DirectQUIC: Transparent QUIC Reverse Proxy

1st Kokthay Poeng
Faculty of Computer Science
University of Namur
Namur, Belgium
kokthay.poeng@unamur.be

2nd Laurent Schumacher
Faculty of Computer Science
University of Namur
Namur, Belgium
laurent.schumacher@unamur.be

3rd Dararith Khun
Faculty of Computer Science
University of Namur
Namur, Belgium
dararith.khun@unamur.be

***Abstract*—Existing QUIC reverse proxies typically operate at Layer 7 by terminating QUIC connections, which breaks end-to-end (E2E) transport semantics, adds overhead, and centralizes TLS keys and certificates at the proxy. We present *DirectQUIC*, a transparent QUIC reverse proxy that preserves E2E encryption and QUIC transport semantics without terminating the connection or storing backend TLS private keys at the proxy. DirectQUIC uses limited control-plane processing during connection establishment and Connection ID-based forwarding in steady state, and supports both shared-IP deployment and a full redirection mode. We implement DirectQUIC with eBPF/XDP and complement it with certificate issuance controls. Experiments with multiple QUIC stacks and major browsers show transparent deployability, while comparisons against existing Layer 7 reverse proxies show lower latency, higher throughput, and lower CPU and memory overhead.**

***Index Terms*—QUIC, Reverse Proxy, Transparent Proxy, eBPF/XDP, Connection ID routing, E2E Encryption**

I. Introduction

QUIC is now a major Internet transport protocol and the foundation of HTTP/3 [1]. Since its introduction by Google in 2017 [2], QUIC has seen rapid adoption across major Internet platforms: HTTP/3 was used by 31.1% of websites globally in 2024 [3], Meta reported that more than 75% of its Internet traffic relied on QUIC in 2022 [4], QUIC carried about 40% of browser traffic in 2023 [5], and accounted for 28% of HTTP requests on Cloudflare's network by May 2023, up from 23% the previous year [6]. Overall, QUIC is estimated to represent nearly 40% of total Internet traffic by early 2025 [7]. Its integrated cryptographic handshake, stream multiplexing, and connection migration improve performance and robustness compared to TCP-based transports.

Reverse proxies remain essential in modern deployments for service multiplexing, load balancing, and hosting multiple services behind a single public IP address [8]. However, existing QUIC reverse proxies typically operate at Layer 7 by terminating QUIC at the proxy and establishing a new connection toward the backend [9], [10], [11]. This approach breaks end-to-end (E2E) QUIC transport semantics, adds handshake and processing overhead, and requires the proxy to manage TLS certificates and private keys for hosted services, increasing security exposure by concentrating cryptographic trust at the proxy.

ISBN 978-3-903176-82-9

Treating QUIC as plain UDP at Layer 4 is also insufficient for Reverse Proxy. For example, Layer 4 reverse proxies such as the NGINX `stream`[12] module can forward UDP traffic, but they do not account for QUIC-specific connection semantics. Robust QUIC reverse proxying therefore requires awareness of QUIC Connection Identifiers (CIDs) and migration behavior, rather than relying on IP addresses and ports.

To address these limitations, we present *DirectQUIC*, a transparent QUIC reverse proxy that preserves E2E encryption and QUIC transport semantics without terminating connections or storing backend TLS private keys at the proxy. DirectQUIC extracts the Server Name Indication (SNI) during connection establishment and uses visible CIDs for steady-state forwarding. We further couple DirectQUIC with certificate issuance controls based on off-proxy DNS-01 validation and CAA restrictions to better preserve E2E encryption.

DirectQUIC makes four main contributions. First, we design a QUIC-aware transparent reverse proxy that preserves E2E encryption and QUIC transport semantics without terminating QUIC connections or storing backend TLS keys and certificates at the proxy. Second, we propose two deployment modes for transparent QUIC reverse proxying: (i) multi-service hosting behind a single shared public IP address, similar to traditional reverse proxies, and (ii) a full redirection mode that combines transparent QUIC forwarding, Direct Server Return (DSR) and path redirection, allowing traffic to bypass the proxy after connection establishment and minimizing steady-state overhead. Fourth, we implement a proof of concept using eBPF/XDP and evaluate DirectQUIC against HAProxy, Caddy, and Envoy, showing lower latency, higher throughput, and lower CPU and memory overhead.

II. Related Work

Existing QUIC-capable reverse proxies such as HAProxy, Caddy, and Envoy support HTTP/3 and QUIC by operating as Layer 7 proxies that terminate QUIC at the frontend and establish new connections toward backend servers [9], [10], [11]. This enables application-layer functionalities, but breaks E2E QUIC transport semantics and requires centralized management of TLS keys and certificates at the proxy. NGINX also supports HTTP/3, but QUIC reverse proxy is not yet available as a stable upstream feature in its mainline release [13].

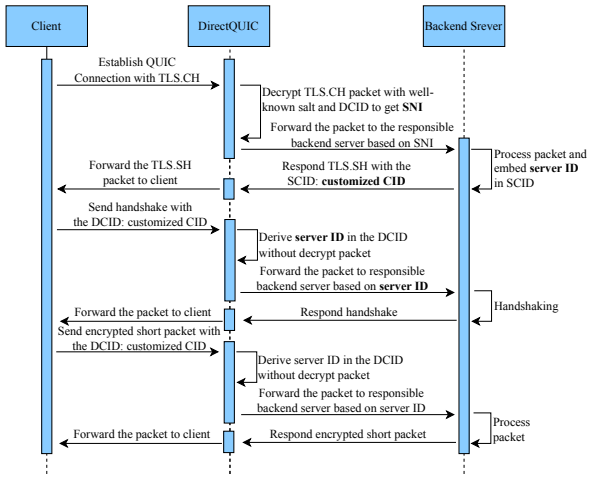

Fig. 1. Connection establishment in DirectQUIC under the shared-IP mode.

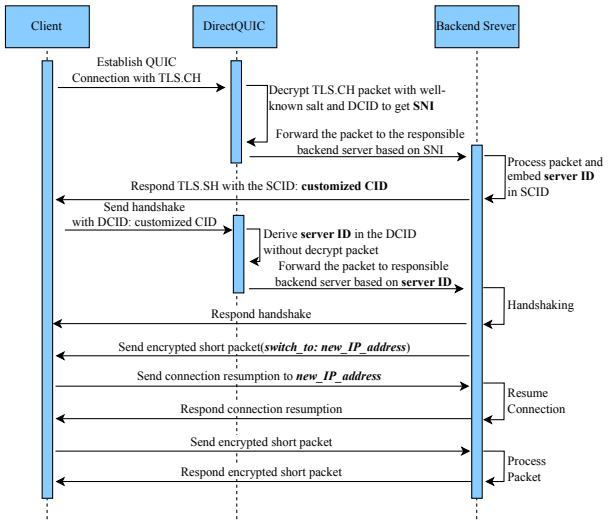

Fig. 2. Connection establishment in DirectQUIC under the full redirection mode.

In contrast, generic Layer 4 UDP forwarding lacks awareness of QUIC connection semantics and therefore cannot robustly support service multiplexing or connection migration based only on IP addresses and ports. QUIC Load Balancers (QUIC-LB) address this limitation by encoding routing information into server-generated CIDs so that load balancers can steer traffic without decrypting QUIC packets [14]. DirectQUIC is inspired by this principle, but extends it from load balancing to transparent reverse proxy and supports both shared-IP multi-service hosting and a redirection mode. In addition, DirectQUIC couples transparent QUIC forwarding with certificate issuance controls to better preserve E2E encryption if the client-facing proxy is compromised.

Prior systems have also explored reducing proxy forwarding overhead through DSR or QUIC-specific stream distribution [15], [16]. Unlike these approaches, DirectQUIC neither terminates QUIC nor shares transport and cryptographic state across proxy and backends. Instead, it preserves the standard E2E QUIC model while forwarding traffic statelessly using routing information encoded in visible CIDs.

III. DirectQUIC

This section describes the design of *DirectQUIC*, detailing system architecture, the connection establishment process, steady-state CID-aware forwarding, eBPF implementation and certificate issuance.

A. Connection Establishment

Figures 1 and 2 illustrate the connection establishment process in DirectQUIC under its two supported deployment modes. In both cases, the process begins when a client initiates a QUIC connection by sending an Initial packet containing the TLS ClientHello to the proxy's IP address. During connection establishment, DirectQUIC performs limited control-plane processing on QUIC Initial packets in order to extract routing metadata, while steady-state forwarding remains fully transparent and CID-based. The Initial packet is therefore exported to userspace for control-plane processing.

The control-plane analyzer extracts the SNI and selects the appropriate backend server, after which the Initial packet is forwarded toward that backend without terminating the QUIC connection. The backend processes the ClientHello and generates a ServerHello response.

During the handshake, the backend embeds routing information in the server-generated CID. The customized CID encodes a server ID that is opaque to the client but meaningful to DirectQUIC. The ServerHello packet carrying this customized CID is forwarded back to the client. Once the handshake completes, the client sends subsequent QUIC packets using the customized CID.

In the shared-IP mode (Figure 1), the proxy remains on the data path and forwards traffic between the client and backend servers located in a private subnet. In contrast, in the full redirection mode (Figure 2), the backend explicitly instructs the client to establish a new QUIC connection directly to the backend's public IP address. Upon receiving this redirection signal, the client initiates a new QUIC connection using connection resumption or 0-RTT, after which all subsequent traffic bypasses the proxy entirely. This mode eliminates steady state proxy overhead at the cost of requiring a public IP address per backend server.

B. Architecture Overview

Figure 3 presents the architecture of DirectQUIC, which adopts a split data plane and control-plane design to enable transparent QUIC routing with minimal overhead. Data plane packet processing is implemented in kernelspace using eBPF/XDP, while control-plane logic and limited protocol analysis are handled in userspace.

The data plane consists of a QUIC-aware XDP program attached to both upstream and downstream network interfaces. On downstream traffic (client to proxy), the XDP data plane performs fast-path packet forwarding while identifying QUIC Initial packets. For these packets, it emits events to userspace

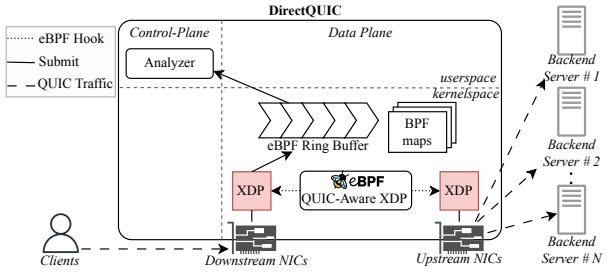

Fig. 3. DirectQUIC architecture overview.

via an eBPF ring buffer without interrupting normal data plane forwarding. The userspace Control-Plane Analyzer consumes these events, extracts the SNI, selects an appropriate backend server, and routes the packet accordingly.

C. Steady-State Forwarding

After connection establishment, DirectQUIC forwards all subsequent QUIC packets exclusively in the data plane without control-plane involvement. Forwarding decisions are performed based solely on the CID carried in the QUIC packet header. The QUIC-aware XDP data plane derives the server ID encoded in the CID and forwards packets accordingly, independent of the client's IP address or port. This enables stateless, per-packet routing at line rate.

In the shared-IP mode, the proxy remains on the data path and continues to forward packets between the client and backend servers using CID-based routing. In the full redirection mode, steady state forwarding occurs directly between the client and backend server after the client establishes a new QUIC connection to the backend's public IP address. In this case, the proxy is completely removed from the data path, eliminating steady state forwarding overhead.

DirectQUIC does not track or learn new CIDs explicitly. All server-generated CIDs, including those issued during connection migration, embed the same server ID, allowing correct forwarding by inspecting the visible CID header without observing encrypted control frames or maintaining per-client CID state across backend servers. By restricting steady-state packet processing to simple CID extraction and XDP-based forwarding, DirectQUIC introduces only minimal overhead.

D. eBPF Deployment

DirectQUIC extends LinkQUIC [17] and is implemented using a combination of eBPF/XDP programs in the kernel and userspace control logic. The data plane bypasses the traditional kernel networking stack by attaching XDP programs directly to network interfaces. Packet forwarding is performed using `bpf_redirect()`, allowing packets to be forwarded between interfaces without traversing the IP, UDP, or socket layers, thereby minimizing per-packet processing overhead. For packets forwarded using IP-based routing in the shared-IP mode, the XDP data plane leverages `bpf_fib_lookup()` to resolve the appropriate next-hop parameters, including destination MAC address and egress interface. This allows DirectQUIC to perform correct forwarding decisions directly at XDP while remaining consistent with the host's routing configuration.

Routing state is maintained in a single BPF map indexed by a server ID embedded in the QUIC CID. In the shared-IP mode, map entries associate server ID with backend IP and port, enabling proxy-based forwarding to backend servers in a private subnet. In the full redirection mode, entries associate server IDs with backend MAC addresses, allowing backend servers to respond directly to clients using the proxy's virtual IP (VIP) while bypassing the proxy in both states.

DirectQUIC requires minimal modifications to QUIC endpoints. Backend servers are extended to embed a server ID into server-generated CIDs during the handshake. No changes to QUIC packet formats or cryptographic protocols are required. To support full redirection, backend servers send an explicit redirection signal to the client after the handshake, instructing it to establish a new QUIC connection to the backend's public IP address. Clients implement a corresponding mechanism to initiate a new connection using QUIC connection resumption or 0-RTT, after which all traffic flows directly between the client and backend.

In the shared-IP mode, the XDP data plane forwards client-to-server traffic by rewriting the destination IP from the proxy's IP address to the selected backend server's IP, while preserving the original client source IP and port. As a result, backend servers observe packets as originating from the real client rather than from a proxy-originated connection. For return traffic, packets from the backend are routed back through the proxy, which rewrites the source IP from the backend server's IP to the proxy's IP before forwarding them to the client. Thus, the proxy remains on the data path in both directions in shared-IP mode, and the client consistently communicates with the proxy's advertised service address.

In the full redirection mode, the proxy advertises the VIP as the externally reachable service address, while backend servers configure the same VIP only on their loopback interfaces and do not advertise it at Layer 2. As a result, ingress traffic is directed to the proxy, which then steers packets to the selected backend by rewriting only the destination MAC address. Because the backend locally owns the VIP on loopback, it can generate responses using the VIP as the source IP without requiring IP or port translation. This design avoids Layer 2 ambiguity, preserves a single public service address, and enables DSR behavior while removing the proxy from steady-state packet processing.

E. Certificate Issuance Control

Preserving E2E encryption in DirectQUIC requires not only avoiding QUIC termination at the proxy, but also constraining certificate issuance so that a malicious client-facing proxy cannot obtain replacement certificates for backend domains. Otherwise, a compromised proxy could obtain a valid certificate, impersonate the backend server, and escalate into a TLS-terminating person-in-the-middle. In our deployment model, backend TLS private keys remain exclusively at the backend

server, while certificate validation is restricted to off-proxy DNS-01 validation under the control of the backend owner. In addition, CAA records restrict which certificate authorities (CAs) and validation methods are authorized for issuance. Under these assumptions, compromise of DirectQUIC alone does not by itself expose backend TLS private keys or enable passive decryption of QUIC application traffic.

This deployment model relies on three main trust components: the backend server, which stores the TLS private key and terminates QUIC/TLS; the DNS zone authority, which controls DNS-based certificate validation; and all CAs expected to enforce the configured issuance restrictions. DirectQUIC itself is not supposed to store backend TLS private keys or terminate the E2E encrypted connection. Extended Validation certificates may further increase the operational barrier against malicious certificate issuance through additional approval procedures.

IV. Evaluation

We evaluate DirectQUIC using a PoC implementation and compare it against representative QUIC reverse proxies: HAProxy, Caddy, and Envoy. Our evaluation considers two aspects: deployability and performance. First, we examine whether DirectQUIC supports practical QUIC deployments without terminating connections or managing TLS keys. Second, we compare its latency, throughput, CPU cycle consumption, and memory usage against Layer 7 QUIC reverse proxies.

NGINX is excluded because QUIC reverse proxy support is not available as a stable upstream feature in its official mainline codebase, hindering reproducible evaluation. In contrast, HAProxy, Envoy, and Caddy provide QUIC reverse proxy support in their official codebases.

Our prototype runs on a VM hosted on an ASUS Mini PC under Proxmox, using a Python-based eBPF BCC control plane and a C-based eBPF data plane. The proxy VM is allocated 4 CPU cores and 32 GB RAM. backend servers run on a separate VM with 8 CPU cores and 32 GB RAM, and clients run on two VMs with 12 CPU cores and 48 GB RAM. All systems are connected through a 1 Gbps local network.

A. Effectiveness and Deployability

We evaluate deployability using nine QUIC clients: six implementations (quic-go, quiche, msquic, aioquic, s2n-quic, and neqo) and three major browsers (Chrome, Firefox, and Safari). All successfully establish QUIC connections and exchange data through DirectQUIC. This is achieved without terminating QUIC at the proxy and without storing backend TLS private keys or certificates.

For browsers, we implement a TCP fallback path based on SNI extracted during the TCP handshake. The backend then advertises HTTP/3 using `Alt-Svc`, allowing subsequent connections to use QUIC. This enables interoperability with unmodified browsers and existing deployment practices.

DirectQUIC aligns with QUIC-LB design principles by avoiding connection termination at the proxy and using visible server-generated CIDs for routing, while extending this model to transparent reverse proxying. Backend selection can be performed using standard policies such as round-robin or hashing, with the selected server ID encoded into the CID. By avoiding proxy-side QUIC termination and centralized storage of backend TLS credentials, DirectQUIC preserves QUIC transport semantics and supports E2E encryption. However, preserving this property against a malicious client-facing proxy additionally requires certificate issuance restrictions so that the proxy cannot obtain replacement certificates and escalate into a TLS-terminating impersonation point. As a result, compromise of DirectQUIC does not by itself expose backend TLS private keys or enable passive decryption of QUIC application traffic. Overall, these results show that DirectQUIC supports practical QUIC deployments while reducing key-management exposure compared to termination-based QUIC reverse proxies.

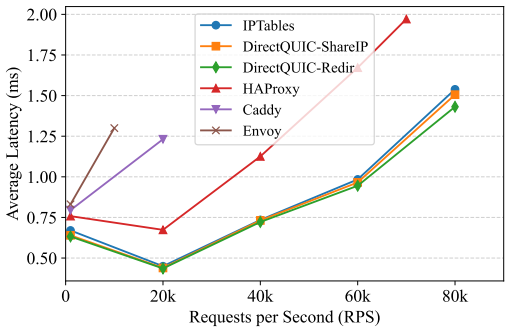

Fig. 4. Average request–response latency under increasing load within the stable operating region of each proxy.

B. Performance

We evaluate performance in terms of latency, throughput, CPU utilization, and memory consumption. For latency and throughput, an iptables-based NAT setup is used as a baseline. All proxies are evaluated under the same network conditions.

Latency. We measure request–response latency while increasing load from 1K to 80K requests/s. Figure 4 shows that both DirectQUIC modes closely track the `iptables` baseline and maintain substantially lower latency than HAProxy, Caddy, and Envoy. Termination-based proxies become proxy-limited earlier, whereas DirectQUIC remains below proxy saturation and is primarily limited by backend or network capacity at high load.

Throughput. We measure throughput over a fixed 10-second interval using 100 concurrent QUIC connections. As shown in Figure 5, both DirectQUIC modes achieve throughput close to the `iptables` baseline, with full redirection approaching direct client-to-server communication. HAProxy, Caddy, and Envoy achieve lower throughput due to QUIC termination and userspace forwarding overhead.

CPU Utilization. Figure 6 reports CPU cycle consumption under increasing request rates. DirectQUIC consistently incurs the lowest CPU overhead because steady-state forwarding is performed in kernelspace using XDP and CID-based routing.

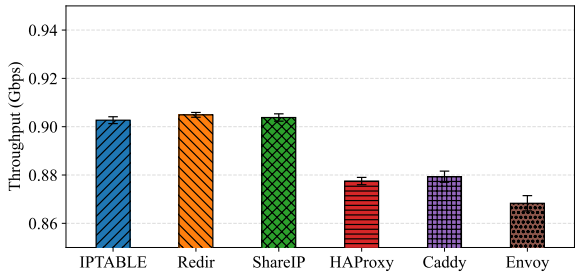

Fig. 5. Average throughput.

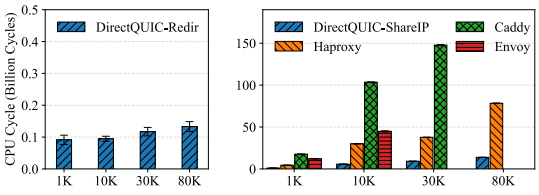

Fig. 6. CPU cycle consumption under increasing request rates.

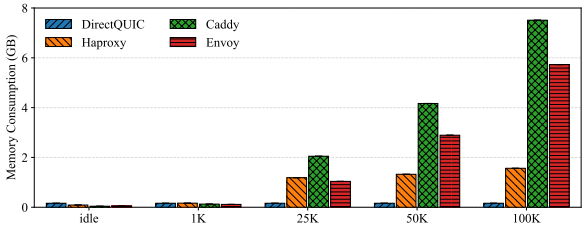

Fig. 7. Memory consumption as a function of concurrent connections.

The full redirection mode achieves the lowest CPU utilization overall, as the proxy participates only during connection establishment and is removed entirely from the steady state data path. Even at 80K RPS, DirectQUIC remains well below CPU saturation, confirming that the proxy itself is not the limiting factor. In contrast, HAProxy, Caddy, and Envoy consume substantially more CPU due to QUIC termination, cryptographic processing, and userspace handling, and saturate at lower request rates.

Memory Consumption. We measure memory usage while gradually increasing the number of concurrent QUIC connections. Figure 7 shows that DirectQUIC maintains a nearly constant memory footprint because its transparent forwarding design avoids per-connection state at the proxy. In the kernel data plane, DirectQUIC uses only a fixed-size BPF map that associates server IDs with forwarding information, which does not scale with the number of connections. In userspace, memory is used only transiently for QUIC Initial packet processing, and no per-connection QUIC, TLS, or stream state is retained. As a result, both DirectQUIC modes exhibit identical and flat memory consumption. In contrast, HAProxy, Caddy, and Envoy show steadily increasing memory usage as connection counts grow, reflecting their need to maintain per-connection QUIC, TLS, and stream state.

Overall, DirectQUIC achieves lower latency, higher throughput, lower CPU overhead, and near-constant memory usage, while preserving E2E QUIC semantics without proxy-side QUIC termination.

V. Discussion

Encrypted Client Hello (ECH) [18] is an extension to TLS 1.3 that encrypts the SNI and other sensitive handshake parameters, preventing passive and active on-path observers from learning the intended destination service. ECH is particularly relevant to QUIC-based deployments, as it further reduces protocol visibility during connection establishment and challenges middleboxes that rely on plaintext SNI for routing decisions. However, the ECH specification explicitly defines a *Client-Facing Server* deployment model, in which an initial server endpoint is trusted to decrypt the ClientHello and recover the inner SNI before forwarding the connection toward the backend service. DirectQUIC is compatible with this model: it can operate either as the Client-Facing Server itself, extracting routing information prior to backend selection, or be deployed behind an existing Client-Facing Server that exposes the decrypted SNI, thereby preserving its design goals while remaining compatible with emerging ECH-enabled deployments.

Shared-IP vs. Full Redirection. DirectQUIC offers two deployment modes, each reflecting a different balance between practicality and performance. In the shared-IP mode, multiple backend services are exposed behind one public IP, which keeps deployment simple and does not require changes to QUIC clients. The drawback is that the proxy remains on the steady-state forwarding path, so some forwarding overhead is unavoidable. The full redirection mode takes the opposite approach: after the initial connection setup, the client is steered to communicate directly with the selected backend. This removes the proxy from the steady-state path and improves efficiency, but it requires every backend server to have a publicly reachable IP and needs extra coordination between the server and the client during setup.

QUIC defines the *Preferred Server Address (PSA)* transport parameter, which allows a server to advertise an alternative IP address to the client during the handshake. Upon receiving this parameter in the ServerHello, a compliant client can migrate the connection to the preferred IP address without establishing a new connection. This mechanism is directly relevant to the full redirection mode of DirectQUIC, as it provides a standardized way for a server to steer the client toward a backend IP address immediately after connection establishment. In principle, PSA could eliminate the need for an explicit redirection signal and client-side reconnection, enabling faster and more efficient path migration. However, support for PSA is currently limited or absent in most widely deployed QUIC implementations, and enabling it would require intrusive modifications or forking of QUIC stacks. To preserve compatibility with existing QUIC stacks, DirectQUIC implements redirection at the application

level using explicit signaling and connection resumption. Even with mature PSA support, full redirection would continue to require routable backend IP addresses, making shared-IP deployment the more practical choice for many environments. These trade-offs highlight that shared-IP and full redirection modes are complementary, enabling DirectQUIC to adapt to diverse deployment scenarios ranging from small-scale edge systems to high-performance cloud infrastructures.

Transport-Layer Proxying vs HTTP/3 Reverse Proxies. Existing QUIC-capable reverse proxies such as HAProxy, Envoy, and Caddy primarily operate as HTTP/3 reverse proxies rather than generic QUIC transport-layer proxies. HTTP/3 is an application-layer protocol that runs on top of QUIC and inherits its transport properties while introducing HTTP-specific semantics. By terminating QUIC connections and processing HTTP/3 streams, these systems provide rich application-layer functionalities, including request routing, header manipulation, authentication, rate limiting, and protocol-aware load balancing. Such capabilities are beyond the scope of DirectQUIC, which intentionally operates below the application layer and does not interpret HTTP semantics.

This transport-layer design allows DirectQUIC to support a broader class of QUIC-based services. It can forward not only HTTP/3 traffic, but also other QUIC applications, provided that routing information such as SNI is available during connection establishment. The cost of this transparency is that backend servers must participate by embedding a server ID into QUIC CIDs, similar to the routing approach used in QUIC-LB. By comparison, HAProxy, Envoy, and Caddy avoid such backend-side QUIC modifications because they rely on full connection termination and centralized TLS handling at the proxy. The contrast is therefore not simply implementation style, but a deeper architectural trade-off between application-layer functionality on one side and transport-layer transparency and generality on the other.

VI. Conclusion

This paper introduced *DirectQUIC*, a transparent QUIC reverse proxy that preserves E2E encryption and QUIC transport semantics without terminating connections or storing backend TLS private keys and certificates at the proxy. DirectQUIC combines limited control-plane processing during connection establishment with stateless CID-based forwarding in the steady state, and we implemented it as a proof of concept using eBPF/XDP.

Our evaluation shows that DirectQUIC interoperates with diverse QUIC stacks and browsers and outperforms termination-based QUIC reverse proxies across latency, throughput, CPU overhead, and memory scalability. These results indicate that transparent, CID-aware QUIC proxying is a practical alternative to Layer 7 QUIC termination for deployments that prioritize E2E transport semantics, efficiency, and reduced key-management exposure.

We further argue that preserving E2E encryption in this trust model requires coupling transparent proxying with certificate issuance restrictions. Under such a deployment model, the risk that a malicious client-facing proxy obtains replacement certificates and silently escalates into a TLS-terminating impersonation point can be significantly reduced. Future work includes broader load-balancing policies, deeper integration with mechanisms such as ECH and PSA, and hardening the design for production deployment.

Acknowledgment

This research is supported by Belgium Walloon Region CyberExcellence Program (Grant #2110186).

References

[1] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport," RFC 9000, 2021. [Online]. Available: https://datatracker.ietf.org/doc/rfc9000/

[2] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196.

[3] I. Singh, "QUIC: The secure communication protocol shaping the future of the internet," Zscaler Blog, Oct 2024, [Online]. Available: https://www.zscaler.com/blogs/product-insights/quic-secure-communication-protocol-shaping-future-of-internet.

[4] T. Ingale, "Watch: Meta's engineers discuss QUIC and TCP innovations for our network," Engineering at Meta, Jul 2022, [Online]. Available: https://engineering.fb.com/2022/07/06/networking-traffic/watch-metas-engineers-discuss-quic-and-tcp-innovations-for-our-network/.

[5] T. Roeder, "QUIC: How DPI with encrypted traffic intelligence supports the web's latest protocol," ipoque.com Blog, Dec 2023, [Online]. Available: https://www.ipoque.com/blog/quic-enhanced-traffic-visibility-dpi.

[6] D. Belson and L. Pardue, "Examining HTTP/3 usage one year on," Cloudflare Blog, Jun 2023, [Online]. Available: https://blog.cloudflare.com/http3-usage-one-year-on/.

[7] A. Walding, "An update on QUIC Adoption and traffic levels," cellstream.com, Feb 2025, [Online]. Available: https://www.cellstream.com/2025/02/14/an-update-on-quic-adoption-and-traffic-levels/.

[8] HAProxy Technologies, "What Is a Reverse Proxy?" https://www.haproxy.com/glossary/what-is-a-reverse-proxy, 2026.

[9] Envoy Project, "HTTP/3 Architecture Overview," https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/http/http3, 2026.

[10] Caddy, "Caddyfile Protocols," https://caddyserver.com/docs/caddyfile/options#protocols, 2026.

[11] HAProxy Technologies, "HAProxy Protocol Support (HTTP/3, HTTP/2, HTTP)," https://www.haproxy.com/documentation/haproxy-configuration-tutorials/protocol-support/http/, 2026.

[12] NGINX, Inc., "NGINX ngx_stream_proxy_module Documentation," https://nginx.org/en/docs/stream/ngx_stream_proxy_module.html, 2026.

[13] arut, "nginx HTTP/3 Upstream Branch (GitHub)," https://github.com/arut/nginx/tree/http3-upstream, 2026.

[14] M. Duke, N. Banks, and C. Huitema, "QUIC-LB: Generating Routable QUIC Connection IDs," https://datatracker.ietf.org/doc/draft-ietf-quic-load-balancers/, 2025, internet-Draft, work in progress.

[15] Z. Wei, Z. Wang, Q. Li, Y. Yang, C. Luo, F. Wang, Y. Jiang, S. Yang, and Z. Yuan, "QDSR: Accelerating layer-7 load balancing by direct server return with QUIC," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 715–730. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/wei

[16] E. M. Makhroute, Q. De Coninck, and T. Barbette, "Multi-end quic: A transport protocol to enable one-to-many communications," ser. CoNEXT-SW '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 3–4. [Online]. Available: https://doi.org/10.1145/3769700.3771696

[17] K. Poeng, L. Schumacher, and S. Touch, "Quic traffic identification in kernel space," in *2025 International Symposium on Networks, Computers and Communications (ISNCC)*, 2025, pp. 1–8.

[18] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood, "Tls encrypted client hello," RFC 9849, 2026. [Online]. Available: https://datatracker.ietf.org/doc/rfc9849/