Evaluation of QUIC-based MASQUE Proxying

Mirja Kühlewind
mirja.kuehlewind@ericsson.com
Ericsson
Herzogenrath, Germany

Matias Carlander-Reuterfelt
Ericsson
Madrid, Spain
matias.carlander-reuterfelt@ericsson.com

Marcus Ihlar
Ericsson
Stockholm, Sweden
marcus.ihlar@ericsson.com

Magnus Westerlund
Ericsson
Stockholm, Sweden
magnus.westerlund@ericsson.com

ABSTRACT

Multiplexed Application Substrate over QUIC Encryption (MASQUE) is a new protocol mechanism that defines an extension to the HTTP CONNECT to support QUIC-based tunneling to a proxy and forwarding of UDP and IP traffic. MASQUE can be used in cases where tunneling is beneficial such as VPN-like scenarios or other proxy-based services. In this paper we investigate impacts on end-to-end QUIC performance when using a MASQUE-based tunnel setup. In particular, we investigate the impacts of reliable versus unreliable transmission of packets in the tunnel connection, as well as the impacts of nested congestion control between the inner connection and the out tunnel connection. We show that tunneling implies per-packet overhead but can also provide additional performance benefits, e.g. for lossy local links.

CCS CONCEPTS

• **Networks** → **Network performance analysis**.

ACM Reference Format:
Mirja Kühlewind, Matias Carlander-Reuterfelt, Marcus Ihlar, and Magnus Westerlund. 2021. Evaluation of QUIC-based MASQUE Proxying. In *Workshop on the Evolution, Performance and Interoperability of QUIC (EPIQ'21), December 7, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3488660.3493806

1 INTRODUCTION

QUIC [4] a is a new, encrypted transport protocol published by the IETF in May 2021. HTTP/3 [2] is being standardized as an application protocol on top of QUIC. The MASQUE working group in the IETF develops extensions to the HTTP CONNECT methods [3, 6] that enables the use of HTTP/3 as a tunneling protocol between a client and a proxy. As shown in Figure 1, an HTTP/3 client initiates a connection to a proxy and then uses MASQUE to request forwarding of either UDP or IP packets to a target server. These packets are sent encapsulated in the QUIC tunnel connection between the client and proxy. The proxy then decapsulates the packets and forwards them to the target server, and encapsulates packets from the server before forwarding to the client.

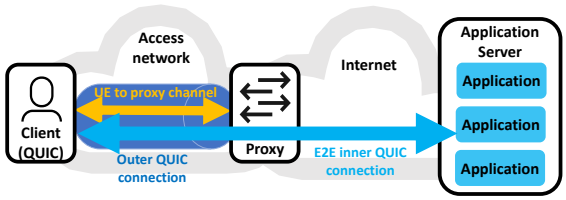

Figure 1: Proxy setup using MASQUE for QUIC-based tunneling (image originally published in [5]).

The described setup can be used for multiple use cases. MASQUE can be used for VPN-like services where additional encryption is desired for a certain segment of the network path or when direct end-to-end connectivity is not possible. Like a NAT, a MASQUE proxy can be instructed to hide the client's IP address from a target server. This use case is already realized and deployed by Apple's private relay service[1] using MASQUE.

With the deployment of these use cases, we expect MASQUE to become part of a new common communication pattern where the client establishes a tunnel to a proxy in the local access network to then request network services from that proxy. The most basic services offered are forwarding and address translation. Several other collaborative network services can be enabled by making use of the direct communication channel MASQUE provides.

In mobile networks explicit proxy cooperation provides an opportunity for improved policy enforcement and performance enhancements compared to current solutions primarily based on Deep Packet Inspection (DPI). With explicit proxy cooperation using MASQUE the client requests a service for a certain flow or group of flows and the proxy checks the client's eligibility, e.g., allow zero rating for video traffic only to a specific set of IP addresses. Another example is the implementation of performance enhancement functions where the proxy provides additional information about the local network conditions to the client as discussed in [5].

The contribution of this paper is twofold. First, we present and discuss our implementation of MASQUE as well as our network emulation setup. In the second part of the paper, we present our

[1] https://developer.apple.com/support/prepare-your-network-for-icloud-private-relay/

EPIQ'21, December 7, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-9135-1/21/12…$15.00
https://doi.org/10.1145/3488660.3493806

evaluation study and discuss various aspects of using QUIC and HTTP/3 to tunnel end-to-end QUIC traffic.

We test a set of basic network conditions such as different delay and loss scenarios or using different QUIC packet sizes. We quantify the overhead of adding an additional encapsulation and discuss the impact of nested congestion control. While performance, as expected, is impacted, we can also show that e.g. local recovery by using reliable streams in the QUIC tunnel can be beneficial in certain scenarios.

The evaluation is based on an implementation of MASQUE on top of aioquic[2] QUIC and HTTP/3 implementation. The evaluation is restricted to a single implementation since there are few QUIC and HTTP/3 stacks that currently support MASQUE. Furthermore, since QUIC is a relatively new protocol, most currently available implementations can be optimized further, e.g. with respect to congestion control [16]. Therefore, the results presented mainly discuss general findings about implications of the use of QUIC as a tunnel encapsulation and do not provide a final performance assessment.

The next section contains an overview of related work, Section 3 describes our MASQUE implementation and explains the network emulation environment and Section 4 shows the findings of the MASQUE evaluation study.

2 RELATED WORK

Proxies are widely deployed in the Internet, in particular in access networks. Sherry et al. [12] investigated enterprise networks and found that there are often as many middleboxes as routers. Often these proxies perform security functions such as access control but also performance enhancement (PE) functions. Zullo et al. found TCP splitting PEPs in 86 % of investigated LTE cellular networks. Xu et al. [15] detected transparent HTTP or TCP proxies on the networks of all four US carriers. Wang et al. [13] detected NATs in about 80% of the investigated networks.

NATs or middleboxes performing PE are transparent to the user. In addition, there is an equally large ecosystem of, usually HTTP-based, explicit proxies that are used to access web content from, e.g., restricted networks. While such proxies promise anonymity and censorship circumvention, Perino et al. [8] detected problems such as ad injections and TLS interception.

MASQUE is addressing both use cases: replacing transparent proxies that perform PE or NAT with explicit user-consent based proxying and web proxying without interception or compromising the end-to-end security.

3 IMPLEMENTATION

3.1 MASQUE

The IETF MASQUE working group is developing specifications that add UDP proxy support to HTTP. At the time of writing this document MASQUE consists of two specifications. The first specification describes the CONNECT-UDP HTTP method [10] and the second specification defines the concept of HTTP datagrams [11].

The CONNECT-UDP method is used to transform an HTTP request stream into a tunnel for UDP payload. The receiver of a CONNECT-UDP request acts as a proxy and converts tunneled UDP payload to UDP datagrams and vice versa. UDP payload is tunneled using HTTP datagrams.

The HTTP datagram specification describes how to use QUIC datagrams [7] with HTTP. QUIC datagram frames are sent unreliably, meaning that a QUIC sender will not attempt to retransmit the frame payload if it is considered lost. HTTP allows datagrams to be multiplexed by associating groups of datagrams with a request stream. The HTTP datagram specification also describes a new HTTP frame type known as CAPSULE. The main intention of CAPSULE frames is to register the use of datagrams, but they can also be used to encode datagrams. A datagram which is encoded as a CAPSULE frame will be transmitted reliably over a QUIC stream. From here on these two ways of transmission will be referred to as datagram and stream mode.

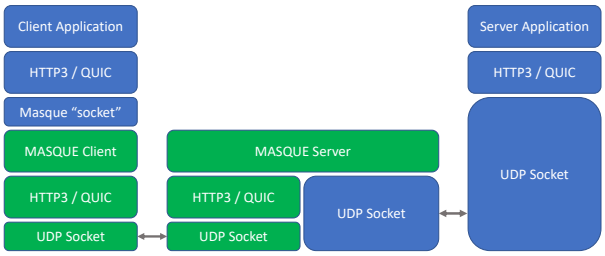

Figure 2: Application structure for the MASQUE setup.

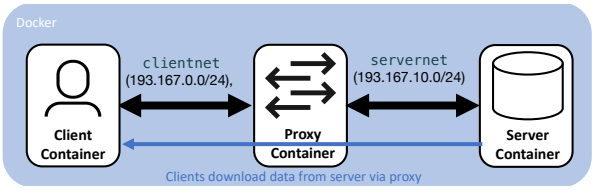

Figure 3: Docker-based emulation setup.

3.2 Python-based MASQUE Prototype

The evaluation in this paper is based on a MASQUE prototype developed on top of the aioquic stack. aioquic is an event driven QUIC and HTTP3 implementation written in Python; it supports pluggable io but ships with the Python asyncio library as default.

The prototype consists of MASQUE enabled client and server applications and standard HTTP applications as shown in Figure 2. The MASQUE applications implement the CONNECT UDP method. A "MASQUE socket" is implemented by extending the anyncio DatagramTranport class that wraps UDP socket calls.

The aioquic HTTP/3 implementation has been modified with support for HTTP datagrams and CAPSULE frames. The QUIC implementation has been modified to support configurable packet sizes and pluggable congestion control. The Cubic [14] congestion control algorithm has been implemented along with a "null algorithm" that effectively disables congestion control.

[2]https://github.com/aiortc/aioquic

3.3 Docker-based Emulation Setup

A Docker-based emulation setup is used to evaluate the various MASQUE aspects discussed in this paper. The emulation setup consists of three containers respectively hosting a MASQUE capable HTTP3 client, a MASQUE server and an HTTP3 server. The containers are connected by two docker networks as shown in Figure 3.

Network conditions and characteristics are emulated using the Linux traffic control (tc) tool. The link between the proxy and server has no bandwidth limitation and a fixed one-way delay of 25 ms in each direction. The properties of the link between the client and proxy are varied between different experiments. The baseline configuration of link between the client and proxy has a bandwidth limitation of 10 Mbit/s, a 5 ms one-way in both directions and a buffer with a depth of 200 ms. These parameters are selected to reflect an access network deployment of the proxy.

All test cases described in this document consist of a client requesting a specified amount of payload from a server using the HTTP/3 GET method. For test cases, where traffic is tunneled between the client and proxy, the client sends a CONNECT-UDP request to the proxy prior to establishing the connection with the server. For test cases where traffic is not tunneled the proxy container works as a router, forwarding packets between the client and the server based on iptables rules. Each test case is executed 25 times.

In our setup, the execution of tests and post processing is automated. A set of qlog [9] files is generated after each test. The qlog files are processed to extract metrics used in the evaluation, such metrics include retransmission rates and byte overheads.

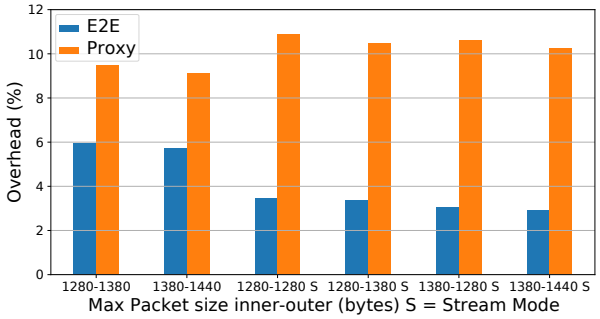

Figure 4: Overhead for different packet size configurations for the inner and outer QUIC connection.

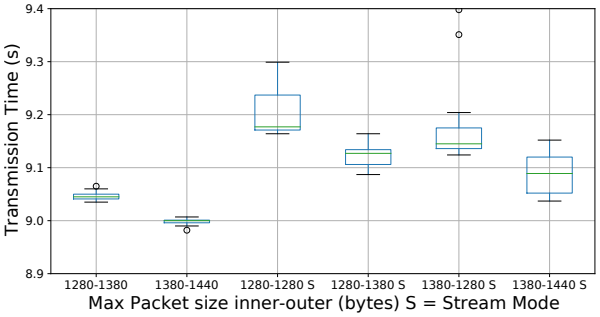

Figure 5: Transmission time for different packet size configs for the inner and outer QUIC connection.

4 EVALUATION

4.1 Discussion on bit overheads and MTU

We will start the evaluation with a discussion about overhead and implications of using HTTP/3 as a tunnel protocol. The addition of frame and packet headers introduced by QUIC and HTTP/3 changes the ratio of throughput and goodput. QUIC assumes a minimum supported IP packet size of 1280 bytes, but for most evaluation scenarios we assume that the MTU between the client and proxy can support larger packet sizes than that. Therefore, the maximum QUIC packet size of the tunnel connection is typically configured to 1380 bytes and the maximum QUIC packet size of the end-to-end connection to 1280 bytes.

Figure 4 shows the average overhead in relation to useful application data for a number of packet size configurations. The overhead includes QUIC and HTTP/3 headers and retransmitted payload; it is calculated in the server and proxy respectively. The X axis shows the different maximum QUIC packet size configurations, where the left value shows the packet size of the end-to-end connection and the right value shows the packet size of the tunnel connection. As expected, there is a slight reduction of overhead with larger packet sizes. In stream mode, parts of the observed overhead is moved from the server to the proxy. The overhead is caused retransmission of loss packet due to self-induced congestion at the bottleneck. In stream mode, the proxy retransmits all packets quickly without triggering any loss detection in the server.

Figure 5 shows a boxplot with whiskers out to 1.5 times the interquartile range (IQR) for the transmission times with the same packet size configurations as in Figure 4. The first two boxes show transfer times when using datagram mode as the tunnel encapsulation. As expected, larger end-to-end packet sizes increase performance and reduce transfer times. When using stream mode for the tunnel encapsulation it is not required to fit an entire end-to-end QUIC packet in a single QUIC tunnel packet. The cases where the end-to-end packet sizes are larger than what can fit in a single packet of the tunnel connection, i.e. 1280-1280 and 1380-1280, see a degradation in performance due to the splitting of end-to-end packets in order to fit in the tunnel packets. Comparing 1280-1280 with 1380-1280 we also observe a slight improvement due to the reduced overhead end-to-end from fewer QUIC and HTTP/3 headers.

Figure 6 shows a set of scenarios where 10 MB of data is requested from the server, divided equally over multiple connections. All end-to-end connections are multiplexed over a single tunnel connection and correspond to separate CONNECT-UDP requests. It should be noted that more interesting scenarios can be considered when multiple servers at different distances are used, however, the use of a single server greatly simplifies the network emulation setup.

Increased overhead is observed with an increased number of connections, this is partially explained by more connection establishment overheads, including CONNECT-UDP requests. A larger increase in overhead is observed in stream mode than in datagram

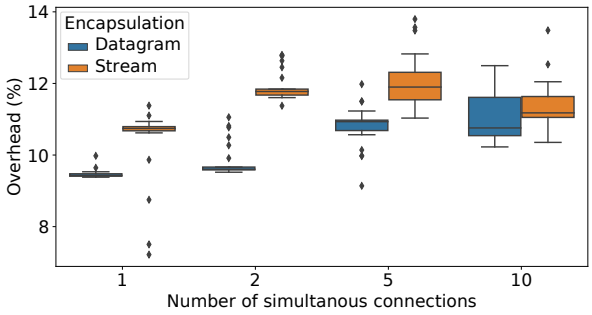

Figure 6: Overhead depending on number of simultaneous end-to-end QUIC connections for 10 MB payload equally split.

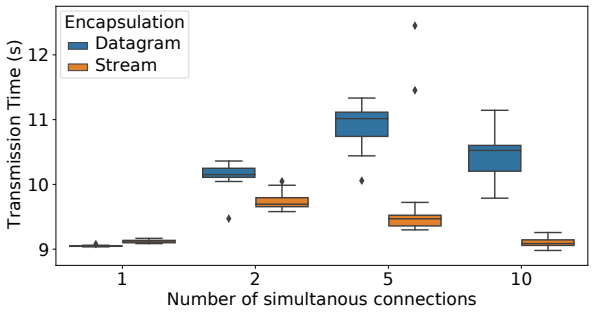

Figure 7: Completion time depending on number of simultaneous end-to-end QUIC connections for 10 MB payload equally split.

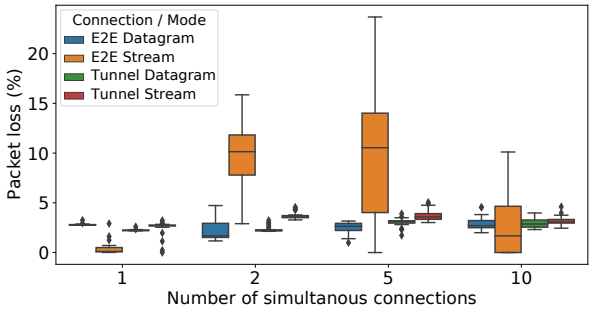

Figure 8: Packet loss rate depending on encapsulation mode and number of simultaneous end-to-end QUIC connections for 10 MB payload equally split.

mode. This is mainly explained by a significant increase in end-to-end retransmissions, in particular in the 2 and 5 connection scenarios as seen in Figure 8.

This increased packet loss in stream mode is mainly caused by the implemented stream scheduling in aioquic. Stream data buffers are always accessed in the same order. This means that data will be written on a single stream until it is either empty or blocked due to flow control. Since the different end-to-end connections are mapped to streams in the MASQUE proxy, a single connection will make forward progress while other connections are being buffered. This causes significant end-to-end delay variation which spuriously triggers timer-based loss detection in the server. The reduction of retransmissions observed with 10 parallel connections is due to less data being scheduled per stream, leading to smaller delay variations.

In contrast, the datagram mode has a single transmission queue across all end-to-end connections, which means that packets will be scheduled based on arrival order to the proxy. In this case we do not observe an increase in end-to-end loss compared to loss detected by the proxy. However, a slight increase of packet loss rates in the tunnel connection is observed when the number of simultaneous end-to-end connections increase. This is likely due to more data being sent to the proxy before it is congestion window limited in the server.

Figure 7 shows the completion times associated with the number of connections. In stream mode we observe a degradation of transfer times in the 2-connection scenario compared to the single connection scenario. Transfer times improve with increased number of parallel connections compared to the 2-connection scenario. This behavior is expected given the observation of the impact of stream scheduling end-to-end loss detection.

However, given there is no stream scheduling for datagrams, this does not explain the degradation of completion times in datagram mode. We observe that the number of end-to-end packets increase with 1.7, 5.3, and 8.1 % respectively for the 2-, 5- and 10-connection scenarios compared to the single connection scenario. The number of retransmissions increase by 7.2, 33.1 and 33.6 % respectively compared to the single connection scenario. However, the increased number of retransmissions only accounts for a small fraction (10-20%) of the total packet overhead. It seems like the completion time degradation is attributed to a combination of the increased congestion loss and increased packet overhead. The exact nature of the packet overhead needs to be further investigated.

4.2 Impact of RTTs and congestion control

In this section we further investigate the effects of nested congestion control (CC). We look at a set of scenarios with varying delays between the client and the proxy. In all the scenarios delay between the proxy and server are fixed with a one way delay (OWD) of 25 ms while the OWD between the client and proxy varies from 1 to 50 ms. Each scenario is evaluated with the Reno and Cubic CC algorithms enabled in the proxy for both stream and datagram modes. The scenarios are also run with CC completely disabled in the proxy in datagram mode. Finally, the results are contrasted with pure end-to-end traffic without any proxying. In all cases the server uses Reno CC algorithm.

Figure 9 shows the time to transmit a 10 MB message from the server to the client. Increased delay results in increased transfer times, and in general completion times are shorter in datagram mode than in stream mode. The results are stable for all scenarios except the one with a OWD of 50 ms, which is the only scenario where the client-to-proxy delay constitutes a larger fraction of the end-to-end delay than the proxy-to-server delay. In this case, the

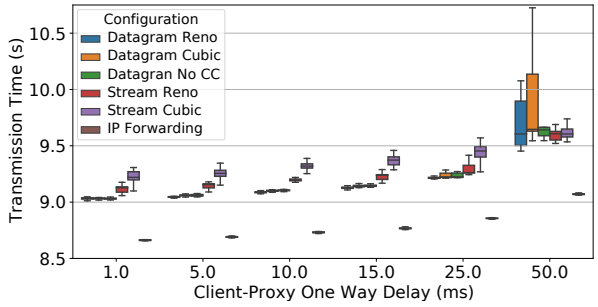

Figure 9: Transmission times for different client-proxy OWDs and congestion control algorithms when proxy-server OWD is 25 ms.

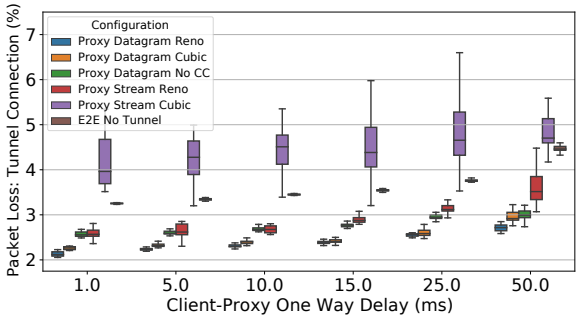

Figure 10: Tunnel Connection Packet Loss for different client-proxy OWDs and congestion control algorithms when proxy-server OWD is 25 ms.

stream mode shows improved transfer times relative to the datagram mode. This can be explained as the impact of congestion losses is higher at higher delays and therefore the more efficient recovery of those losses by the proxy becomes evident. For smaller delays, the larger encapsulation overhead of the stream encapsulation is impacting the longer transfer times relative to the datagram mode.

When the proxy uses stream-based encapsulation, end-to-end round trip times (RTT) of up to 2 seconds are observed. This delay is caused by the stream data buffer in the proxy, which is unbounded in our implementation. Losses repaired by the proxy are effectively converted into delay as seen by the server. While longer RTTs slow down the growth of the servers congestion window, the server congestion window keep constantly growing with losses that would cause a congestion window reduction. Therefore the congestion window becomes very large and significant amount of data is buffered at the proxy. Respectively, in the 50 ms OWD scenario, in stream mode performance is predominately governed by the proxy CC, while in datagram mode losses impacts both the proxy and server CCs.

Figure 9 also shows that transmission times are shorter when no proxy is involved compared to proxying with both stream and datagram based encapsulation. One cause of this observation are the CC interactions between the end-to-end and proxy-to-client connections. To measure the effect of such interactions, we compare

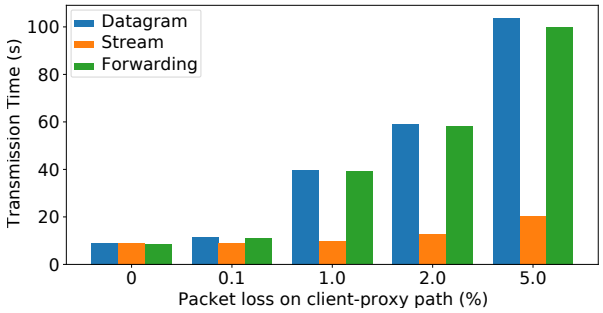

Figure 11: Transmission times for different loss scenarios using datagram, stream mode or no MASQUE.

runs where the CC is disabled in the proxy with runs where the proxy uses Reno and Cubic CCs. With a OWD of 1 ms no significant difference is observed between the different CC configurations in datagram mode. However, as the OWD increases, Reno shows slightly shorter transfer times than the other CC configurations. While this is not what we expected, the reason for this behavior can be observed in Figure 10. Higher packet loss rates are seen for both the Cubic and No CC configurations. These losses are self-induced congestion losses reducing efficiency. A similar behavior can be seen in stream mode where Cubic has a longer transmission time, but also a significantly higher packet loss rate. These properties are likely to change in the presence of cross traffic and non-congestion losses, but further study is required.

Encapsulation overhead introduced by the tunneling itself is another reason for the difference in transfer times. For datagram mode an overhead of approximately 3% is observed, see Figure 4. That corresponds to at least 240 ms at a bottleneck speed of 10 Mbps for a 10 MB message. Moreover, as the stack used in these evaluations is python-based and not optimized for performance, the HTTP/3 and MASQUE processing at the proxy and client stacks also impacts the maximum transmission rate and therefore increases the transmission time. Further work on performance optimization and comparison to other stack to quantify this impact is planned.

4.3 Using Streams for Local Loss Recovery

In this section we evaluate the impact of loss recovery due to the use of stream based encapsulation. It is expected that unreliable datagrams will be the most common encapsulation format used in MASQUE connections. However, if the link between the client and proxy is known to induce losses, local recovery on the access link can be beneficial to end-to-end performance. The following scenarios assume a low delay on the access link with a 5 ms OWD and a delay of 25 ms on the proxy-to-server link. Loss rates on the access link are varied from from 0.1% loss up to 5%.

Figure 11 compares the results for 0%, 0.1%, 2%, and 5% random loss on the local link and a bottleneck bandwidth between the client and proxy of 10Mbit/s. It shows the total transmission times in datagram and stream mode. In addition, we also compare these transmission times without MASQUE proxying. Both the inner and the outer connection use loss-based congestion control and therefore heavy impairments are expected. However, due to the

lower delay and therefore faster recovery between the client and the proxy, transmission time is improved in all scenarios when stream mode is used compared to both, datagram mode or when using a direct connection to the target server without MASQUE. Especially for high loss rates the transmission time improves significantly. This confirms our expectation that use of reliable streams in a MASQUE proxy setup is beneficial for links with notable link layer losses.

It is uncommon for modern fixed networks to exhibit non-congestion induced packet loss rates in the ranges tested here. However, in wireless networks such as wi-fi and cellular reasons such as handover and poor signal conditions [1] they are relevant. With the use of access network proxies it could be possible to simplify lower layers and let clients selectively enable reliability when needed.

5 CONCLUSION

In this paper we have presented a MASQUE implementation based on the aioquic QUIC and HTTP/3 stack. We have also introduced a docker-based emulation environment to test and evaluate various network scenarios.

In our evaluation we see that there is an overhead introduced by tunneling and that there are interactions to consider between the end-to-end and tunnel connections. In particular, it is important to consider the type of stream scheduling used by the proxy when sending data reliably. A naive scheduling approach might have significant performance impacts. We also see that reliable delivery tends to hide congestion signals from the target server, leading to both inflated RTTs and high resource consumption at the proxy.

While there are several issues to consider when transmitting data reliably, we also observe cases where it can have significant performance benefits. For mobile networks, this might provide an opportunity to accept network configurations with simpler link layer loss recovery schemes and only use local loss recovery when explicitly required by an application.

The presented results provide initial insights of this proxy-based setup and are limited by potential inefficiencies in the underlying stack and the simple topology of the emulation setup. We believe these results present valuable input to the on-going MASQUE standardization process.

ACKNOWLEDGMENTS

The Authors would like to thank Ingemar Johansson, Thorsten Lohmar, and Attila Mihály for feedback on this paper during its writing. We also like to thank Zaheduzzaman Sarker, and Zsolt Krämer that contributed to the implementation of the prototype and test environment.

REFERENCES

[1] Dziugas Baltrunas, Ahmed Elmokashfi, Amund Kvalbein, and Özgü Alay. 2016. Investigating packet loss in mobile broadband networks under mobility. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. 225–233. https://doi.org/10.1109/IFIPNetworking.2016.7497225

[2] Mike Bishop. 2021. *Hypertext Transfer Protocol Version 3 (HTTP/3).* Internet-Draft draft-ietf-quic-http-34. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34 Work in Progress.

[3] Roy T. Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231. https://doi.org/10.17487/RFC7231

[4] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://doi.org/10.17487/RFC9000

[5] Zsolt Krämer, Mirja Kühlewind, Marcus Ihlar, and Attila Mihály. 2021. Cooperative Performance Enhancement Using QUIC Tunneling in 5G Cellular Networks. In *Proceedings of the Applied Networking Research Workshop* (Virtual Event, USA) *(ANRW '21)*. Association for Computing Machinery, New York, NY, USA, 49–51. https://doi.org/10.1145/3472305.3472320

[6] Patrick McManus. 2018. Bootstrapping WebSockets with HTTP/2. RFC 8441. https://doi.org/10.17487/RFC8441

[7] Tommy Pauly, Eric Kinnear, and David Schinazi. 2021. *An Unreliable Datagram Extension to QUIC.* Internet-Draft draft-ietf-quic-datagram-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-quic-datagram-04 Work in Progress.

[8] Diego Perino, Matteo Varvello, and Claudio Soriente. 2018. ProxyTorrent: Untangling the Free HTTP(S) Proxy Ecosystem. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 197–206. https://doi.org/10.1145/3178876.3186086

[9] M. Seemann R. Marx, L. Niccolini. 2021. Main logging schema for qlog. In *IETF Internet-Draft*.

[10] David Schinazi. 2021. *The CONNECT-UDP HTTP Method.* Internet-Draft draft-ietf-masque-connect-udp-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-masque-connect-udp-04 Work in Progress.

[11] David Schinazi and Lucas Pardue. 2021. *Using Datagrams with HTTP.* Internet-Draft draft-ietf-masque-h3-datagram-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-masque-h3-datagram-03 Work in Progress.

[12] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Helsinki, Finland) *(SIGCOMM '12)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2342356.2342359

[13] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. 2011. An Untold Story of Middleboxes in Cellular Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 374–385. https://doi.org/10.1145/2018436.2018479

[14] Lisong Xu, Sangtae Ha, Injong Rhee, Vidhi Goel, and Lars Eggert. 2021. *CUBIC for Fast and Long-Distance Networks.* Internet-Draft draft-ietf-tcpm-rfc8312bis-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-rfc8312bis-04 Work in Progress.

[15] Xing Xu, Yurong Jiang, Tobias Flach, Ethan Katz-Bassett, David Choffnes, and Ramesh Govindan. 2015. Investigating transparent web proxies in cellular networks. In *International Conference on Passive and Active Network Measurement*. Springer, 262–276.

[16] Alexander Yu and Theophilus A. Benson. 2021. *Dissecting Performance of Production QUIC*. Association for Computing Machinery, New York, NY, USA, 1157–1168. https://doi.org/10.1145/3442381.3450103