4] on the xdp-tutorial repository [13]. And, to our surprise, a Principal Kernel Engineer at Red Hat and one of the main engineers of XDP, Toke Høiland-Jørgensen, answered our question with a simple solution: use xdp_program__test_run. However, the issue is that no documentation was available detailing the current problem with bpf_prog_run and why use xdp_program__test_run instead. Only a release note [12] of libxdp briefly mentioned the purpose of the new function. Therefore, this example only exemplifies how improving the documentation for both eBPF and, more importantly, XDP would be highly important.

5.3.3 Final considerations

With the arguments presented in the previous subsections, it is clear that XDP can be a good framework for transport layer programming. All the good aspects shown are characteristics and elements of the tool that match (or compare to) what is necessary to allow the implementation of transport algorithms. Meanwhile, for most of the cons presented, we found ways to handle them or, at least, minimize their problems. Additionally, we suggest a few possible additions for XDP that could significantly improve XDP performance and programmability, both for the transport layer specifically and for the tool itself.

Moreover, as new Kernel versions are released, new updates and tools are also added to eBPF, XDP, and libraries. This is a good sign that XDP can meet its full potential as a transport layer programming framework in the future. For instance, in the recently released version 6.9 of the Kernel, a new tool called bpf_arena was introduced to eBPF [71, 16, 7]. This tool consists of another kind of memory space that both sides, eBPF and user space can access. In this area, user-defined data structures can be declared and used from both sides.

Meanwhile, regarding documentation, eBPF Docs [7] is a good source of information for the many elements present in eBPF and the many subtypes of programs it has. Moreover, this documentation has recently been receiving several updates, adding new interesting and relevant details of eBPF for new and older tools. This might indicate the documentation of eBPF, XDP, and their libraries can be significantly improved in the near future.

Chapter 6

Conclusion

The transport layer is constantly changing. These changes are caused by the escalation of traffic, the appearance of new kinds of workloads, and the utilization of new hardware network infrastructures. However, developing transport layer protocols can be challenging, due to many aspects present within the network stack and the environment they execute in. This motivated us to create MTP, a transport layer programming framework that simplifies the development of such algorithms, by providing a high level of abstraction and increasing the modularization. Additionally, we created an XDP back-end to which the MTP are compiled and executed.

6.1 Summary of contributions

MTP (Chapter 2) follows an event-driven logical model. In this design, requests from the neighbouring layers (i.e. requests from application and packets from the network) are parsed into events by incoming interfaces. Moreover, these events follow a scheduling policy, to decide which will be processed next. During their execution, events can be processed by a chain of modular user-defined modules, called event processors. As events are processed, data might need to be transmitted to the application or new packets can be generated, both handled by outgoing interfaces. Additionally, MTP provides a language for the declaration of the user-defined modules, while they are implicitly linked following the framework's model.

XDP back-end (Chapter 3) was created to serve as the first target to which MTP programs are compiled. This back-end is separated into two major parts, the user space

and XDP. The user space creates and enqueues application events, parses redirected packets and transmits new packets out. Meanwhile, the XDP side receives and parses packets from the network, process events, and redirects mutated packets to the user space. Considerable effort was spent making design decisions to improve performance, such as choosing optimal maps. Additionally, there are a few design aspects that can be considered somewhat unnatural, such as redirecting packets from XDP to user space for new packets to be created and sent out. However, this was motivated by the limitations imposed by XDP's characteristics and nature.

A code generator (Chapter 4) was created to translate MTP programs into code that can be plugged into the XDP back-end. Translating from one language to another was relatively simple since both are C-based. Nevertheless, some characteristics of XDP made the translation process slightly more complex, such as how it handles loops and void pointers.

We evaluated these contributions in three ways (Chapter 5). First, to analyze the expressiveness of MTP's language, we developed two transport layer protocols, TCP and RoCEv2. Although both were well represented in MTP, we noticed that the list type used in RoCEv2 may not be ideal for most targets. Second, we evaluated the performances of the protocol implementations after successfully translating them from MTP to XDP back-end. Both passed the syntax and semantic analysis of the eBPF compiler and, most importantly, were accepted by the eBPF verifier. And, given two servers connected with 100Gbps links to an Ethernet switch, the TCP implementation reached 80Gbps throughput using 16 cores. Meanwhile, the RoCEv2 write operation can send a small number of packets with a low throughput. Finally, we evaluated if XDP can be considered a good framework for transport programming. After analyzing its strengths and weaknesses, we concluded that XDP already presents good features and, as new updates are being released, it has the prospect of being a great tool for transport programming.

6.2 Future Work

There are many possible future directions available to explore, that being:

Extend XDP back-end: across this thesis, many possible extensions to the XDP back-end were mentioned. First, the user space side has yet much to be explored. Since our main focus was on XDP, the design of the user space program was left aside for a long time. Thus, exploring other designs in terms of multithreading and application request dispatching could be valuable. Second, because not all map types are available

in a per-CPU variation, the XDP back-end only utilizes non-per-CPU maps. However, using a combination of both map variations could significantly reduce core contention and, therefore, improve the performance. Third, rate limiting still needs to be implemented in the XDP back-end. We aim to do this at user space, by controlling the number of packets created from those at the RX queue of AF_XDP. Fourth, due to the limitations of XDP in terms of dynamically allocated data structures, the RoCEv2 example needs to be reworked to avoid using list type from MTP. Finally, we aim to explore tail calls to avoid reaching the eBPF maximum stack length to, potentially, allow the definition of even more complex MTP programs.

Explore the details of XDP: While developing the XDP back-end, we noticed strange problems regarding the eBPF compiler and verifier. These problems include the unusual way the maximum eBPF stack size is estimated and how seemingly safe memory accesses resulted in out-of-bounds access errors. We believe that the rigid restrictions imposed to assure the safety of the Kernel make the eBPF verifier too course-grained to distinguish minor details between safe and unsafe code. Therefore, one of our future goals is to explore the eBPF/XDP ecosystem itself to understand how its background operates and to write more robust code.

Handling debugging messages in MTP: when developing code in any language, users expect to receive meaningful error messages in case any bugs appear. And the same goes for MTP. However, we have another layer of complexity in MTP's case. This is because the target to which the MTP code is translated has its own error messages that are only meaningful for the target itself. Therefore, since the MTP programmer may not know how the target works in the background, MTP must return the target's error message in a format that is meaningful to MTP. For example, the many eBPF verifier error messages (i.e. out-of-bounds memory access, unbounded loops, stack length) are unique to eBPF and they cannot be simply understood by an MTP programmer oblivious to XDP. Thus, we aim to add a debugging module to MTP, which converts errors from the target into meaningful feedback in the MTP language.

References

[1] 802.1qbb – priority-based flow control. Available: https://1.ieee802.org/dcb/802-1qbb/. Accessed: Nov. 22, 2024.

[2] About bpf xdp adjust tail. Available: https://github.com/xdp-project/xdp-tutorial/issues/234#issuecomment-882115985. Accessed: Nov. 24, 2024.

[3] Antlr. Available: https://www.antlr.org/. Accessed: Oct. 8, 2024.

[4] Avoid loading and attaching xdp dispatcher. Available: https://github.com/xdp-project/xdp-tutorial/issues/435. Accessed: Nov. 27, 2024.

[5] bcc. Available: https://github.com/iovisor/bcc, note = Accessed: Oct. 29, 2024.

[6] eBPF. Available: https://ebpf.io/. Accessed: Oct. 7, 2024.

[7] ebpf docs. Available: https://docs.ebpf.io/. Accessed: Oct. 17, 2024.

[8] eBPF Documentation. Available: https://ebpf.io/what-is-ebpf/. Accessed: Oct. 7, 2024.

[9] libbpf. Available: https://github.com/libbpf/libbpf. Accessed: Oct. 9, 2024.

[10] P4 open source programming language. Available: https://p4.org/. Accessed: Oct. 22, 2024.

[11] xdp-tools. Available: https://github.com/xdp-project/xdp-tools. Accessed: Oct. 10, 2024.

[12] xdp-tools release v1.3.0. Available: https://github.com/xdp-project/xdp-tools/releases/tag/v1.3.0. Accessed: Nov. 27, 2024.

[13] xdp-tutorial. Available: https://github.com/xdp-project/xdp-tutorial. Accessed: Nov. 27, 2024.

[14] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. 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*, pages 632–647, 2020.

[15] Marcelo Abranches, Oliver Michel, Eric Keller, and Stefan Schmid. Efficient network monitoring applications in the kernel with ebpf and xdp. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 28–34, 2021.

[16] Daroc Alden. A proposal for shared memory in bpf programs [lwn.net]. https://lwn.net/Articles/961941/, 2024. (Accessed: Nov. 27, 2024).

[17] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[18] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. Technical report, 2009.

[19] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 29–43, 2016.

[20] Infiniband Trade Association. Supplement to infiniband architecture specification volume 1 release 1.2.1 annex a17: Rocev2, 2014.

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[22] Lawrence Brakmo. Tcp-bpf: Programmatically tuning tcp behavior through bpf. *Proc. NetDev*, 2:1–5, 2017.

[23] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.

[24] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, 1974.

[25] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82, 2009.

[26] Intel Corporation. Introduction to intel® ethernet flow director and memcached performance. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf. (Accessed: Nov. 24, 2024).

[27] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th international conference on emerging networking experiments and technologies*, pages 160–166, 2017.

[28] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. {PCC}: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, 2015.

[29] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. {PCC} vivace:{Online-Learning} congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018.

[30] Wesley Eddy. Transmission Control Protocol (TCP). RFC 9293, August 2022.

[31] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM Sigplan Notices*, 46(9):279–291, 2011.

[32] Linux Foundation. Data plane development kit. Available: https://www.dpdk.org/. Accessed: Oct. 7, 2024.

[33] Alexander Frommgen, Tobias Erbshäußer, Alejandro Buchmann, Torsten Zimmermann, and Klaus Wehrle. Remp tcp: Low latency multipath tcp. In *2016 IEEE international conference on communications (ICC)*, pages 1–7. IEEE, 2016.

[34] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–12, 2015.

[35] Pablo Garrido, Isabel Sanchez, Simone Ferlin, Ramon Aguero, and Ozgu Alay. rquic: Integrating fec with quic for robust wireless communications. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2019.

[36] Pierre Geurts, Ibtissam El Khayat, and Guy Leduc. A machine learning approach to improve congestion control over wireless computer networks. In *Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 383–386. IEEE, 2004.

[37] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. {BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501, 2021.

[38] Paul Grun. Introduction to infiniband for end users. https://network.nvidia.com/pdf/whitepapers/Intro_to_IB_for_End_Users.pdf, 2010. (Accessed: Nov. 25, 2024).

[39] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[40] Kjell H. Lock-Free Single-Producer - Single Consumer Circular Queue. Available: https://www.codeproject.com/Articles/43510/Lock-Free-Single-Producer-Single-Consumer-Circular#heading_acquire_release_model. Accessed: Oct. 14, 2024.

[41] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[42] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[43] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[44] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, August 1988.

[45] Mathieu Jadin, Quentin De Coninck, Louis Navarre, Michael Schapira, and Olivier Bonaventure. Leveraging ebpf to make tcp path-aware. *IEEE Transactions on Network and Service Management*, 19(3):2827–2838, 2022.

[46] Cheng Jin, D.X. Wei, and S.H. Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4, pages 2490–2501 vol.4, 2004.

[47] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[48] Madhan Kanagarathinam, Syed Faraz Hasan, Sujith Rengan, Sukhdeep Singh, Gunjan Kumar Choudhary, Nawab Muhammad Faseeh Qureshi, and Hoejoo Lee. Enhanced quic protocol for transferring time-sensitive data. In *2022 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2022.

[49] Magnus Karlsson and Björn Töpel. The path to dpdk speeds for af xdp. In *Linux Plumbers Conference*, volume 37, page 38, 2018.

[50] Eddie Kohler, Mark Handley, and Sally Floyd. Designing dccp: Congestion control without reliability. *ACM SIGCOMM Computer Communication Review*, 36(4):27–38, 2006.

[51] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[53] Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed Meleis. Qtcp: Adaptive congestion control with reinforcement learning. *IEEE Transactions on Network Science and Engineering*, 6(3):445–458, 2018.

[54] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. {Multi-Path} transport for {RDMA} in datacenters. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 357–371, 2018.

[55] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.

[56] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černỳ. Event-driven network programming. *ACM SIGPLAN Notices*, 51(6):369–385, 2016.

[57] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for ebpf-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151, 2021.

[58] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[59] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. *Acm sigplan notices*, 47(1):217–230, 2012.

[60] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[61] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 30–43, 2018.

[62] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. Dynamic tcp initial windows and congestion control schemes

through reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1231–1247, 2019.

[63] Vern Paxson, Mark Allman, Jerry Chu, and Matt Sargent. Computing tcp's retransmission timer. Technical report, 2011.

[64] Larry Peterson, Bruce Davie, and Lawrence Brakmo. *TCP Congestion Control: A Systems Approach*. Systems Approach LLC, Tucson, AZ, 2023.

[65] Gregory F Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, 42(617-632):10, 2001.

[66] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. {FlowBlaze}: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, 2019.

[67] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 780–794, New York, NY, USA, 2022. Association for Computing Machinery.

[68] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.

[69] Tom. Shanley, Joe. Winkles, and Inc. MindShare. *InfiniBand network architecture*. PC system architecture series. Addison-Wesley, Boston, MA, 1st edition edition, 2003.

[70] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 213–226, 2014.

[71] Alexei Starovoitov. bpf: Introduce bpf arena. [lwn.net]. https://lwn.net/Articles/961594/, 2024. (Accessed: Nov. 27, 2024).

[72] Wright Stevens. Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Technical report, 1997.

[73] Kun Tan, Jingmin Song, Qian Zhang, and Murad Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proceedings-IEEE INFOCOM*, 2006.

[74] Mellanox External WORD Template. Infiniband faq rev 1.3. https://network.nvidia.com/pdf/whitepapers/InfiniBandFAQ_FQ_100.pdf, 2014. (Accessed: Nov. 25, 2024).

[75] Linus Torvalds. Linux kernel source tree. Available: https://github.com/torvalds/linux, 2024. Accessed: Oct. 6, 2024.

[76] Viet-Hoang Tran and Olivier Bonaventure. Making the linux tcp stack more extensible with ebpf. In *Proc. of the Netdev 0x13, Technical Conference on Linux Networking*, 2019.

[77] Tobias Viernickel, Alexander Froemmgen, Amr Rizk, Boris Koldehofe, and Ralf Steinmetz. Multipath quic: A deployable multipath transport protocol. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2018.

[78] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review*, 43(4):123–134, 2013.

[79] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. Ictcp: Incast congestion control for tcp in data center networks. In *Proceedings of the 6th International COnference*, pages 1–12, 2010.

[80] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *IEEE INFOCOM 2004*, volume 4, pages 2514–2524. IEEE, 2004.

[81] Yue-Dong Xu, Zhen-Yu Wang, and Hua Wang. Ared: a novel adaptive congestion controller. In *2005 International Conference on Machine Learning and Cybernetics*, volume 2, pages 708–714. IEEE, 2005.

[82] Zhiyuan Xu, Jian Tang, Chengxiang Yin, Yanzhi Wang, and Guoliang Xue. Experience-driven congestion control: When multi-path tcp meets deep reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1325–1336, 2019.

[83] Ticao Zhang and Shiwen Mao. Machine learning for end-to-end congestion control. *IEEE Communications Magazine*, 58(6):52–57, 2020.

[84] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, et al. Xlink: Qoe-driven multi-path quic transport in large-scale video services. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 418–432, 2021.

[85] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with {eBPF}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, 2023.

[86] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.