MASQUE NADA: QUIC-based proxying with `tokio-quiche`

Alexander Fink, Daniel Petri*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: alexander.fink@tum.de, petriroc@net.in.tum.de

***Abstract*—Multiplexed Application Substrate over QUIC Encryption (MASQUE) is a recent approach to proxying that leverages the QUIC transport protocol and HTTP/3 to tunnel traffic. In this paper, we design and implement a proof-of-concept MASQUE proxy and client, NADA, using Cloudflare's tokio_quiche library in Rust. The prototype establishes UDP tunnels via CONNECT-UDP and forwards datagrams between clients and target hosts through the proxy. We further incorporate the Capsule Protocol to enable reliable data transport between the client and proxy and discuss the implementation constraints regarding payload-size/MTU limits introduced by QUIC and HTTP Datagram encapsulation. We demonstrate that one can easily set up a MASQUE proxy with a moderate amount of code in Rust.**

***Index Terms*—MASQUE, QUIC, HTTP/3, CONNECT-UDP, HTTP Datagrams, Capsule Protocol, tokio-quiche, Rust**

1. Introduction

Traditional HTTP Proxies use the HTTP `CONNECT` method to open TCP tunnels to target hosts [1], [2]. While this is suitable for TCP-based protocols, until recently there was no convenient general-purpose way to proxy UDP-based protocols (e.g., QUIC) over an HTTP proxy, aside from protocol-specific workarounds.

MASQUE (Multiplexed Application Substrate over QUIC Encryption) is a recently standardized set of protocols that tackle this shortcoming by enabling the tunneling of non-TCP-based protocols over HTTP [3]. As a relatively new standard, support remains limited and implementations are still emerging. In this work, we present a proof-of-concept MASQUE proxy and client implementation for UDP in Rust using Cloudflare's `tokio-quiche` library. While MASQUE's proxying capabilities have been extended to support the tunneling of arbitrary IP and Ethernet packets, our implementation focuses on UDP tunneling [4], [5].

2. Tunneling UDP with MASQUE

Scenario: Figure 1 shows the MASQUE proxy scenario used throughout this work. A MASQUE client establishes an HTTP/3 connection over QUIC to a MASQUE proxy (outer connection). The client then requests a UDP tunnel to a target host and port using `CONNECT-UDP`. After the proxy accepts the request, it forwards the tunnel payloads as plain UDP to the target server and vice versa.

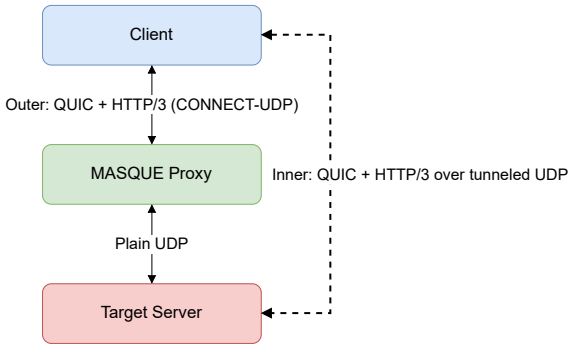

Figure 1: MASQUE proxy scenario used in this work.

`CONNECT-UDP` can carry arbitrary UDP payloads; it does not require QUIC [3]. In our experiments, we tunnel QUIC/HTTP/3 inside the `CONNECT-UDP` tunnel (inner connection) to simplify client-side testing.

At a high level, tunneling UDP with MASQUE works as follows [3]:

1) The client first establishes a secure HTTP/3 connection to the MASQUE proxy.
2) The client sends an extended `CONNECT` request with `:protocol` set to `connect-udp` to instruct the proxy to open a UDP tunnel to a specified target host and port.
3) If the proxy accepts the request, it opens a UDP socket to the target server and sends back a success response, effectively establishing the tunnel.

Once the tunnel is set up, the client and proxy exchange data: the client sends packets to the proxy and the proxy forwards the UDP payloads to the target server. Likewise, UDP responses received by the proxy are relayed back to the client over the outer HTTP/3 connection.

HTTP Datagrams and the Capsule Protocol: Once the UDP tunnel is established, MASQUE defines two ways to carry the UDP packet data between client and proxy: HTTP Datagrams and Capsules [3], [6]. When using HTTP Datagrams, MASQUE leverages QUIC `DATAGRAM` frames to transmit the HTTP Datagrams. The HTTP Datagram header itself is rather small, as it only contains the Quarter Stream ID to identify the HTTP stream that the datagram belongs to and is followed by the UDP payload [6]. These HTTP Datagrams are unreliably delivered and unordered, similar to UDP itself. Thus, reliable transmission, when required, is left to the internal connection. Hence, if any datagram is lost in transit be-

 doi: 10.2313/NET-2026-05-1_10

tween client and proxy, the outer QUIC connection will not take care of retransmitting it.

The proxy, upon receiving a QUIC DATAGRAM frame containing an HTTP Datagram from the client, extracts the UDP payload and forwards it to the corresponding target over UDP. Similarly, incoming UDP packets from the target are sent back to the client in HTTP Datagrams.

For certain applications, e.g., when there is strong packet loss between client and proxy, the unreliable behavior is undesired. In those cases, the Capsule Protocol mode is better suited for exchanging the UDP payloads with the server [6].

Unlike pure HTTP Datagrams, capsules are sent over the reliable HTTP stream; hence, they are delivered in order and reliably between client and proxy. The structure of the capsule header is rather simple: The header contains information about the content-type, a length field, and is followed by the payload. For tunneling UDP payloads, there is a special DATAGRAM capsule content-type defined that expects HTTP Datagrams as payload [6]. In our MASQUE proxy scenario (Figure 1), when switching to capsule mode, all exchanges between client and proxy happen through capsules. The server extracts the HTTP Datagram from inside and then proceeds with it as in datagram mode. The responses from the target server are packed into DATAGRAM capsules before sending them back to the client [6].

Both endpoints discover at connection initialization whether the other party supports HTTP/3 datagrams via the QUIC SETTINGS frame at the beginning of the connection, where the SETTINGS_H3_DATAGRAM setting is set to 1 in case it is supported [6], [7]. This is only a hint that HTTP Datagrams are supported, but it is not mandatory for client and proxy to use it. In fact, the Capsule Protocol is always active as soon as the CONNECT-UDP request gets acknowledged by the server [3]. However, it is up to the implementation to decide not to send any packets through capsules; thus, from an external observer's perspective, it looks like the Capsule Protocol is not used.

One trade-off with capsule-based tunneling is reintroducing ordering and head-of-line blocking for that flow [8]. Since all capsules for a given tunnel flow in one HTTP/3 stream, if a packet is lost and retransmitted, subsequent data for the same tunnel must wait. In contrast, in datagram-only mode, each datagram is independent. Loss of one datagram does not delay delivery of later ones; they can arrive out of order. Thus, capsule mode is appropriate when reliability is required, but it may incur latency penalties if loss occurs.

A caveat of MASQUE UDP proxying is that proxies must respect the path MTU of the outer QUIC connection. QUIC DATAGRAM frames cannot be fragmented at the QUIC layer, and QUIC endpoints are required to avoid IP-layer fragmentation [8], [9]. Not only can the MTU on the path between client and proxy be smaller than on the path between proxy and target, but the effective maximum size of inner datagrams is additionally limited by the QUIC configuration of client and proxy. When QUIC DATAGRAM frames are used, the max_datagram_frame_size transport parameter also limits the maximum DATAGRAM frame size, and this limit can be further restricted by max_udp_payload_size [9]. On top of these limits, one has to take into account that the maximum size of inner datagrams on the path between client and proxy is further reduced by the overhead of packing the UDP payload into HTTP Datagrams and QUIC DATAGRAM frames [6], [10].

In our implementation (Section 3), we build a MASQUE proxy and client on top of tokio-quiche that provides a QUIC/HTTP/3 stack in Rust. This reduces boilerplate code and lets us focus on MASQUE-specific logic rather than the details of I/O integration.

3. MASQUE NADA Implementation

As mentioned in Section 2, our MASQUE implementation consists of the MASQUE client and the MASQUE proxy. Both components use the tokio-quiche library, an open-source library that builds on top of quiche and provides a high-level, asynchronous interface that integrates Tokio [11].

3.1. QUIC / HTTP/3 Stack

Tokio is an asynchronous runtime for Rust that provides an event-driven platform to manage network connections and tasks efficiently [12]. It allows writing networking code using async/await, with the runtime handling the low-level details of scheduling and I/O polling.

The underlying quiche library exposes a relatively low-level API: It requires explicitly managing I/O integration like receiving and sending datagrams and forwarding the payload to quiche [11]. Implementing a MASQUE proxy directly on top of quiche would therefore involve substantial boilerplate and additional overhead, thus we choose tokio-quiche. By delegating low-level UDP socket management and QUIC handshake/events to tokio-quiche, our implementation can focus on HTTP/3 request handling and MASQUE-specific logic.

3.2. Tunnel initialization and HTTP Datagrams

As illustrated in Figure 2, in our implementation a user executes the MASQUE client as a CLI application, which establishes an HTTP/3 connection to the MASQUE proxy. The client then issues a CONNECT-UDP request to create a UDP tunnel to a target server. In more detail:

1. **Connection Setup**: The client opens a QUIC connection to the MASQUE proxy [3]. Both client and proxy use the tokio-quiche library, which handles the QUIC handshake and sets up an HTTP/3 session over QUIC. The MASQUE client container opens a QUIC connection to the MASQUE proxy container (Figure 2). Both sides use tokio-quiche to perform the QUIC handshake and establish an HTTP/3 session. This greatly simplifies initialization, because tokio-quiche handles UDP socket I/O and HTTP/3 session setup [11]. Once the connection is established, HTTP/3 streams can be used for sending requests and responses.

2. **Extended CONNECT Request**: Once the HTTP/3 connection is established, the client initiates a MASQUE tunneling request by sending an extended CONNECT request with :protocol = connect-udp. The destination of the UDP tunnel (target IP and port) is encoded in the request :path. In addition, the capsule-protocol = ?1 header

 doi: 10.2313/NET-2026-05-1_10

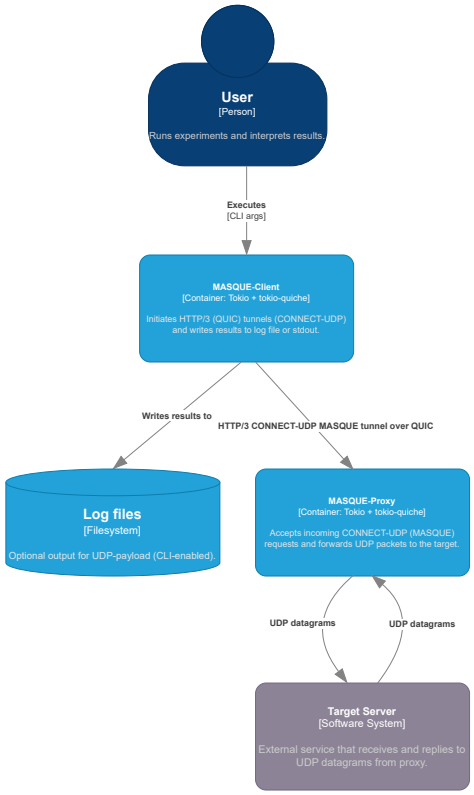

Figure 2: C4 Container Diagram of the MASQUE Rust implementation.

indicates that the client supports exchanging UDP payloads as `DATAGRAM` capsules on the `CONNECT` stream (we send `?1` whenever the header is present).

For example, to reach a UDP endpoint at *target-ip* on *port*, the client sends a request of the following form:

```
:method    = CONNECT
:protocol  = connect-udp
:authority = masque.technology
:scheme    = https
:path      = /.well-known/masque/udp/target-ip/port
capsule-protocol = ?1
```

As outlined in Section 2, after the proxy accepts the `CONNECT-UDP` request, the tunnel is established and the proxy forwards traffic in both directions, carrying client datagrams to the target UDP endpoint and returning UDP packets from the target through the HTTP/3 tunnel.

The MASQUE Client may optionally store the received packet payload to disk. In our implementation, this corresponds to writing output to the "Log files" container (a filesystem) shown in Figure 2.

QUIC UDP Payload Limits: We first implemented `CONNECT-UDP` with HTTP Datagrams and tested it with Cloudflare's `async_http3_server` [13]. Testing it with the example server failed, while the same client appeared to work when tunneling to public targets (e.g. google.com). This discrepancy suggested the implementation was not fundamentally broken. We therefore investigated the effective packet budget of the outer QUIC connection and confirmed the issue was caused by faulty QUIC configuration: the inner UDP payload plus HTTP Datagram and QUIC overhead exceeded `tokio-quiche`'s default `max_send_udp_payload_size = 1200 bytes` setting [13]. In the Cloudflare example the `max_send_udp_payload_size` is set to 1350 bytes [13]. This mismatch led to dropped packets on the proxy, as datagrams are not allowed to be fragmented and the received packets from the target server were bigger than what the proxy can send. To solve this situation, we can either increase `max_send_udp_payload_size` on the proxy or decrease the `max_recv_udp_payload_size` QUIC setting in the inner connection between client and target server, as by default it is 65527 bytes. The `max_recv_udp_payload_size` setting value is advertised by the client during connection initialization to the target server, signaling that the client will not process packets with larger payloads [8]. While servers are not strictly required to follow this advertised setting value, most implementations do. That gives us the capability to limit the sender's maximum UDP payload size from the client. After setting the relevant QUIC parameters accordingly, the client–proxy tunnel worked for larger payloads within the configured boundaries. Note that the `max_send_udp_payload_size` setting itself is never advertised by the proxy to the client or target server. We therefore recommend setting `max_send_udp_payload_size` and `max_recv_udp_payload_size` to the same value. The advantage here is that the client can take the payload size setting from the outer connection, decrease it by potential MASQUE overhead and use that value to configure the inner connection's `max_recv_udp_payload_size`.

In our implementation, we configure QUIC's `max_udp_payload_size` transport parameter to 1300 bytes, which is a cap on the size of inner datagrams so that, after adding HTTP Datagram encapsulation and QUIC packet headers, the resulting QUIC packet still fits onto the path. We follow Cloudflare's recommendation to cap `max_udp_payload_size` at 1300 bytes to reduce the risk of exceeding typical path MTUs once encapsulation overhead is included [10]. For typical paths with an MTU of 1500 bytes, they argue that subtracting IP/UDP headers and QUIC `DATAGRAM` frame + HTTP Datagram overhead leaves roughly 1200–1300 bytes for tunneled payloads and that using the fixed cap is a good practice in order to avoid datagrams that are too large for the proxy-client flow while still allowing QUIC clients to send the required initial 1200-byte QUIC packets [8].

3.3. Reliable Transport with the Capsule Protocol

After resolving the payload-size limitations observed with QUIC `DATAGRAM` and HTTP Datagram encapsulation, we added Capsule Protocol support to our implementation to enable reliable delivery between client and proxy. As stated above, it is left up to the implementation whether to use HTTP Datagrams or the Capsule Protocol in case both are supported by the implementation, thus we allow choosing between datagram and capsule mode via a command-line interface flag for the client.

When the Capsule Protocol becomes active, the communication changes as follows: Instead of sending UDP payloads in QUIC DATAGRAM frames, the client sends them as HTTP/3 Data frames on the CONNECT request stream, but wrapped in a DATAGRAM capsule, a structure defined by the Capsule Protocol [3], [6]. The proxy reads DATAGRAM capsules from the CONNECT stream, extracts the enclosed UDP payload, and forwards it to the target over UDP. UDP responses from the target are encapsulated into DATAGRAM capsules and written back to the client on the same stream.

Our initial implementation followed the HTTP Datagram approach: We implemented a simple encoder/decoder to wrap each UDP packet with the capsule header (as defined in RFC 9297) [6]. Each outgoing UDP packet was encoded in one DATAGRAM capsule and incoming data was decoded as one capsule. This assumption turned out to be incorrect because capsules are carried over HTTP/3 streams [7]. Unlike QUIC DATAGRAM frames, which preserve message boundaries, HTTP/3 streams are stream-oriented. Thus, reads from them do not preserve message boundaries and can return split or combined data. For example, capsule headers (type/length varints) can span multiple reads, and multiple capsules can be returned back-to-back in a single read. Consequently, our stateless decoder failed for certain payloads.

To correctly handle the capsule mode, we implemented a capsule parser as a state machine. Incoming stream chunks are appended to a buffer, and parsing proceeds in three stages:

1) Decode the capsule type once enough bytes are available
2) Decode the capsule length once enough bytes are available
3) Wait until the buffer contains length bytes to extract the complete capsule payload

Upon full reconstruction of DATAGRAM capsules, we extract their enclosed UDP payload. The server then forwards the payload via UDP to the target server like in HTTP Datagram mode. The client processes received DATAGRAM capsules analogously. The client then extracts the enclosed UDP payload from completed DATAGRAM capsules and processes it exactly as in datagram mode. Any remaining bytes stay in the buffer for the next parsing iteration of the capsule parser. This design makes the implementation independent of how HTTP/3 streams chunk data into QUIC stream frames.

4. Conclusion and future work

We implemented a proof-of-concept MASQUE client and proxy in Rust using Cloudflare's tokio-quiche, demonstrating that a functional MASQUE setup can be realized with a reasonably small amount of MASQUE-specific code.

Our prototype supports MASQUE's CONNECT-UDP mechanism to establish a UDP tunnel to a target host and forward traffic between client and target via the proxy. As a result of that, our client is able to build inner HTTP/3 connections to the target through the proxy. We implemented both data transport options relevant to MASQUE UDP tunneling: unreliable delivery using HTTP Datagrams over QUIC DATAGRAM frames, and reliable delivery using the Capsule Protocol by sending DATAGRAM capsules over the CONNECT stream. In addition, we emphasized constraints that matter in interacting with arbitrary target servers, in particular MTU and payload-size limitations imposed by QUIC and HTTP Datagram encapsulation. MASQUE NADA is publicly available [14] for further experimentation and research purposes.

Future improvements for our proof-of-concept could include adding support for proxying arbitrary IP packets using the CONNECT-IP method, implementing proxy authentication and client authorization as well as testing interoperability with other MASQUE implementations.

References

[1] R. T. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9110

[2] J. Wignall. (2025, May) Beneath the masque — a dive into network relay technology on apple platforms. Blog post. [Online]. Available: https://jedda.me/beneath-the-masque-network-relay-on-apple-platforms/

[3] D. Schinazi, "Proxying UDP in HTTP," RFC 9298, Aug. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9298

[4] T. Pauly, D. Schinazi, A. Chernyakhovsky, M. Kühlewind, and M. Westerlund, "Proxying IP in HTTP," RFC 9484, Oct. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9484

[5] A. Sedeño, "Proxying Ethernet in HTTP," Internet Engineering Task Force, Internet-Draft draft-ietf-masque-connect-ethernet-08, Oct. 2025, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-masque-connect-ethernet/08/

[6] D. Schinazi and L. Pardue, "HTTP Datagrams and the Capsule Protocol," RFC 9297, Aug. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9297

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

[8] 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

[9] T. Pauly, E. Kinnear, and D. Schinazi, "An Unreliable Datagram Extension to QUIC," RFC 9221, Mar. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9221

[10] Cloudflare. (2022, Mar.) Unlocking QUIC's proxying potential with MASQUE. Cloudflare Blog. [Online]. Available: https://blog.cloudflare.com/unlocking-quic-proxying-potential/

[11] Mendes, L. Blöcher, E. Rittenhouse, and F. Darling. (2025, Nov.) Async QUIC and HTTP/3 made easy: tokio-quiche is now open-source. Cloudflare Blog. [Online]. Available: https://blog.cloudflare.com/async-quic-and-http-3-made-easy-tokio-quiche-is-now-open-source/

[12] Tokio Contributors, "Crate tokio 1.48.0 (rust crate documentation)," Docs.rs. [Online]. Available: https://docs.rs/tokio/1.48.0/tokio/

[13] Cloudflare. (2025, Aug.) Github: tokio-quiche. [Online]. Available: https://github.com/cloudflare/quiche/tree/0.24.5/tokio-quiche

[14] D. Petri. Nada, a prototype MASQUE implementation. GitHub repository. [Online]. Available: https://github.com/danielpetri1/nada