Accelerating QUIC with XDP

Minu Föger, Marcel Kempf *
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: foeger@in.tum.de, kempfm@net.in.tum.de

***Abstract*—After the standardization of the new transport protocol QUIC in 2021, QUIC implementations are increasingly optimizing for performance. Multiple implementations have started to utilize the eXpress Data Path (XDP) technology to speed up QUIC packet processing. XDP is a high-speed network data path in the Linux kernel based on the eBPF virtual machine. It allows bypassing the in-kernel network stack. A similar technique is being developed for Microsoft Windows. Due to the simplicity of UDP processing in QUIC, XDP shows great promise for reducing kernel overhead in QUIC implementations.**

***Index Terms*—Linux, XDP, QUIC**

1. Introduction

The transport protocol QUIC has seen increasing adoption on the Internet, with most modern browsers supporting QUIC [1]–[3] and with major web applications being serviced via HTTP/2-over-QUIC or HTTP/3 (e.g. most Google web applications) [4].

With the rising adoption of QUIC, major QUIC implementations have become more mature and are investing in performance optimizations. One potential optimization would be to use kernel bypass mechanisms to speed up lower-layer network packet processing. Most modern operating systems (like Linux or Windows) have generic in-kernel network stacks that usually provide transport layer abstractions (most prominently the Berkeley socket abstraction [5]). These network stacks are heavily optimized but are typically designed for high flexibility and generality. However, in specific cases this generality can result in inefficiencies [6].

QUIC is built on top of the old UDP protocol, which is typically implemented inside the kernel network stack (exposed via the socket API). However, QUIC is designed to require only minimal packet processing at lower layers, e.g. IP fragmentation is prohibited by the standard [7]. Therefore, it would be more efficient to perform QUIC-specific UDP payload extraction on incoming QUIC traffic than to process it via the generic kernel network stack. QUIC implementations would thus benefit from kernel bypass mechanisms. For this reason, multiple QUIC implementations (*s2n-quic* [8], *MsQuic* [9]) are currently starting to utilize the XDP kernel bypass mechanism to reduce the kernel overhead of their implementations.

There exist multiple techniques for high-speed packet processing that gain performance by bypassing the kernel network stack. One relatively new mechanism is the XDP/AF_XDP mechanism [6] that shall be further described in Section 2.

A popular alternative is DPDK (Data Plane Development Kit), a project facilitating high performance network I/O via specialized user-space Poll-Mode Drivers that poll the NIC directly and thus bypass the kernel network stack [10]. Initially developed by Intel in 2010 and targeting only the Linux kernel, it today is hosted collaboratively under the Linux Foundation and is also ported to other operating systems like Windows. DPDK has been optimized quite heavily over the years, often by taking advantage of hardware-specific features (especially on Intel hardware). Therefore, DPDK still remains able to achieve higher performance than XDP, though at the disadvantage of being highly invasive [11]–[13] (see also Section 2.6). There have been academic proposals to utilize DPDK to accelerate QUIC packet processing, e.g. *picoquic-dpdk*, which claims to have improved throughput by a factor of 3 compared to the original *picoquic* implementation [14].

There are also further kernel bypass mechanisms like PF_RING ZC, a proprietary module from the PF_RING project of ntop. PF_RING ZC is a zero copy packet processing framework that aims to provide a simple and hardware independent API and thus differs slightly in its abstraction level from the more low-level XDP and DPDK [15]. Since release 7.6.0 it has incorporated XDP and currently is built on top of AF_XDP by default [16].

2. XDP

The eXpress Data Path (XDP) is a subproject of the IO Visor Project hosted by the Linux Foundation, initially introduced in the Linux kernel in 2016 [18]. XDP is a high performance data path in the Linux kernel based on the eBPF runtime (extended Berkeley Packet Filter). It acts as an early hook in the Linux network receive (RX) path that can be used to bypass the kernel network stack [6]. XDP was initially designed to be a purely in-kernel mechanism for bypassing the network stack. A primary use case was DoS mitigation, i.e. the ability to drop packets from malicious traffic as early as possible. However, the development of the AF_XDP socket type in 2018 allowed XDP programs to redirect network packets efficiently to user space. This enabled the development of high performance networking applications in user space that bypass the kernel network stack completely [11].

 doi: 10.2313/NET-2024-04-1_03

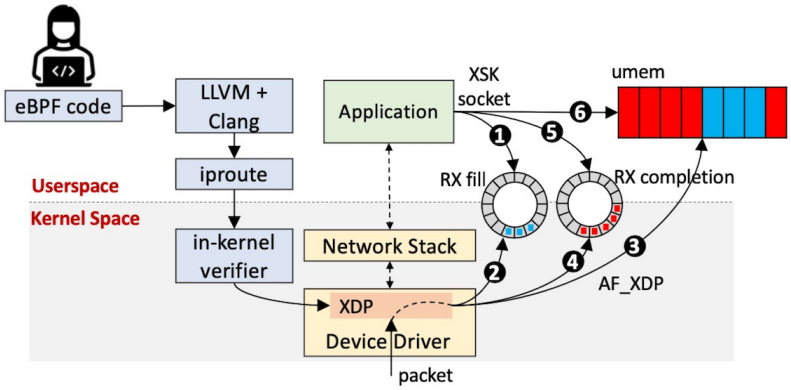

Figure 1: Receiving raw layer 2 frames via an AF_XDP socket (figure taken from [17])

2.1. eBPF

XDP builds on top of the extended Berkeley Packet Filter (eBPF), an extension to the largely deprecated classical Berkeley Packet Filter (cBPF) [5] in the Linux kernel. eBPF was initially released in 2014 (cBPF in 1992) and, like XDP, is part of the IO Visor Project. eBPF is an in-kernel virtual machine that allows to load and unload user-defined programs into designated code paths of the kernel network stack at runtime. The LLVM compiler backend supports compiling to eBPF bytecode. C (*libbpf* [19]) and Rust (*aya* [20]) libraries allow the programming of eBPF applications in higher level languages. eBPF programs have access to kernel networking infrastructure (e.g. routing tables) via BPF helper functions and are verified by the kernel on load to ensure program termination and to enforce in-kernel safety requirements.

The initial use case of BPF was to capture and filter network packets (e.g. by using tools like tcpdump). This is usually done via a tc (traffic control) eBPF program (type BPF_PROG_TYPE_SOCKET_FILTER) that hooks into either the ingress or egress point of the networking data path. Such eBPF programs are triggered before classical netfilter hooks, but after initial processing of the network stack (queuing, metadata extraction, GRO, etc.) as the hooks already reside inside the tc (traffic control) layer of the generic kernel network stack [21].

For high performance packet processing it is however more desirable to have an even earlier hook on the receive data path that circumvents the whole network stack. For such use cases the XDP program type was developed.

2.2. eBPF XDP Programs

An XDP program is a specific type of eBPF program (type BPF_PROG_TYPE_XDP). Once loaded on a network interface it is triggered directly after the networking driver has processed an incoming layer 2 frame from the NIC. It therefore bypasses the kernel network stack completely, runs before socket buffer allocation or GRO processing and operates on the raw layer 2 frame in the receive ring of the network driver (raw `xdp_buff` instead of metadata rich `sk_buff`). XDP programs can only be hooked to this ingress point of the networking data path and not to an egress point like with tc eBPF programs. The XDP program can arbitrarily process (and even modify) the received frame and eventually exits with a return value corresponding to a desired action to be triggered:

- **XDP_ABORTED**: drops the received frame and throws an eBPF tracepoint exception
- **XDP_DROP:** drops the received frame silently
- **XDP_PASS**: passes the received frame on to the in-kernel network stack
- **XDP_TX**: transmit the received frame back via the NIC it arrived on
- **XDP_REDIRECT**: redirect the received frame to another NIC or a user space AF_XDP socket

2.3. AF_XDP

AF_XDP is a specific socket type of the Linux Socket API that can be used to pass raw network packets efficiently between kernel space and user space via the fast XDP data path. The AF_XDP socket provides both receive and transmit semantics (though not via the classical syscalls). An AF_XDP socket (in user space) can receive raw layer 2 frames that a separate in-kernel XDP eBPF program has specifically redirected to it (via the XDP_REDIRECT action). This process is illustrated in Figure 1. An AF_XDP socket in user space can additionally transmit raw layer 2 frames out to the kernel that directly passes them to the networking driver. The socket type therefore bypasses the generic in-kernel network stack completely. The user-space application is thus fully responsible for parsing/processing the raw incoming layer 2 frames (received on the socket) and is also fully responsible for assembling the raw outgoing layer 2 frames (transmitted via the socket).

AF_XDP sockets are more complex in usage and configuration than regular socket types (e.g of type AF_INET). However, they enable high performance user-defined networking applications with similar performance to kernel bypass mechanisms that rely on hardware-specific user space drivers like DPDK. While AF_XDP provides a generic Linux interface that works regardless of hardware support, it only realize its full performance potential with driver support, e.g to facilitate zero copy semantics (see the XDP_ZEROCOPY flag) [21].

An AF_XDP socket is associated with the following data structures that users must configure and interact with:

- **UMEM**: A user-defined buffer of virtual contiguous memory consisting of equally sized frames. Network packets are passed between kernel space and user space via this buffer. The UMEM has two ring buffers associated with it:
  - **COMPLETION ring**: A consumer ring buffer in which the kernel stores pointers to UMEM frames when transmission of the corresponding frames has been completed.
  - **FILL ring**: A producer ring buffer in which the user space application can store pointers to UMEM frames that can be filled by the kernel when receiving on the socket.
- **TX ring**: A producer ring buffer in which the user space application can store pointers to UMEM frames that shall be transmitted by the kernel.
- **RX ring**: A consumer ring buffer in which the kernel stores pointers to UMEM frames that were received on the socket.

As with all sockets, a user application can wait for incoming packets or transmission completion by using the standard polling mechanisms of the Linux kernel (`poll()`, `select()`, `epoll()`). AF_XDP sockets can therefore be polled without pinning a whole CPU core to perform wasteful busy polling as was common in DPDK (however DPDK also supports event driven polling since version 17.05 [22]). Busy polling can still be performed with AF_XDP sockets via the `SO_BUSY_POLL` mechanism of the Linux socket API. This can be desirable, as busy polling ensures that AF_XDP applications can run using a single CPU core. Otherwise, two CPU cores will typically be used: one for the actual application and one for the RX/TX NAPI interrupt processing. This degrades performance as it causes costly coherency traffic between the cores [11].

2.4. XDP-for-Windows

XDP is a Linux technology, however Microsoft has also introduced an open source XDP implementation for the Windows operating system in 2022. This "XDP-for-Windows" implementation is heavily inspired but not a direct fork of the Linux XDP project [23].

As of September 2023, the implementation does not support eBPF programs, though the developers are claiming to aim for integration of XDP-for-Windows with the eBPF-for-Windows project. Instead of eBPF, XDP-for-Windows currently provides a custom built-in in-kernel XDP program. This program can not be coded directly by the user, but can be configured with matching rules and actions. This built-in XDP program can also be configured to redirect incoming network packets into user-space AF_XDP sockets (similar to the XDP_REDIRECT action in Linux XDP). The AF_XDP socket API is fully supported, but is not fully compatible with the Linux AF_XDP socket API. The underlying data structures used are analogous to Linux XDP, with a UMEM buffer and a TX, RX, COMPLETION, FILL ring buffer.

MsQuic, the QUIC implementation of Microsoft, already has experimental support for utilizing XDP-for-Windows in its Windows port [24].

2.5. Caveats of XDP

XDP is a mechanism to bypass the kernel network stack. Therefore, XDP can only benefit applications that do not require the functionality provided by the network stack (e.g. packet filtering for DoS mitigation) or can implement the functionality in a faster application-specific way (e.g. minimal UDP payload extraction in QUIC) [21]. Furthermore, an AF_XDP socket is not simply a high performance socket, but rather a way to pass raw layer 2 frames efficiently between the kernel and user space applications without having to traverse the kernel network stack. Unless applications can implement a more efficient way than the kernel network stack for processing raw layer 2 frames to the desired abstraction that classical sockets provide (e.g. a TCP byte stream), AF_XDP sockets can not replace classical socket usage in existing networking applications. When shortcutting the network stack via XDP, user applications also need to think about possible safety and security implications of their custom packet processing, e.g. if firewalls are bypassed, as the existing kernel networking mechanisms do not affect traffic redirected to AF_XDP sockets. For example, the QUIC implementations described in section 3 that utilize XDP bypass any classical firewalling on the system. Like all required functionality that would otherwise be provided by the kernel network stack, firewalling would have to be manually re-implemented when using XDP.

2.6. Comparison to DPDK

The main difference to other high speed packet processing mechanisms like DPDK is that XDP is highly integrated with the kernel. This means that XDP can provide an interface that is integrated into the standard operating system interfaces like eBPF or the socket API. XDP programs can also reuse existing networking capabilities of the kernel (e.g. routing tables). A major advantage compared to DPDK is that XDP facilitates device sharing, i.e. networking devices used by XDP applications remain visible/usable to non XDP applications [21]. XDP also does not require the use of huge pages like DPDK does [25]. A major drawback of XDP, however, is that it does not reach the performance of DPDK (e.g. 115 Mpps reached with DPDK against 100 Mpps reached with XDP in a packet drop benchmark on five cores [6]). CPU usage of DPDK was worse than XDP in the past due to DPDK requiring busy polling, but DPDK has since added support for event driven polling. Overall, XDP seems to be better suited in use cases that require minimal invasiveness and are more hardware agnostic (e.g. classical desktop applications), whereas DPDK seems to be better suited for specialized use cases that can tolerate high invasive measurements to achieve maximal performance (e.g. the classical DPDK use case of accelerating telecom NFV).

3. QUIC Implementations

The QUIC protocol is designed to replace the widely used TCP/TLS stack for web traffic. The proposed HTTP/3 standard is explicitly implemented on top of QUIC [26]. The protocol was standardized in 2021 with RFC 9000 [7]. QUIC implementations are still under

active development. But as QUIC is increasingly being adopted on the Internet, they are becoming ever more mature and are starting to invest more into optimizing performance. Major QUIC implementations are currently starting to utilize kernel bypass mechanisms to speed up their implementations, in particular XDP.

3.1. Concept of XDP Usage in QUIC

Conceptually, XDP is utilized in QUIC implementations to speed up UDP payload extraction from sockets. The QUIC protocol is designed to be implemented in user space instead of inside the kernel (where transport layer protocols typically are implemented) to allow for quicker and more flexible development cycles. However, the QUIC standard mandates a minimal UDP layer underneath QUIC, mainly to enable the traversal of legacy middleboxes [7].

A lot of the functionality of the generic kernel network stack is not necessary to process incoming QUIC traffic. Due to a combination of restrictions in the QUIC standard and its underlying network stack as well as the behavior of modern NICs, UDP payloads can be easily extracted from incoming layer 2 frames.

- **Layer 2:**
  The layer 2 protocol can be assumed to be Ethernet, since most modern NICs (or their corresponding drivers) will always emit incoming frames in ethernet format (even if it is received via WLAN it will wrap the received 802.11 payload into a "fake" ethernet frame). MAC addresses also do not need to be checked as networking devices are not set to promiscuous mode by default. Payloads are extracted by simply removing the Ethernet header.
- **Layer 3:**
  The layer 3 protocol used almost exclusively on the Internet is either IPv4 or IPv6. The QUIC standard forbids IP fragmentation by design [7]. This simplifies payload extraction, since one raw ethernet frame always holds exactly one UDP datagram. The IP header only needs to be processed minimally: IP addresses and the ECN (explicit congestion notification) must be stored for processing in higher layers of QUIC (e.g. connection migration and congestion control). The IPv4 checksum is not checked to improve performance. This can be assumed to be safe, since stronger integrity checks are employed in both lower layers (e.g. via ethernet CRCs) and higher layers (e.g. via TLS MAC/AEAD) of the employed QUIC network stack. IPv6 does not employ a checksum field for similar reasons [27]. Other fields of the IP header are also ignored by the examined implementations. Payloads are extracted by simply removing the IP header.
- **Layer 4:**
  Although QUIC itself implements layer 4 functionality, it builds on top of UDP by design. UDP is a minimalistic protocol that requires minimal processing. The ports are noted for processing in the QUIC connection migration mechanism. The UDP checksum is not checked to improve performance. This can be assumed to be safe for the same reasons as with IPv4 checksums. Payloads are extracted by simply removing the UDP header.

These steps can be done via XDP to process the incoming QUIC traffic on the RX path manually up to the UDP layer, bypassing the kernel network stack. For the TX path the process is inverted, however, all fields must be filled (even those ignored on the RX path) and all checksums must be computed.

3.2. Specific QUIC Implementations

To the best of the author's knowledge, currently two major QUIC implementations exist that utilize XDP:

- ***MsQuic* from Microsoft [9]:**
  The XDP implementation of *MsQuic* is currently (September 2023) under active development. It is released under "preview support" for the Windows porting of *MsQuic* and uses XDP-for-Windows (Linux is not supported yet) [24]. *MsQuic* is implemented in C++ and can therefore directly use the native XDP-for-Windows libraries. The implementation configures the built-in XDP program of XDP-for-Windows to redirect incoming traffic to AF_XDP sockets. The manual packet processing of UDP frames as described above is performed completely in user space. Currently the application supports only busy polling on AF_XDP sockets. Microsoft claims that by utilizing XDP it can improve both latency and throughput by ca. 100% on serialized HTTP requests [28].
- ***s2n-quic* from Amazon AWS [8]:**
  The XDP implementation of *s2n-quic* is currently (September 2023) also under active development and only available for Linux. *s2n-quic* is implemented in Rust and uses the *aya* eBPF/XDP library. It must wrap Linux C syscalls for use in Rust. The implementation uses a custom in-kernel eBPF XDP program to redirect traffic destined to the correct UDP port into AF_XDP sockets. The manual packet processing of UDP frames as described above is then performed in user space. *s2n-quic* supports both busy polling and event-based polling (via the *tokio* runtime) on AF_XDP sockets. XDP is not yet fully integrated into the API, but *s2n-quic* does provide a full server-client example ("s2n-quic-qns") that can be configured to use XDP. AWS does not provide detailed performance analysis but released profiling data indicates a significant reduction of kernel overhead when using XDP [29].

In addition, there has been an attempt by LiteSpeed Technologies to add XDP support to their *lsquic* QUIC implementation [30]. The design of the implementation is similar to *s2n-quic*: it utilizes Linux XDP with a separate eBPF XDP program that efficiently extracts UDP payloads and redirects to AF_XDP sockets. LiteSpeed claims that this gave a 43% improvement in throughput compared to standard UDP sockets [31]. However, this was only implemented as a proof-of-concept and was never incorporated into the official *lsquic* implementation.

4. Conclusion and future work

Kernel bypass mechanisms for networking applications are important tools to facilitate high-performance packet processing on general purpose processors/operating systems. XDP is an upcoming technology of this kind. It allows bypassing the kernel network stack and optionally passing raw layer 2 frames efficiently between user space applications and the kernel. Although it is still slower than its main competitor DPDK, it has the advantage of being highly integrated with the kernel (e.g. it does not require user space applications to take full control of the NIC).

Major implementations of the new QUIC transport protocol have started to utilize the XDP technology to speed up their implementations by reducing the kernel overhead of the standard UDP socket API. Due to the specifics of the QUIC protocol and its underlying network stack, the extraction of QUIC payloads from its underlying UDP layer can be done in a minimalistic manner that is more efficient than the generic kernel network stack. XDP can leverage this fact and has shown promising results for improving performance in major QUIC implementations.

References

[1] I. S. David Schinazi, Fan Yang, "Chrome is deploying HTTP/3 and IETF QUIC," 2020, accessed: 2023-09-30. [Online]. Available: https://blog.chromium.org/2020/10/chrome-is-deploying-http3-and-ietf-quic.html

[2] D. Damjanovic, "QUIC and HTTP/3 Support now in Firefox Nightly and Beta," 2021, accessed: 2023-09-30. [Online]. Available: https://hacks.mozilla.org/2021/04/quic-and-http-3-support-now-in-firefox-nightly-and-beta

[3] Apple Inc., "Safari 14 Release Notes," 2020, accessed: 2023-09-30. [Online]. Available: https://developer.apple.com/documentation/safari-release-notes/safari-14-release-notes

[4] E. Sy, C. Burkert, H. Federrath, and M. Fischer, "A QUIC Look at Web Tracking." *Proc. Priv. Enhancing Technol.*, vol. 2019, no. 3, pp. 255–266, 2019.

[5] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture." in *USENIX winter*, vol. 46, 1993, pp. 259–270.

[6] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. 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*, 2018, pp. 54–66.

[7] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[8] Amazon.com, Inc, "s2n-quic Github," accessed: 2023-09-30. [Online]. Available: https://github.com/aws/s2n-quic

[9] Microsoft Corporation, "MsQuic Github," accessed: 2023-09-30. [Online]. Available: https://github.com/microsoft/msquic

[10] A. LLC, "DPDK White Paper: Myth-Busting DPDK in 2020," The Linux Foundation, Tech. Rep., 2020.

[11] M. Karlsson and B. Töpel, "The path to DPDK speeds for AF_XDP," in *Linux Plumbers Conference*, 2018.

[12] E. Freitas, A. T. de Oliveira Filho, P. R. do Carmo, D. F. Sadok, and J. Kelner, "Takeaways from an experimental evaluation of eXpress Data Path (XDP) and Data Plane Development Kit (DPDK) under a Cloud Computing environment," *Research, Society and Development*, vol. 11, no. 12, pp. e26 111 234 200–e26 111 234 200, 2022.

[13] J. D. Brouer and T. Høiland-Jørgensen, "XDP: challenges and future work," in *Proc. Linux Plumbers Conference*, 2018.

[14] N. Tyunyayev, M. Piraux, O. Bonaventure, and T. Barbette, "A high-speed QUIC implementation," in *Proceedings of the 3rd International CoNEXT Student Workshop*, 2022, pp. 20–22.

[15] A. Cardigliano, "Positioning PF_RING ZC vs DPDK," 2017, accessed: 2023-09-30. [Online]. Available: https://www.ntop.org/pf_ring/positioning-pf_ring-zc-vs-dpdk/

[16] ntop, "PF_RING 7.6.0 release," 2020, accessed: 2023-09-30. [Online]. Available: https://github.com/ntop/PF_RING/releases/tag/7.6.0

[17] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff, "Revisiting the open vswitch dataplane ten years later," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 245–257.

[18] IO Visor Project, "XDP: eXpress Data Path," accessed: 2023-09-30. [Online]. Available: https://www.iovisor.org/technology/xdp

[19] Linux Project, "libbpf Github," accessed: 2023-09-30. [Online]. Available: https://github.com/libbpf/libbpf

[20] Aya Project, "aya Github," accessed: 2023-09-30. [Online]. Available: https://github.com/aya-rs/aya

[21] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[22] "DPDK Release 17.05," 2017, accessed: 2023-09-30. [Online]. Available: https://doc.dpdk.org/guides/rel_notes/release_17_05.html

[23] XDP-for-Windows, "Frequently Asked Questions," accessed: 2023-09-30. [Online]. Available: https://github.com/microsoft/xdp-for-windows/blob/main/docs/faq.md

[24] Microsoft Corporation, "MsQuic v2.2.0," 2018, accessed: 2023-09-30. [Online]. Available: https://github.com/microsoft/msquic/releases/tag/v2.2.0

[25] N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "Accelerating virtual network functions with fast-slow path architecture using express data path," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1474–1486, 2020.

[26] M. Bishop, "HTTP/3," RFC 9114, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9114

[27] D. S. E. Deering and B. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 8200, Jul. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8200

[28] Y. Huang, "Balance Performance in MsQuic and XDP," 2022, accessed: 2023-09-30. [Online]. Available: https://techcommunity.microsoft.com/t5/networking-blog/balance-performance-in-msquic-and-xdp/ba-p/3627665

[29] C. Bytheway, "PR: feat(s2n-quic-qns): add XDP server #1765," 2023, accessed: 2023-09-30. [Online]. Available: https://github.com/aws/s2n-quic/pull/1765

[30] LiteSpeed Technologies Inc, "lsquic github," accessed: 2023-09-30. [Online]. Available: https://github.com/litespeedtech/lsquic

[31] R. Perper, "Performance Comparison of QUIC with UDP and XDP," 2022, accessed: 2023-09-30. [Online]. Available: https://blog.litespeedtech.com/2020/06/01/performance-comparison-quic-udp-xdp