



TUM SCHOOL OF COMPUTATION,  
INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# **eBPF-Assisted Relays for Multimedia Streaming**

Daniel Alexander Antonius Pfeifer



TUM SCHOOL OF COMPUTATION,  
INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# **eBPF-Assisted Relays for Multimedia Streaming**

## **eBPF-Unterstützung für Multimedia-Streaming-Netznoten**

Author:	Daniel Alexander Antonius Pfeifer
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Mathis Engelbart, M.Sc.
Submission Date:	15.08.2024

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.08.2024

Daniel Alexander Antonius Pfeifer

## Acknowledgments

# Abstract

In this thesis we propose a new relay setup for multimedia streaming that allows for avoidance of userspace processing by utilizing BPF programs in the Linux kernel. In a sample implementation we demonstrate the feasibility of this approach by designing a relay that is capable of forwarding packets between a server-side and a client-side QUIC connection while still being able to do adaptive bitrate streaming based on client congestion.

We show that this approach saves processing time and reduces latency compared to userspace processing with the relay still adhering to specifications of the QUIC standard and the 'Media over QUIC' (MoQ) draft. One limitation that is not addressed in depth in this thesis is the need for a de- and encryption hardware offload onto a SmartNIC to allow the BPF program to access the packet payload without any restrictions. Since the QUIC standard is still fairly new we are confident that a solution for a potential hardware offload will be found in future research.

# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Research Question . . . . .	1
1.2 Scope . . . . .	3
1.3 Structure of this Thesis . . . . .	4
<b>2 Background and Related Work</b>	<b>5</b>
2.1 QUIC . . . . .	5
2.1.1 Connections and Streams . . . . .	5
2.1.2 quic-go . . . . .	6
2.1.3 QUIC's Importance to Fast-Relays . . . . .	6
2.2 eBPF . . . . .	6
2.2.1 eBPF Hook Points . . . . .	7
2.2.2 eBPF Verifier . . . . .	8
2.2.3 Important eBPF Concepts . . . . .	8
2.3 Media over QUIC (MoQ) . . . . .	9
2.3.1 Solving Scaling versus Latency . . . . .	9
2.3.2 Design of a MoQ Relay . . . . .	10
2.3.3 moqtransport . . . . .	11
2.4 Adaptive Bitrate Streaming . . . . .	11
2.4.1 Mechanisms and Ideas . . . . .	11
2.4.2 Implications for Servers and Relays . . . . .	11
2.4.3 Impact on Media Streaming . . . . .	12
2.5 Related Work . . . . .	12
2.5.1 eQUIC Gateway . . . . .	12
2.5.2 Kernel Bypass . . . . .	12
2.5.3 Priority drop . . . . .	13
<b>3 Fast-Relays</b>	<b>14</b>
3.1 QUIC Adaptions . . . . .	14
3.1.1 Function-Pointer Style Additions . . . . .	14
3.1.2 Direct Changes to the Library . . . . .	15

3.2	eBPF Setup . . . . .	15
3.2.1	Different eBPF Programs . . . . .	15
3.2.2	Packet Registration . . . . .	21
3.2.3	Retransmissions of Forwarded Packets . . . . .	22
3.3	Userspace Synchronization . . . . .	22
3.3.1	Subscription and State Management . . . . .	23
3.3.2	Relay Caching . . . . .	23
3.4	Congestion Considerations . . . . .	24
3.4.1	Client Congestion . . . . .	24
3.4.2	Packet Filtering and Dropping . . . . .	24
3.5	Integration and Prototype . . . . .	25
3.5.1	Compatibility . . . . .	25
3.5.2	Source Code Repositories . . . . .	25
<b>4</b>	<b>Testing</b>	<b>27</b>
4.1	Setups . . . . .	27
4.1.1	Namespace Environment for Local Testing and Development . .	27
4.1.2	Physical Server Setup for Real-World Testing . . . . .	27
4.2	Testing and Results . . . . .	28
4.2.1	Delay Reduction of eBPF Forwarding . . . . .	28
4.2.2	CPU Utilization Comparison . . . . .	30
<b>5</b>	<b>Future Work</b>	<b>32</b>
5.1	Hardware Offload . . . . .	32
5.2	Compatibility Expansion . . . . .	32
<b>6</b>	<b>Conclusion</b>	<b>33</b>
	<b>List of Figures</b>	<b>34</b>
	<b>List of Tables</b>	<b>35</b>
	<b>Bibliography</b>	<b>36</b>

# 1 Introduction

The fact that online streaming tends to be slower than cable TV is likely something most people have already experienced first-hand. Live sports events, music shows or news broadcasts arrive an order of seconds later when streamed compared to using traditional cable connections. Now despite this delay not necessarily being a deal-breaker for most people, designing networks that tighten the gap between cable and streaming is still a worthwhile endeavor. With very optimized and fast networks already in place, we are at a point where providing such a faster information delivery is highly non-trivial. To do that we have even gone as far as developing completely new standards, such as QUIC. Those new standards aim to improve the shortcomings of some of the most fundamental protocols of the internet, one of them being TCP, which have been around more than 40 years.

Besides introducing new protocols, one could also look at existing setups and figure out how to trade some generality for a smaller delay when handling data. The ISO/OSI model, which is a foundational concept in networking, provides “a common basis for the coordination of standards development for the purpose of systems interconnection” [Sta94]. As one can imagine, such a “common basis”, even though convenient for large scale systems, can cause unnecessary overhead. In some cases additional speed-ups can be achieved by using more application-specific approaches. This thesis will consider one of such cases and explore the possibilities of avoiding or delaying certain processing steps of the ISO/OSI model in order to increase the overall speed of a network relay.

## 1.1 Research Question

As already mentioned above the usage of application specific approaches in networking allows for a reduction in latency. In this thesis we will consider a media streaming scenario that runs on top of QUIC by using the “Media over QUIC” (MoQ) transport protocol [Int24a]. The central question we will try to answer in this thesis will then be:

*How can we improve the performance of a relay in a media streaming scenario by using eBPF technology?*

with more specific sub-questions being:



1. How can we avoid the need to direct a packet through userspace?
2. How to handle the fact that packets are heavily encrypted?
3. What communication between userspace and eBPF program is necessary to stay coherent with potential state?
4. How can our approach be generalized to other protocols?

By using eBPF technology together with kernel hook points provided by the Linux-Kernel, we will try to find a setup that improves relay performance using eBPF programs that handle basic relay capabilities, such as packet forwarding and congestion control. Since the QUIC protocol is designed to handle a large portion of its workload in userspace we look into possibilities of delaying any userspace processing until **after** the packet has been forwarded to the client. This way the raw delay that the packet experiences from the initial media server to the client could be reduced. However, since QUIC is a connection oriented protocol, we need to make sure that the QUIC connection state stays coherent despite the additional processing steps done by the eBPF program. We will investigate which additional processing steps are needed in our case, how they compare to challenges when expanding our approach to other protocols and how they can be implemented in an eBPF program.

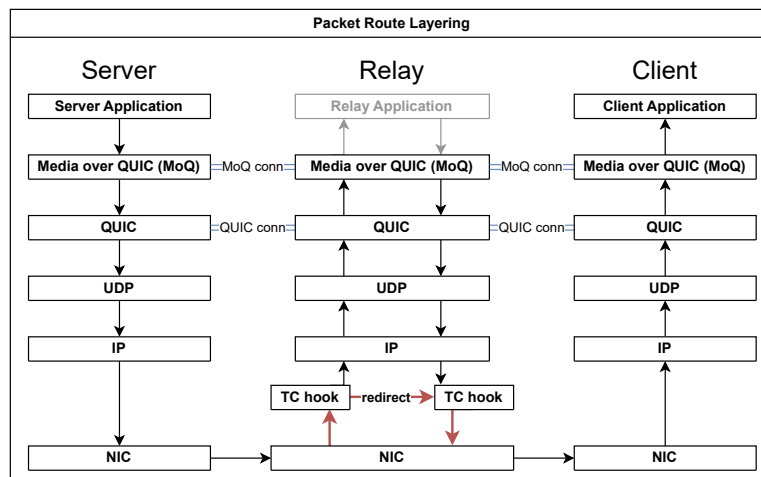


Figure 1.1: Conventional networking layers a packet passes. The red loop indicates again the “short-cut” that is utilized by the fast-relay (eBPF packet-forwarding - no need to go up to userspace).

## 1.2 Scope

The main improvement this thesis aims to achieve is shortening the critical path a packet takes from a media server to a client. This will be done by avoiding the immediate need of a packet traversal up the network stack to the application layer. Instead, any communication with the application layer will happen in a delayed fashion (after the packet was sent) by utilizing eBPF-Maps for storing any necessary (meta-) information. The main reason this communication between userspace and eBPF-program is required lies in the fact that relays in MoQ are an application layer concept. That means the QUIC connections to from relay to server and from relay to client will be different and the packets that have been eBPF-forwarded to egress directly will need changes in their header data in order to match the state of the outgoing client connection.

This approach is highly dependent on the used standards and protocols. This thesis will operate on top of the QUIC protocol [Int24b] and the “Media over QUIC” (MoQ) transport protocol [Int24a]. For the application layer the quic-go library [See24] will provide the implementation and any additional (non-eBPF) program will also be written in Go. Since the setup is dependent on retrieving data from eBPF-Maps the QUIC library providing the implementation will need some adaptations. We will mainly introduce simple function pointer style additions that allow the adapted library to be run both with and without the eBPF setup. The developer of the relay will then also have more freedom to setup the eBPF part of the relay as they see fit since the Go code that will interact with eBPF parts will also have to be provided by said developer.

Additionally we will run a performance analysis on our implementation of the relay to confirm the potential this approach has. These performance tests will look at the raw delay speedup as well as the impact on CPU utilization this setup has. All the tests will be done in a lab-like environment to isolate the performance changes as best as possible from any outside noise. The payloads used will only contain dummy data since our approach does not interfere with payload contents and there is no need for creating and using real media stream data.

Despite our approach only considering QUIC and MoQ, we will argue that the general idea of our setup will be independent of any of these protocols and can be changed to fit ones needs.

With this we will provide answers to the research questions regarding packet-redirection, communication between userspace and eBPF as well as setup-generalization. Regarding the question on how to handle the encryption of the packets, we will not focus on this since we did not find a suitable hardware offload that would have allow for en- and decryption after and before the used eBPF hook points respectively. Instead we will emulate this behavior by turning off the encryption in the QUIC library itself which will provide a similar result.

### 1.3 Structure of this Thesis

In chapter 2 this thesis will provide some overview of used technologies and related ideas. Section 2.1 will give an introduction to the QUIC protocol and its main features and section 2.2 will provide an overview of eBPF technology together with features related to our approach. Section 2.3 will introduce the ‘Media over QUIC’ (MoQ) transport protocol which will be used for our application level relay setup. After that section 2.4 will explain the ideas and challenges of adaptive bitrate streaming while section 2.5 will mention some work related to the aforementioned topics. What will follow in chapter 3 is a detailed description of the setup that allowed us to improve relay performance. We will look at the adaptations to the used QUIC library in section 3.1 as well as our eBPF setup in 3.2. Besides those two we will also look at some more specific details and challenges in the subsequent sections. In chapter 4 we will then provide a basic performance analysis of our setup to show current improvements and limitations. Finally we will conclude with a summary together with some ideas for future work in this field in chapters 5 and 6.

## 2 Background and Related Work

### 2.1 QUIC

Many fundamental internet protocols still used today have been around for a very long time. For example the Transmission Control Protocol (TCP) has been used as the backbone of the internet for more than 40 years. It has been designed to be reliable and to provide a connection-oriented way of transmitting data, but the modern environment of the internet with needs like lower latency, better multiplexing or improved security makes it hard for TCP to keep up. Limitations in the design and resulting issues like head-of-line blocking have raised demand for a newly designed protocol that can keep up with the modern internet. All of these issues paired with the want for a more flexible development cycle led to new creations. QUIC, which started off as the ‘Quick UDP Internet Connections’ protocol, and has since been standardized by the IETF with QUIC being its own trademark, is a transport layer protocol built on top of UDP that is designed to be reliable, cryptographically secure and more performant than TCP. QUIC, partly because it operates both in user- and kernel-space, has been designed to allow for a more rapid deployment cycle than TCP. Similar to TCP it is a connection-based protocol that uses TLS for encryption [Lan+17]. Already back in 2018, QUIC was the default protocol for the Google Chrome browser which, at the time, made up 60% of the web browser market [Lie18]. A little over two years later, Facebook, now Meta, was using QUIC for more than 75% of their internet traffic which led to improvements regarding request errors, tail latency and header size [JC20]. As of May 2024, QUIC already made up 8.0% of all internet traffic with support from pretty much every major browser [W3T24; Rod24]. With big players like Google, Meta, Microsoft or YouTube putting emphasis on using QUIC to improve their services, this number is likely to increase even further.

#### 2.1.1 Connections and Streams

Since QUIC is a connection-based protocol, some initial overhead to establish a connection is needed. However, the design incorporates some features that aim for an efficient way of establishing connections, e.g. by using 0-RTT (zero round-trip-time) handshakes. Latency improvements like the 0-RTT handshake however come at the cost of security, since that opens the door for replay attacks. Another part where QUIC tries to optimize connection management is the use of streams. Streams are designed to be lightweight and can be opened without the need of a handshake. This goes as far as one single

packet being able to open a new stream, transferring stream data as well as the closing the stream again. This allows for new techniques to improve data transmission and will also be part of the fast-relay setup in this thesis. Aside from streams, apparent since QUIC is based on UDP, it is also possible to send data via unreliable datagrams. This further improves versatility of the protocol and allows for new ways of optimizing data transmission.

### 2.1.2 quic-go

There are many implementations of the QUIC protocol available providing libraries for a lot of today's most popular programming languages. The implementation we settled on for this thesis is the quic-go library which provides a pure Go approach to implementing the standards of RFC-9000, RFC-9221 as well as some others which are not important for our usecase. However, since we need some special behavior of the userspace part of QUIC we will introduce some modifications into quic-go. Those modifications will be explained further in section 3.1.

### 2.1.3 QUIC's Importance to Fast-Relays

The QUIC protocol will be a fundamental part of the fast-relay setup in this thesis, yet the ideas used to make relays faster is not limited to QUIC and can be extended to other protocols as well. QUIC is chosen as an example protocol due to its increasing popularity which offers big potential in early adoption and deployment of fast-relays. Also the easy incorporation of changes into libraries providing RFC implementations makes it a good starting point for experimenting what can and cannot be done regarding our research questions. This includes the possibility of neglecting the difficulties that the heavy encryption of QUIC brings with it, by just turning off the related functionality.

## 2.2 eBPF

In 1992 a technology called 'Berkeley Packet Filter' (BPF) was introduced into the Unix kernel. By using BPF it is possible to attach a small BPF-program to some pre-defined hook points in the network stack of the kernel and filter packets there in a stateless manner. This provided more efficiency since the packets did not need to be copied into userspace anymore but could directly be processed in the kernel. A need for better tracing capabilities of the Linux kernel led to the development of an extended version of BPF called "eBPF" which was introduced in 2014 and heavily influenced by a tracing tool called "dtrace" [Tig].

### 2.2.1 eBPF Hook Points

The Linux kernel offers several hook points where eBPF-programs can be attached to. There are two prominent ones that we considered for our suggested setup. The first one allows one to access the Traffic Control (TC) subsystem while the second one allows one to access the eXpress Data Path (XDP) subsystem.

XDP would generally provide a better performance since it is located lower on the network stack, namely directly in the NIC driver, than the TC-hook point, which is located in the link-layer. TC on the other hand offers a more versatile way of packet processing since the used `sk_buff` provides access to metadata that is not available when using XDP and its `xdp_buff`. What ultimately led us to choose TC over XDP was however the fact that XDP only allows ingress packet processing while TC allows for both ingress and egress. That means that with XDP we would not have been able to redirect packets to be handled at egress which is crucial for the fast-relay setup we are aiming for.

Figure Figure 2.1 illustrates again the relative positions of the TC and XDP hook points in the network stack.

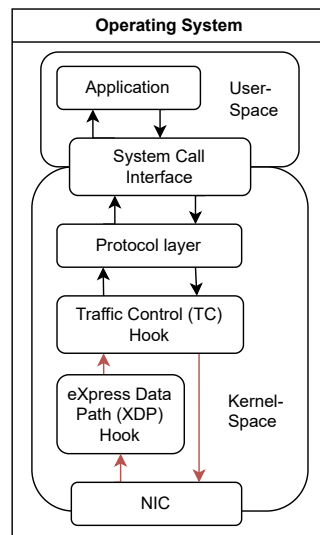


Figure 2.1: Abstracted view of Traffic Control (TC) and eXpress Data Path (XDP) hook points in the Linux kernel network stack. The red loop indicates the ‘short-cut’ that is utilized by the fast-relay. TC hook allows redirection directly to egress while XDP hook is only available for ingress processing.

### 2.2.2 eBPF Verifier

Since eBPF programs are executed in the kernel it is quite obvious that extensive security checks need to be in place to ensure that the kernel does not experience problems like infinite loops, accesses to invalid memory locations or other security related issues. This explains the existence of the so-called ‘BPF-verifier’ which inspects every BPF-program for its safety by simulating possible program paths, looking at the graph representation of the program and more [Fou]. This imposes some restrictions on the complexity of the programs that can be used within the kernel. In our case this did not impose too many issues though since we do not rely on very complex control structures.

### 2.2.3 Important eBPF Concepts

One of the most important concepts in eBPF which we do however use quite extensively is the ‘eBPF-map’. Such a map boils down to a section in memory that is reserved for the eBPF-program and which can be used as a key-value store for arbitrary data. This part of memory can then also be accessed from userspace and thus provides the main way of communication between the eBPF-program and our application. When we define an eBPF-map we can choose between different types as well as configure size, key-type, value-type and the way the map is stored. An example of two eBPF-map definitions can be seen in Figure 2.2. It shows two different types of maps, a hash map and a ring buffer, that are used in our fast-relay setup. Those and some other relevant map types are listed in table Table 2.1.

```
1 struct {
2     __uint(type, BPF_MAP_TYPE_HASH);           // Hash map
3     __type(key, struct client_info_key_t);      // Specific client key
4     __type(value, uint32_t);                   // 32 bit id
5     __uint(max_entries, MAX_CLIENTS);          // Maximum number of clients
6     __uint(pinning, LIBBPF_PIN_BY_NAME);       // Pin by name to the tc filesystem
7 } client_id SEC(".maps");
8
9 struct {
10    __uint(type, BPF_MAP_TYPE_RINGBUF);         // Ring buffer
11    __uint(max_entries, MAX_PACKET_EVENTS);     // Maximum number of packet events
12    __uint(pinning, LIBBPF_PIN_BY_NAME);       // Pin by name to the tc filesystem
13 } packet_events SEC(".maps");
```

Figure 2.2: Exemplary eBPF map definitions.

Type	Description
BPF_MAP_TYPE_HASH	A hash map where keys and values can be arbitrarily defined.
BPF_MAP_TYPE_PERCPU_HASH	A hash map with separate value slots for each CPU, providing improved performance in multi-core environments.
BPF_MAP_TYPE_ARRAY	An array map that allows random access to elements by index.
BPF_MAP_TYPE_PERCPU_ARRAY	An array map with separate value slots for each CPU, useful for per-CPU data storage.
BPF_MAP_TYPE_RINGBUF	A ring buffer for implementing high-performance data queues.

Table 2.1: Some eBPF map types. (defined in `/usr/include/linux/bpf.h`)

## 2.3 Media over QUIC (MoQ)

On the application layer we will use the Media over QUIC (MoQ) protocol which is as of summer 2024 still being in the process of standardization by the IETF. MoQ targets live-streaming and real-time collaboration applications like Zoom, Microsoft Teams, or Google Meet. It is built on top of the QUIC protocol with the possibility of using WebTransport for browser support. A general publisher/subscriber model is used and the draft tries to combine performant approaches from protocols like RTP (for real-time features) and HLS/DASH (for scalability).

### 2.3.1 Solving Scaling versus Latency

For a long time now there have been two different camps with regards to media-data-transmission-protocols and -setups. One is heavily focused on low latency while the other is aiming for high scalability. Systems of the former kind include real-time collaboration tools like aforementioned Zoom, Teams, or Meet. The latter ones are often huge platforms like Twitch, YouTube or Netflix which need to reach millions of users at the same time. The one thing both have in common is that it turns out to be difficult to incorporate both low latency and high scalability into the system at the same time. The MoQ protocol tries to solve this by providing a setup that is both low-latency and highly scalable. To achieve this it supports performance enhancing approaches like relay caching or support for adaptive rate mechanisms.



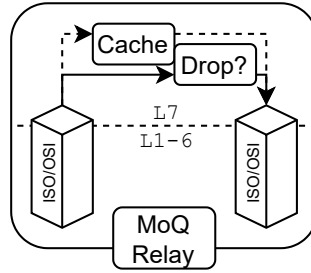


Figure 2.3: Rough MoQ relay architecture.

### 2.3.2 Design of a MoQ Relay

The charter for the IETF working group describes what MoQ, and therefore also a relay that wants to meet the MoQ requirements, needs to support. These requirements for the publication- and distribution-setup mention the support of multiple formats, dynamic rate adaption mechanisms (e.g. used for congestion handling) as well as cache-friendly mechanisms.

Figure 2.3 gives a visualization of the rough architecture of a MoQ relay. It hints at key components like the relay level cache and the congestion handling mechanism. What one can also infer is the place of MoQ in the OSI-model namely at the application layer which itself builds on top of lower level protocols like QUIC, UDP, IP and Ethernet. In figure 2.4 and figure 2.5 one can see a comparison between the delayed fan-out of the media content caused by the MoQ relay and a more traditional setup. The former is able to reduce the traffic through a network by making multiple transmissions of the same data between server and relay obsolete.

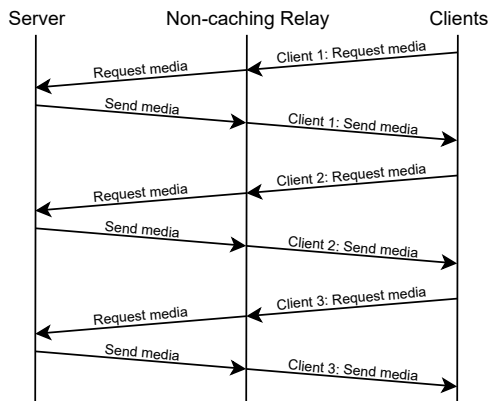


Figure 2.4: Multiple data transmissions for different clients.

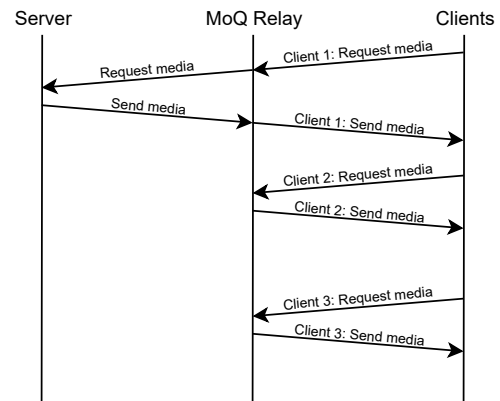


Figure 2.5: Only single data transmission towards the relay.

This caching mechanism fits quite naturally into our proposed eBPF setup since we

will need to communicate packet data between kernel- and user-space anyway to keep the QUIC library in a consistent state. In addition to that the congestion handling functionality of the MoQ relay can also be integrated fairly easy within eBPF. This is because packet dropping is ultimately one of the main use cases of plain eBPF programs and as such easy to implement. In a later section will go into detail on how the eBPF setup actually uses mechanisms like priority fields for this and how those meet the standard specifications.

### 2.3.3 moqtransport

In order to use the MoQ protocol in our setup we will make use of a library that implements the MoQ transport protocol (moqtransport or MoQT) in Go [PE24] The goal of MoQT is to define a media transport protocol that is operating on top of QUIC and WebTransport which is providing the concrete designs that the charter of the MoQ working group is aiming for. This includes, for example, actual structure of the different message types or error handling in case of a wrong state. As of writing this thesis the MoQT draft is in its fifth version [Int24a].

## 2.4 Adaptive Bitrate Streaming

Multimedia streaming is a big part of the internet and many optimizations have been developed to improve the quality of service for the end-users. This includes considering (in real-time) parts of the clients connection state, such as available bandwidth, and adapting the rate at which a server sends data. Such a process is called ‘Adaptive Bitrate Streaming’ and is employed in many of todays streaming setups. An example setup can be seen in Figure 2.6 where the an encoder creates streams with different resolutions and edge server that manages the connections to the clients can switch between those streams based the connection state of a specific client. Youtube and Netflix are examples where, although more complex, similar setups are used to provide a better user experience.

TODO: more info (what to explain?)

### 2.4.1 Mechanisms and Ideas

TODO: actually needed or only to have 3 subsections everywhere?

### 2.4.2 Implications for Servers and Relays

TODO

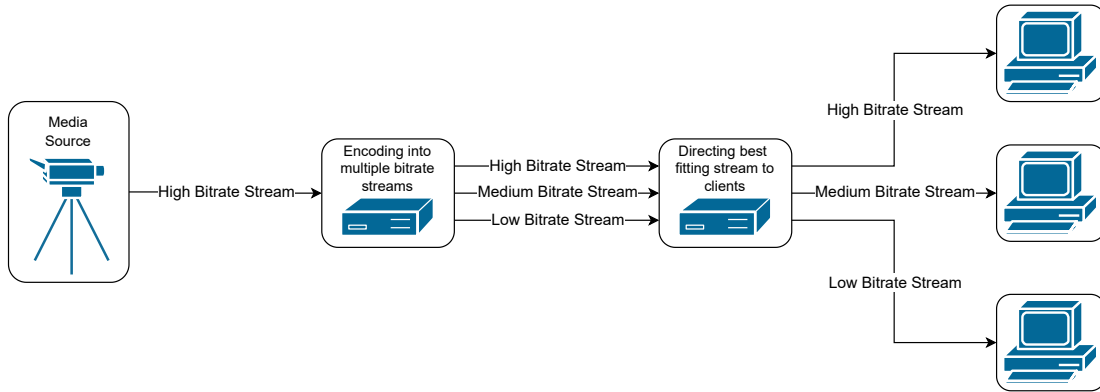


Figure 2.6: Behind the scenes multiple streams with different resolutions exists to allow adapting to a users bitrate.

### 2.4.3 Impact on Media Streaming

TODO

## 2.5 Related Work

### 2.5.1 eQUIC Gateway

There have been previous publications on making QUIC more efficient by using BPF programs such as [PVV21], where a BPF program is used together with the Linux eXpress Data Path (XDP) to filter packets based on information provided by the userspace. This approach provided significant performance improvements with an increase of throughput by almost a third and a reduction of CPU time consumption caused by filtering packets by more than 25%. This shows that a setup leveraging Linux kernel features such as BPF has a lot of potential to improve current infrastructure.

### 2.5.2 Kernel Bypass

Another interesting approach which follows a similar idea of speeding up packet processing by avoiding the Linux network stack is [Tyu22]. The difference in this work is that DPDK is used to bypass the network stack to then process packets in userspace instead of using BPF programs like this thesis does. This, for example, offers more flexibility as the userspace program is not as limited (e.g. by the BPF verifier) as the BPF program but might also lead to slightly more system calls, especially in the setup of a system, when user- and kernel-space need to communicate.

### **2.5.3 Priority drop**

The idea of dropping packets based on their priority to adapt a connection in a congestion event has also been around for a while. [KWF03] explores this in more detail. Mainly improvements like a more tailorable congestion handling than the sole usage of discrete video quality levels as well as an improvement to, potentially randomized, frame dropping are discussed. This thesis, similar to [KWF03], will not focus on how the priority for packets is determined but rather on how those marked packets are handled. For this it is assumed that a higher level protocol has correctly determined the packet priorities and can handle the drop of packets with lower priority in case of limited bandwidth.

## 3 Fast-Relays

### 3.1 QUIC Adaption

As was already mentioned in the previous chapter, our setup requires some adaptations to the quic-go library. One initial change that was necessary was to turn off packet encryption, happening within quic-go. Given that we operate on the QUIC-header data within the eBPF-program we need access to fields that are encrypted using QUICs header-protection. For obvious reasons sending unencrypted packets is not something that would be wanted in a production environment but for our setup it is required since no fitting hardware offload was available at time of writing that would have allowed us to ‘push down’ encryption and decryption onto a smartNIC. Given that such a hardware offload is added in the future, the encryption and decryption can be turned on again which makes this change more of a temporary solution to show the feasibility of our approach.

#### 3.1.1 Function-Pointer Style Additions

Another type of change that we needed to introduce into the quic-go library is caused by connection state management. We essentially added support for communication with the eBPF-program by using an approach similar to C-style function pointers. On multiple locations we added conditional function calls like the one depicted in Listing 3.1. The function that is called here will be defined by the developer of the relay and therefore allow for customizability without the need for changing the library itself.

```
1 /* Function pointer call within actual quic-go code */
2 if packet_setting.ConnectionUpdateBPFHandler != nil /* && potentially other
   conditions */ {
3     packet_setting.ConnectionUpdateBPFHandler(connId.Bytes(),
         uint8(connId.Len()), p.connection)
4 }
```

Listing 3.1: An example of a function-pointer addition to the quic-go library.

```
1 /* Function pointer signature definition within additional config file */
2 ConnectionUpdateBPFHandler func(id []byte, l uint8, conn QuicConnection) = nil
```

Listing 3.2: Only the signature will be defined within the library itself.

The definition of the function that the developer of the relay wished to be executed at the specifically defined points will be defined locally in the relay code and provided to the configuration of the quic-go library. An example of how this could look like is shown in Listing 3.3.

```
1 /* Definition of the function within the local relay code */
2 func UpdateConnectionId(id []byte, l uint8, conn
   packet_setting.QuicConnection) {
3     /* handle the connection update by interacting with the eBPF-program */
4 }
5
6 /* Providing the function to the quic-go library */
7 func main() {
8     /* ... */
9     packet_setting.ConnectionUpdateBPFHandler = common.UpdateConnectionId
10    /* ... */
11 }
```

Listing 3.3: An example of how the addition looks on the relay side.

The need for these additions arises since the eBPF-program works with its own copy of the current state of a connection. This, for example, includes the connection-id that will be used when changing the packet header before sending it out. Since a connection-id can change, i.e. be updated or retired, during the lifetime of a connection we need a way to inform the eBPF-program to no longer use outdated state-information. These function-pointer style additions provide a minimal way of adding such functionality without limiting flexibility or adding too much application specific code to the library itself as it would be the case if the library would access the eBPF-Maps directly.

### 3.1.2 Direct Changes to the Library

TODO

## 3.2 eBPF Setup

### 3.2.1 Different eBPF Programs

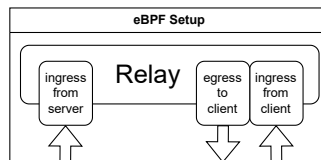


Figure 3.1: The relay has to be equipped with three BPF programs.

In order to allow the relay to forward packets independently of the userspace, we need to equip the relay with three BPF programs as seen in figure. Those three programs are

- a program that handles incoming traffic **from** the clients (client ingress),
- a program that handles outgoing traffic **to** the clients (client egress) and
- a program that handles incoming traffic **from** the video server (server ingress).

Their responsibilities then are

- handling the initial registration of new clients and storing their information such as MAC addresses in a BPF map,
- intercepting the packets from the video server, duplicating and redirecting them to the egress program (as well as sending one unaltered packet to userspace for state management purposes),
- receiving the redirected packets at egress, altering them using the client specific data, deciding (based on packet priority and client congestion) if a packets should be dropped or sent, storing info on sent out packets for future congestion control purposes and finally sending them out to the clients.

This setup allows us to separate any state management and congestion control from the actual packet forwarding and thus makes leaving out any immediate userspace processing possible.

Following is a more detailed description of the responsibilities of each of the three programs.

### **Client Ingress**

The ingress eBPF program initially does some simple packet inspection on every incoming packet looking if the packet uses the correct protocols and addresses the right application layer. This is done by initially parsing the Ethernet, IP and UDP headers, if present, and checking if the port matches the listening port the relay application is listening on. This means the correct port is to be defined prior such that the eBPF program can associate a single or (multiple) relay instance(s) with the correct port. In our case then, since we use QUIC, the program will check for QUIC long header packets that setup an initial connection and saves the transmitted state information such as connection-id, stream related states, etc. in an eBPF map together with information that is not directly known by the userspace such as the MAC address of the client. Saving data like the MAC address directly once the connection is set up allows to omit any further Address Resolution Protocol (ARP) steps later on.

## Server Ingress

Another ingress related program, this time for packets coming from the video server, is needed to handle packet duplication and forwarding. This program will receive the actual video packets from the server and then, based on an internal counter of how many clients actually want to receive the video, duplicate the packet accordingly. The counter of clients will be updated by the userspace once a new connection is fully established and the client is ready to receive the actual video data. This might potentially cause some miniscule delay when updating the counter but sending cached video data to the client for a brief moment when updating the counter could be a solution for that. Figure 3.2 shows the packet duplication and forwarding process high level for an example setting with three clients that want to receive the video data. In the ingress program from the server we need to consider a few things to ensure the correctness of our approach. These are:

1. The program can **only** forward packets that containing video data and must not forward any other packets that contain e.g. control data.
  - a) This is fairly easy to achieve by doing some header inspection of the QUIC header which contains the packet type. Also since payload is generally sent using 0-RTT- (i.e. short-) headers there is no need to consider long-header packets.
2. The program should pass an unaltered copy up to userspace to allow the QUIC library to gain knowledge of the packet and handle any state changes accordingly.
  - a) Generally speaking this is not strictly necessary as one could just have a separate setup of registering packets that came from the server but as this is not needed it is considerably easier to just pass the packet up to userspace and let the library handle it normally. This does not impose any additional overhead as the forwarding of any duplicate packets happens independently of course.
  - b) Any packet that has been identified as not part of our dataflow should of course be passed up to userspace without being forwarded to egress.

Duplicating and forwarding incoming packets happens using that were identified as payload can be done using `bpf_clone_redirect(skb, egress_ifindex, 0)` which is a helper function that allows one to clone the packet buffer provided as the first argument and add it to the queue of the interface that is provided as the second argument. The last argument allows to specify additional flags. Aside from `bpf_clone_redirect` there are also other helper functions regarding packet redirection with slightly different behavior so it is crucial to choose the appropriate one to get the desired outcome. Table 3.1 shows them together with a brief description of their behavior:

Based on the descriptions of the redirection helper functions mentioned in table 3.1 it becomes clear that `bpf_clone_redirect` is the only suitable for our use case. This is



Helper	Description
<code>bpf_clone_redirect</code>	Clones and redirects a packet to the interface associated with the provided index.
<code>bpf_redirect</code>	Redirects a packet to the interface associated with the provided index. The packet is not cloned (no underlying call of <code>skb_clone()</code> ) so it is slightly more efficient (25% pps increase according to commit message). The packet is also not redirected immediately but after the function finishes.
<code>bpf_redirect_peer</code>	Similar to <code>bpf_redirect</code> but instead of redirecting the packet to the interface provided as a parameter it is redirected to its peer device. This works only between different netns to allow for an efficient “ingress to ingress netns switch”. The switch is more performant since the packet does not need to go through the CPU backlog queue.
<code>bpf_redirect_neigh</code>	Again similar to <code>bpf_redirect</code> but allows to redirect a packet to another net device. This helper does also fill in all the correct L2 addresses of the neighboring subsystem. Internally this executes a neighbor lookup to find the needed L2 information.

Table 3.1: Helper functions for packet redirection.

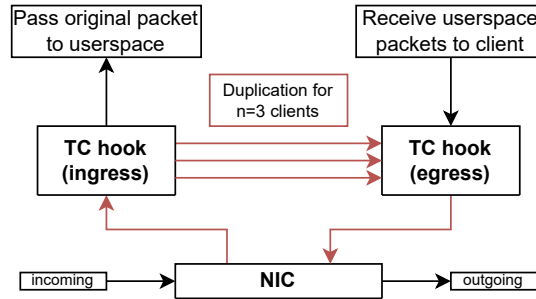


Figure 3.2: Duplication and forwarding of video packets to egress directly from ingress.

because we need the cloning aspect since we essentially want to duplicate the incoming packets multiple times. Also the redirection to another namespace or another net device as provided by `bpf_redirect_peer` and `bpf_redirect_neigh` respectively is not needed in our case since we operate in the same relay-namespace throughout the whole process.

### Client Egress

The central part of the eBPF setup where all the state-management and forwarding of the other eBPF programs comes together is the program that handles the outgoing traffic towards the clients. The client egress program sees every packet that leaves the relay, which includes packets that have been redirected by the ingress program as well as packets that have been generated by the relay itself (i.e. come from the relay userspace). This means that the program essentially merges two streams of packets into one stream that needs to be in a consistent state. This interleaving of packets grows the requirements of the program to the following list. Bold numbers indicate that a requirement is caused by the interleaving of packets.

1. Obviously, similar to the other eBPF programs any packet that is not part of our traffic should be passed on normally without any further processing.
2. In case the packet is QUIC (for the correct client connection) the packet number needs to be changed to a program internal counter to avoid issues of reusing packet numbers. This is the only place where we can guarantee sequential packet numbers since neither userspace nor the server know what the highest packet number sent at any moment in time is. As there is no way of synchronizing lookups (e.g. using eBPF maps) of information between userspace and eBPF program changing it right before sending it out is the most efficient way.
3. In case the packet (additionally to being QUIC for a client connection) contains stream frames we need to do a similar translation of the stream id. Reasons and methods for this are the same as for the packet number translation. Figure 3.3

shows a flow diagram of the stream id translation process. Important steps include checking if the stream id translation is already existing as well as checking if the packet is a retransmission. The former is necessary in case a payload is split into multiple packets and the latter is necessary since the retransmission physically come from the relay but should be treated as if they came from the server.

4. Again in case the packet is QUIC and for the right connection the program needs to read the priority of the packet and decide via map-lookup if the connection allows for sending packets with the given priority.
5. In case the packet is a redirected media stream data packet the program needs to find out which client is the receiver. This is done by saving the client id in some part of the packet that will be overwritten at egress (namely the connection-id) before redirecting the packet. The egress program, knowing where the id is saved in case of a redirected packet, can then lookup the correct address data of the client (i.e. MAC address, IP address, etc.) and overwrite the respective fields in the buffer before sending it out.

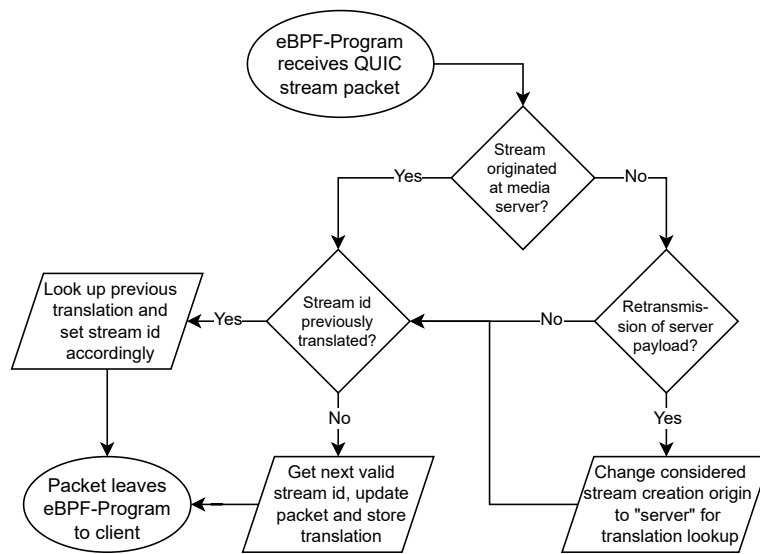


Figure 3.3: Flow diagram of stream id translation process.

The aforementioned packet number translation means that the packets sent by the userspace are likely to have a different packet number than the one chosen by the QUIC library. This might lead to inconsistencies again when receiving acknowledgements but can be avoided by remembering the translation in a map as well as storing only packet-objects in the Go library that have the correct state. This technique of “storing”

a packet that the userspace either has sent itself or does not know about since it comes from the media server will be referred to as “packet registration”. This initially gives a brief window where a packet was sent out but is not saved in the history of the QUIC library but once the packet is then processed by the userspace routine handling the registration, any incoming ACKs for this packet can be processed correctly. The next section will go more into detail on how the packet registration works. Later we will also look at the implications of sending a packet the library does not know on retransmissions. Even though the stream id translation works very similar to the packet number translation it does not have the need for any additional work after the packet has been sent out since our approach uses unidirectional streams only and thus the relay does not care about changes in the used stream id.

### 3.2.2 Packet Registration

In order to make the congestion control algorithm that is running in userspace usable we need to inform the QUIC library about the forwarded packets. This again happens via BPF maps and a separate go routine that continuously polls new entries in the map and processes them. Entries are then added to the packet history to allow the receipt of ACKs. Besides that, the congestion control algorithm will be informed about the forwarded packet in order to be able to react to potential congestion events. Figure 3.4 visualizes the setup for this process.

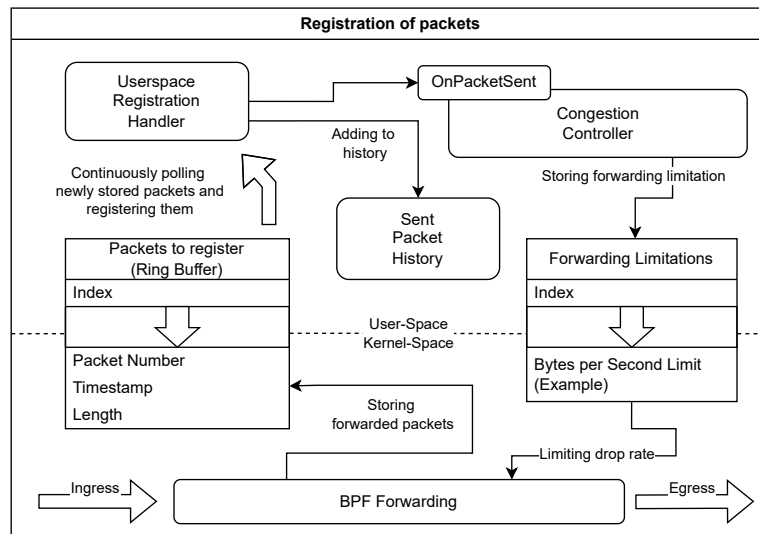


Figure 3.4: Internal setup for registering forwarded packets as well as incorporating forwarding limitations for the BPF program.

### 3.2.3 Retransmissions of Forwarded Packets

TODO

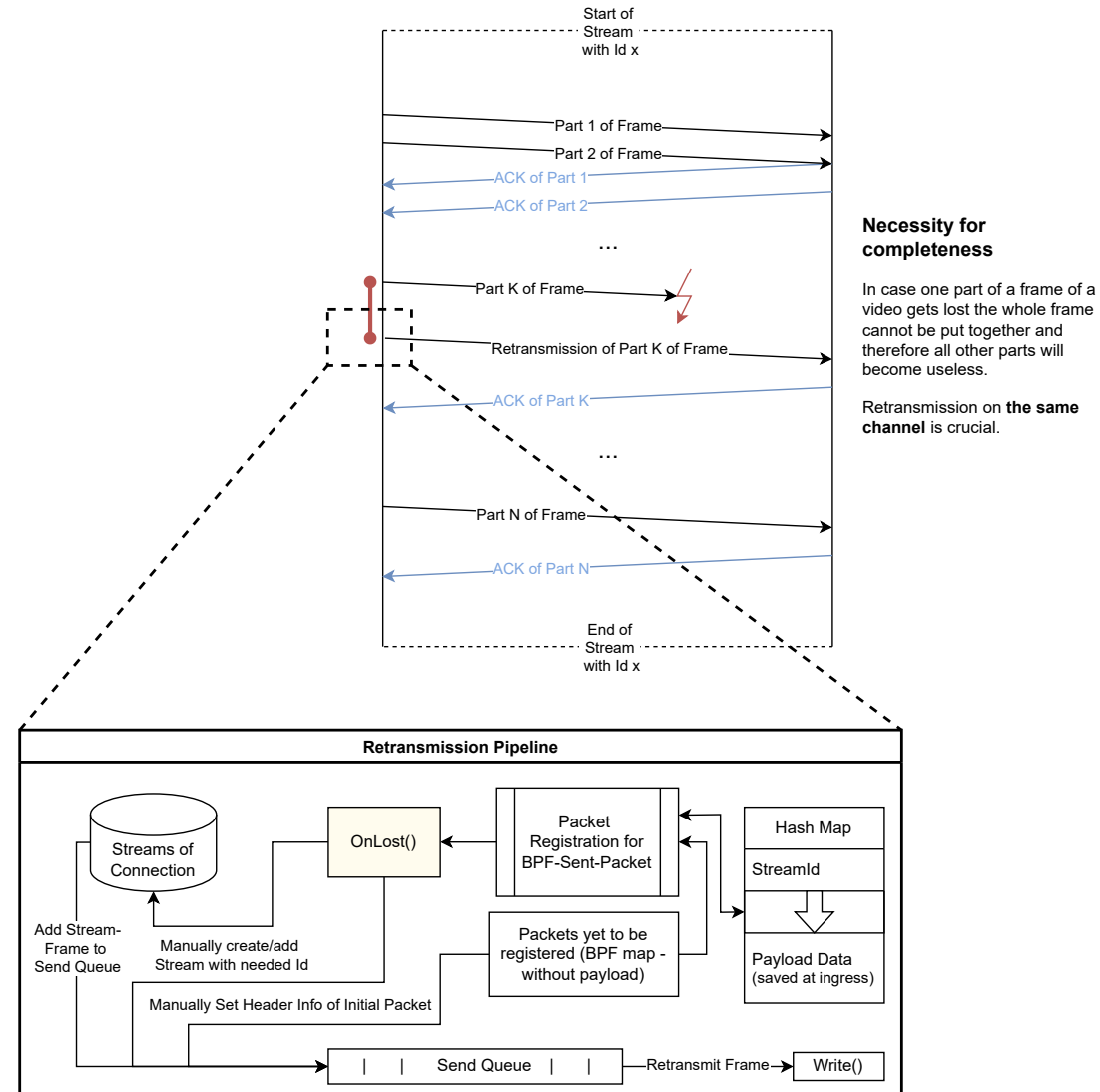


Figure 3.5: Internal setup for retransmitting packets forwarded from server connection.

## 3.3 Userspace Synchronization

Since one of the main ideas we propose is to avoid passing a packet all the way through the network stack up to userspace at the relay, we create the problem that the application itself is not aware of packets that are being sent. In order to conquer

this issue, we suggest a setup that establishes communication between the application- and the eBPF-program. To keep the improvement we gained by not passing packets to userspace during the critical path, this communication will happen in a delayed fashion that is decoupled from the actual sending of media data.

The main resource we use for this communication will again be eBPF maps that will contain information on the time a packet was sent together with protocol specific data like packet numbers or stream identifiers (for both packet number and stream id the old and the new, i.e. translated, values will be stored).

### **3.3.1 Subscription and State Management**

As already mentioned in section 3.2, an eBPF program handling incoming traffic from the client will save client connection information like MAC address, IP address, and so on in a map for later access. Also an internal counter will give each client a unique identifier. With that the only thing that happens in terms of communication between the application and the relay in case of a new subscription is that the application will update the number of clients counter that is accessed by the eBPF program and used for packet duplication purposes.

Regarding stream state management there is also little need for communication since the server is expected to use QUIC's unidirectional streams for sending the media data. That means the relay does not really need to know about the stream details except in case it has to trigger a retransmission. If that is the case the stream id contained in the packet (meta-)data that was read from the eBPF map will be used to manually create a stream with the correct id. It is important to manually set the correct id since the relay might not use the same id for the next unidirectional stream it opens. Also the client expects the retransmission to be sent on the same stream id as the original packet since retransmission happen within the same stream-context.

### **3.3.2 Relay Caching**

Regarding the caching of data within a relay, that is required by the MoQ standard, we handled this by passing on an unaltered copy of any incoming packet from the server to the application at the same time we forward all the other packet copies to egress. Now the application is able to receive any data from the server as if this was a normal connection and store, e.g. the last second(s) worth of data in a cache. Then the relay could, once a new connection is established, parallel to incrementing the kernel counter representing the number of clients also send out cached data already so that the client receives it as early as possible.

One aspect for such a setup that we still left open is the point in time where the relay should stop sending cached data since the forwarded data is up-to-date. This also includes the question of how the client can handle potentially duplicate media data if

the cached- and the forwarded data happen to overlap. Such questions would likely require some further experimenting and testing to find a good solution.

### 3.4 Congestion Considerations

QUIC, just like many other modern transport protocols, contains congestion control mechanisms that regulate the rate at which data is sent to a client. This is primarily done to avoid the network of becoming congested but it also has the secondary effect of circumventing the problem of overwhelming the client. Simply forwarding all packets the relay receives from the server would cause the relay-client connection to not have its own congestion control anymore. Rather the rate at which the relay sends/forwards to the client would be determined by the server's congestion control algorithm, i.e. the network congestion between server and relay. Obviously this is not a desirable situation so our approach suggests the eBPF program at egress to have its own congestion control functionality.

#### 3.4.1 Client Congestion

Already hinted at in figure 3.4, it is shown that once a packet is registered there will also be a map update that will be triggered by the congestion controller. This map update will tell the eBPF egress program how much data it is allowed to send out. In figure 3.4 this is visualized exemplarily as "Bytes per Second Limit" but the idea is that both the function determining how limits and thresholds are calculated from the information of incoming packets as well as the actual handling within the egress program will be application specific and defined by the relay engineer. We experimented with approaches that use the QUIC internal measurements like the RTT, introduce new measurements like exponential-weighted moving averages, or even use an out-of-band connection where the relay expects direct feedback from the client. All these possibilities show that there is a lot of room for experimentation and optimization in this area. This however will not be explored further in this thesis and is left for future work.

#### 3.4.2 Packet Filtering and Dropping

Assuming that the network congestion state is known and communicated to the relay, one can use the priority-information within a packet (that is expected to be set by the server) to decide which packets should be forwarded and which ones should be dropped. One difficulty in this approach is that the dropping mechanism essentially works as an online- algorithm, meaning that it does not have full knowledge of the traffic especially not of future packets. This means that a case like the following could happen:

- The traffic contains packets within the priority range of 1 to 5 (5 being the highest priority).
- Given the current network congestion to the client the relay decides to drop all packets below priority 3 as well as 50% of the packets with priority 3.
- The remaining byte limit to be sent out is running low and a packet with priority 3 comes in which is sent since the relay already dropped a lot of previous priority 3 packets.
- The next packet turns out to be a priority 5 packet which would overshoot the byte limit if sent.

In this example one could use many different heuristics to handle the situation. One could always keep enough byte limit left that a high priority packet can always be sent. This, however, essentially just lowers the byte limit for all other packets while making it higher for high priority packets. Therefore an individual byte limit per priority could also be used right away. Also this approach might cause problems if a lot of high priority packets come in at once, e.g. in case of very “bursty” traffic. Another way that could be used to handle this uncertainty situation is to allow for temporary overflows of the byte limit. This would make the limit more of a soft limit that can be exceeded for a short time. This however is another heuristic that is highly dependent on the specific use case and traffic patterns which is why we did not implement it in our prototype. Overall we can say that implementation wise it is not hard to drop packets but the actual difficulty lies in finding a reasonable way of deciding which packets to drop.

## 3.5 Integration and Prototype

TODO

### 3.5.1 Compatibility

TODO

### 3.5.2 Source Code Repositories

For the development of the relay and the eBPF programs, we have come up with the following repositories:

- **Fast-Relay** [Pfe24b]: This is the main repository providing the eBPF program implementations as well as examples of server, relay and client implementations in Go.



- **Quic-Go Adaptation** [Pfe24a]: This repository is a fork of the QUIC library “quic-go” [See24] and provides a plain Go implementation of the QUIC protocol. For our thesis we needed to make some adaptations to the library to support some hook points for separate functions which should be specifically designed to handle the underlying eBPF setup with its eBPF-Map usage.
- **MoQ-Transport Adaptation** [PE24]: This repository is a fork of the “MoQ-Transport” [Int24a] protocol repository and provides some needed adaptations to our examples. One such adaptation is that the server needs to support a categorization of payloads into different priorities in order for the eBPF program to be able to deliberately drop packets in case of congestion. Getting these priorities could be as simple as differentiating only between I- and P-frames in a video stream or more complex based on the needs of the application and the wanted granularity of the congestion control.

## 4 Testing

## 4.1 Setups

TODO

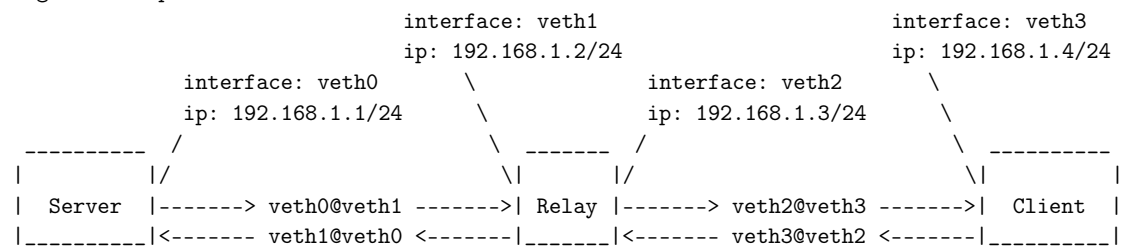
### 4.1.1 Namespace Environment for Local Testing and Development

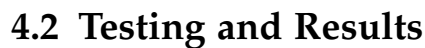
TODO

### 4.1.2 Physical Server Setup for Real-World Testing

TODO

Logical setup:





### 4.2.1 Delay Reduction of eBPF Forwarding

Another thing the figure shows is that the delay has a smaller variance due to the fact that the eBPF program path is somewhat similar for each packet whereas, in contrast, the userspace path can have buffers, queues, or similar that lead to a higher difference in processing time between packets. This effect however might be less observable in a real world scenario due to the ubiquitous network jitter which was influential in the used namespace environment. TODO

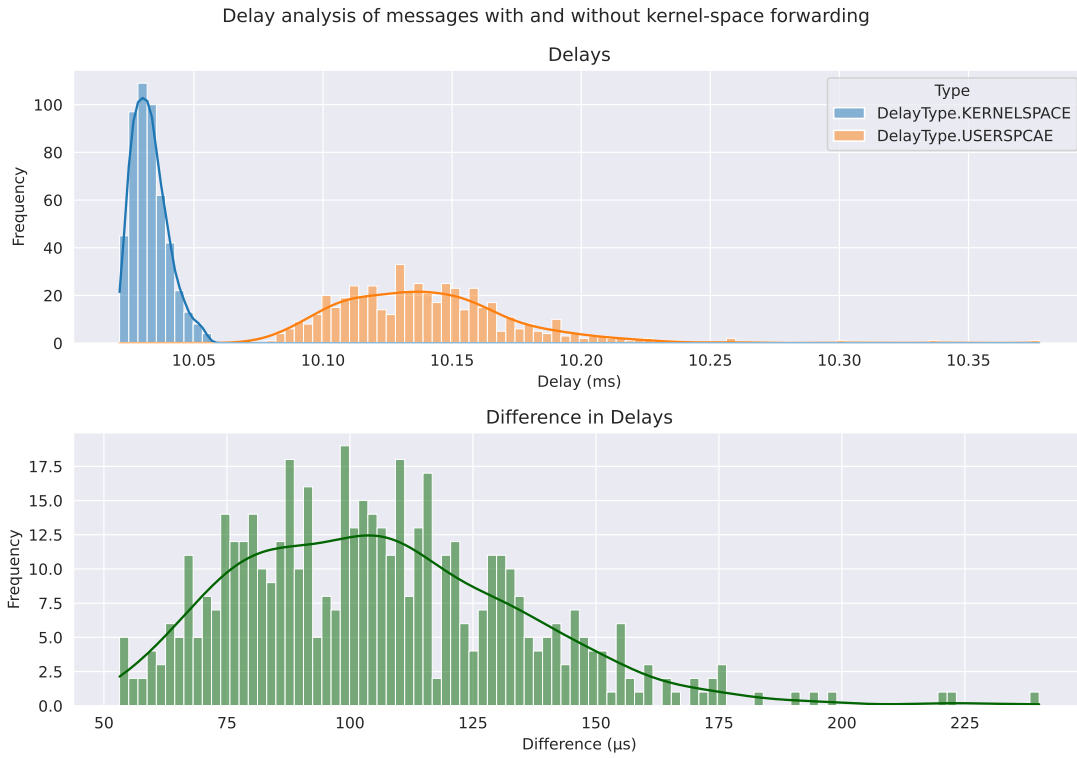


Figure 4.1: By avoiding userspace when processing the 1-RTT packets that contain the payload the delay of a single packet can be reduced. The longer-delay-operations are either handled directly in the eBPF program (e.g. the case for deciding where to redirect a packet to) or handled after the packet has already been sent out (e.g. the case for registering a packet such that the QUIC library knows about it).

## 4.2.2 CPU Utilization Comparison

TODO

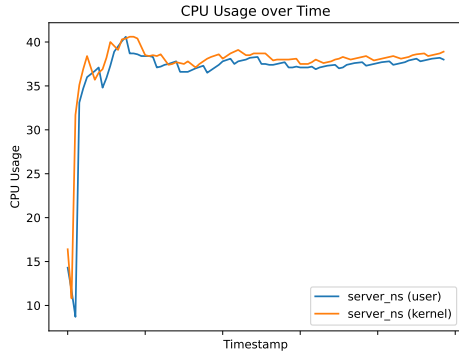


Figure 4.2: Server namespace CPU usage comparison.

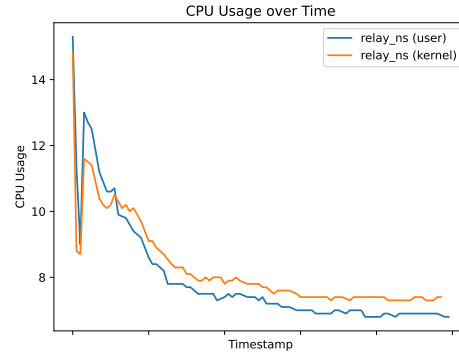


Figure 4.3: Relay namespace CPU usage comparison.

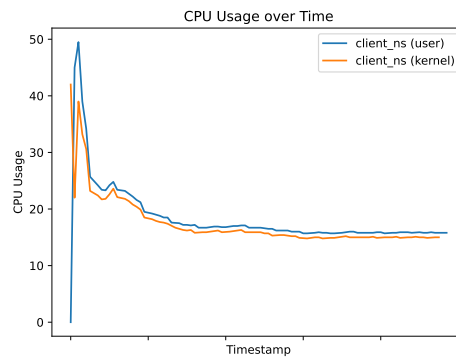


Figure 4.4: Client namespace CPU usage comparison.

TODO

Table 4.1: Table for CPU usage of relay Go processes

flat	flat%	sum%	cum	cum%	cause
440ms	28.39%	28.39%	440ms	28.39%	runtime/internal/syscall.Syscall6
140ms	9.03%	37.42%	140ms	9.03%	runtime.futex
120ms	7.74%	45.16%	120ms	7.74%	runtime.cgocall
40ms	2.58%	47.74%	40ms	2.58%	runtime.write1
30ms	1.94%	49.68%	60ms	3.87%	runtime.checkTimers
30ms	1.94%	51.61%	50ms	3.23%	runtime.mapaccess2
20ms	1.29%	52.90%	70ms	4.52%	.../cilium/ebpf/internal/unix.Syscall
20ms	1.29%	54.19%	20ms	1.29%	net.IPString
20ms	1.29%	55.48%	20ms	1.29%	runtime.casgstatus
20ms	1.29%	56.77%	20ms	1.29%	runtime.duffcopy

## 5 Future Work

### 5.1 Hardware Offload

This thesis heavily relies on the fact that the relay can access certain fields (e.g. packet numbers) of the packet, which are generally not accessible without prior decryption. In the current setup, this is made possible by turning off encryption altogether but to be of any use in a real-world scenario, the encryption of incoming and the decryption of outgoing packets would need to be pushed down below the lowest used BPF hook point in the stack. This means that a hardware offload of encryption and decryption similar to what is done for TCP/IP checksums would be necessary.

Once compatible SmartNIC offload implementations are available one can, besides en- and decryption, also offload the BPF program itself. This then would provide another way of accelerating performance.

Some previous work in this direction has already been done since at least 2019 [Yan+20] but for the purpose of this thesis we did not find any suitable open-source implementation that would allow incorporation into our fast-relay example implementation.

### 5.2 Compatibility Expansion

This thesis used the QUIC protocol together with media over QUIC (MoQ) to demonstrate how fast-relays that circumvent userspace by utilizing BPF programs can be designed. However, generally speaking the design of fast-relays is not limited to any of these protocols and could be expanded given modifications to necessary fields are possible (i.e. not prevented by encryption) and there is a way to encode the priority of a packet in the packet itself. The latter point could always be realized by using part of the payload which forces a deeper packet inspection within the BPF program but avoids the need to fit the priority into the header of an existing protocol.

## 6 Conclusion

TODO



# List of Figures

1.1	Packet path schematic regarding network stack . . . . .	2
2.1	Hook points within network stack . . . . .	7
2.2	Exemplary eBPF map definitions . . . . .	8
2.3	Rough MoQ relay architecture . . . . .	10
2.4	Non-caching relay traffic . . . . .	10
2.5	MoQ relay traffic . . . . .	10
2.6	Adaptive streaming schematic . . . . .	12
3.1	Types of eBPF programs at relay . . . . .	15
3.2	Video packet duplication . . . . .	19
3.3	Stream id translation schematic . . . . .	20
3.4	Packet registration schematic . . . . .	21
3.5	Packet retransmission schematic . . . . .	22
4.1	Delay analysis of eBPF approach . . . . .	29
4.2	Server CPU usage comparison . . . . .	30
4.3	Relay CPU usage comparison . . . . .	30
4.4	Client CPU usage comparison . . . . .	30

## List of Tables

2.1	Subset of eBPF map types . . . . .	9
3.1	Redirection helpers for packet buffer . . . . .	18
4.1	Table for CPU usage of relay Go processes . . . . .	31

# Bibliography

- [Fou] L. Foundation. *eBPF Verifier*. Accessed: July 13th, 2024. URL: <https://docs.kernel.org/bpf/verifier.html>.
- [Int24a] Internet-Engineering-Task-Force. *Media over QUIC Transport*. Accessed: July 19th, 2024. 2024. URL: <https://datatracker.ietf.org/doc/draft-ietf-moq-transport/>.
- [Int24b] Internet-Engineering-Task-Force. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Accessed: May 26th, 2024. 2024. URL: <https://datatracker.ietf.org/doc/html/rfc9000>.
- [JC20] M. Joras and Y. Chi. *How Facebook is bringing QUIC to billions*. Accessed: May 26th, 2024. Oct. 2020. URL: <https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/>.
- [KWF03] C. Krasic, J. Walpole, and W.-c. Feng. *Quality-Adaptive Media Streaming by Priority Drop*. June 2003. doi: 10.1145/776322.776341.
- [Lan+17] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. "The QUIC Transport Protocol: Design and Internet-Scale Deployment." In: *SIGCOMM '17: Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Aug. 2017), pp. 183–196. doi: 10.1145/3098822.3098842.
- [Lie18] E. Liebetrau. *How Google's QUIC Protocol Impacts Network Security and Reporting*. Accessed: May 26th, 2024. June 2018. URL: <https://www.fastvue.co/fastvue/blog/googles-quic-protocols-security-and-reporting-implications/>.
- [PE24] D. Pfeifer and M. Engelbart. *Media over QUIC Transport*. Accessed: May 26th, 2024. 2024. URL: <https://github.com/danielpfeifer02/priority-moqtransport.git>.
- [Pfe24a] D. Pfeifer. *A QUIC implementation in pure Go*. Accessed: May 26th, 2024. 2024. URL: <https://github.com/danielpfeifer02/quic-go-prio-packs.git>.
- [Pfe24b] D. Pfeifer. *Adaptive Media over QUIC Transport*. Accessed: May 26th, 2024. 2024. URL: [https://github.com/danielpfeifer02/Adaptive\\_MoQ.git](https://github.com/danielpfeifer02/Adaptive_MoQ.git).

- [PVV21] G. Pantuza, M. A. M. Vieira, and L. F. M. Vieira. *eQUIC Gateway: Maximizing QUIC Throughput using a Gateway Service based on eBPF + XDP*. 2021. DOI: 10.1109/ISCC53001.2021.9631262.
- [Rod24] L. Rodriguez. *The Future of the Internet is here: QUIC Protocol and HTTP/3*. Accessed: May 26th, 2024. Jan. 2024. URL: <https://medium.com/@luisrodri/the-future-of-the-internet-is-here-quic-protocol-and-http-3-d7061adf424f>.
- [See24] M. Seemann. *A QUIC implementation in pure Go*. Accessed: May 26th, 2024. 2024. URL: <https://github.com/quic-go/quic-go.git>.
- [Sta94] I. O. for Standardization. *ISO/IEC 7498-1:1994*. Accessed: July 3rd, 2024. 1994. URL: <https://www.iso.org/obp/ui/en/#iso:std:iso-iec:7498:-1:ed-1:v2:en>.
- [Tig] Tigera. *eBPF Explained: Use Cases, Concepts, and Architecture*. Accessed: July 19th, 2024. URL: <https://www.tigera.io/learn/guides/ebpf/>.
- [Tyu22] N. Tyunyayev. *Improving the performance of picoquic by bypassing the Linux Kernel with DPDK*. Master's Thesis. 2022.
- [W3T24] W3Techs.com. *Usage statistics of QUIC for websites*. Accessed: May 26th, 2024. May 2024. URL: <https://w3techs.com/technologies/details/ce-quic>.
- [Yan+20] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi. "Making QUIC Quicker With NIC Offload." In: *EPIQ '20: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Aug. 2020), pp. 21–27. DOI: 10.1145/3405796.3405827.