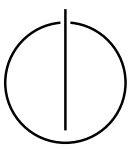
TUM SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

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

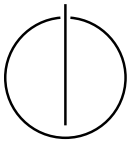
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

eBPF-Assisted Relays for Multimedia Streaming

eBPF-Unterstützung für Multimedia-Streaming-Netzknoten

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

Abstract

Multimedia streaming constitutes a significant portion of internet traffic, aiming to provide a seamless user experience with low latency and high quality, even amidst network congestion. Analyzing the packet path from media servers to clients reveals opportunities for latency reduction, particularly regarding network stack traversal within relays.

We propose leveraging eBPF to avoid most of the network stack traversal and directly forward packets from ingress to egress within the kernel. However, this introduces the challenge of maintaining userspace awareness of forwarded packets. To address this, we propose a delayed notification setup that informs the userspace application post-forwarding. We utilize eBPF maps to provide a low-latency communication channel between userspace and the eBPF program.

Additionally, we discuss necessary mechanisms for tasks like congestion control and retransmissions. We will also present a prototype demonstrating the basic functionality and showcasing latency improvements of our approach compared to traditional userspace processing.

Contents

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. Despite this delay not necessarily being a deal-breaker, designing networks that tighten the gap between cable and streaming is still worthwhile. With very optimized and fast networks already in place, we are at a point where providing faster information delivery is highly non-trivial. To do that, we have even gone as far as developing entirely new standards, such as QUIC. Those new standards aim to improve the shortcomings of some of the most fundamental protocols of the Internet, which have been around for more than 40 years, one of them being TCP.

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" [25]. 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 such case and explore the possibilities of avoiding or delaying specific processing steps of the ISO/OSI model to increase the overall speed of a network relay.

1.1 Research Question

As mentioned above, applying 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 [10]. The central question we will try to answer in this thesis will be:

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

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. They will handle basic relay capabilities, such as packet forwarding and congestion

control. Since the QUIC protocol is designed to handle a significant 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 can 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 using eBPF. Therefore, more specific sub-questions we try to answer are:

1. *How can we remove userspace packet-processing from the critical path?*
2. *How to handle packet en- and decryption?*
3. *What communication between userspace and the eBPF program is necessary to stay coherent?*
4. *How can our approach be generalized to support other protocols?*

1.2 Scope

This thesis will work in a server-relay-client environment. A relay is a special type of server with individual connections to the previous and the next hop in the network path. In our case the previous and next hop will already be the media server and the client, respectively. This is visualized in Figure 1.1.

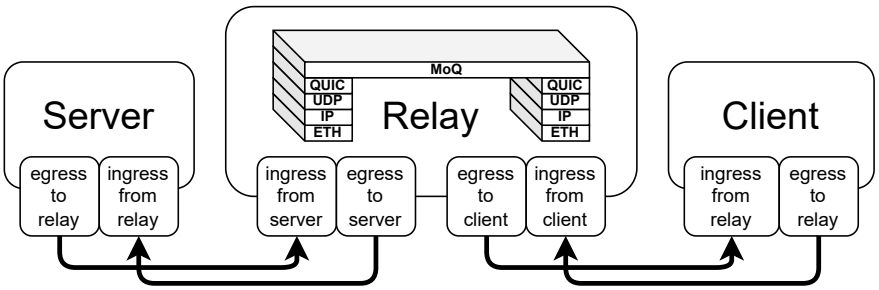


Figure 1.1: Relay setup.

The task of a relay is to pass on the traffic it receives from its predecessor to one or more successors. To achieve this, a packet goes all the way up the relay's application layer. We aim to avoid such an expensive traversal up the network stack.

The main improvement this thesis aims to achieve is shortening a packet's critical path from a media server to a client. This will be done by avoiding the immediate need for 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 is sent) by utilizing eBPF maps for storing any necessary (meta-) information. This communication between userspace and the eBPF program is required because relays in MoQ are an application layer concept. That means the QUIC connections from the relay to the server and from the relay to the client will be different, and the packets that have been eBPF-forwarded to egress directly will need changes in their header data 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 [12] and the MoQ transport protocol [10]. For the application layer, the quic-go library [24] 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 RFC 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 set up 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 relay implementation to confirm this approach's potential. These performance tests will look at the raw delay speedup and 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. Most 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 actual 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 one's 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 of 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 allowed 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 an 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 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 real-time streaming as well as 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 and our eBPF setup in section 3.2. Besides those two, we will look at more specific details and challenges in the subsequent sections. In chapter 4, we will provide a basic performance analysis of our setup to show current improvements and limitations. Finally, we will conclude with a summary and some ideas for future work in this field in chapter 5.

2 Background and Related Work

In this chapter, we will look at some technologies and concepts that will be important for the understanding of the following chapters. We will look at the QUIC protocol, which was our choice for the transport layer, eBPF, which we used for implementing the packet forwarding, Media over QUIC (MoQ), on top of which we built our example application, and Adaptive Bitrate Streaming, as well as Real-time Communication, which are fundamental concepts in the area of media streaming. Finally, we will mention some related work that tackles similar problems and provides interesting approaches.

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 now being its own trademark, is a transport layer protocol built on top of UDP. It aims 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 [17]. Back in 2018, QUIC was already the default protocol for the Google Chrome browser, which, at the time, made up 60% of the web browser market [18]. 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 [14]. As of August 2024, QUIC already made up 8.1% of all internet traffic with support from pretty much every major browser [28, 23]. On another note, Cloudflare-Radar has reported that at the time of writing this thesis, 30% of HTTP traffic is HTTP/3, and therefore using QUIC [4]. With big players like Google, Meta, and Microsoft putting emphasis on using QUIC to improve their

services, this number will likely 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 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. QUIC's 1-RTT handshake does not have this issue, while still being faster than e.g. the handshake of TCP/TLS, since it handles all setup tasks using only a single round trip. Another part where QUIC tries to optimize connection management is stream usage. Streams are designed to be lightweight and can be opened without the need for a handshake. This goes as far as one single packet being able to open a new stream, transfer stream data, and close 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, it is also possible to send data via unreliable datagrams [11]. This is possible since QUIC is based on UDP. It further improves the 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 [12], RFC-9221 [11], as well as some others, which are not important for our use case. 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. 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 with 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. Another reason we opted for QUIC is that it allows for easy packet-dropping. In order to do that, we just need to send each frame in a lightweight unidirectional stream and in case of a drop

of said frame, we can just close the corresponding stream. This would not be possible with TCP, due to its reliability-based nature.

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 predefined hook points in the network stack of the kernel and filter packets in a stateless manner. This provided more efficiency since the packets did not need to be copied into userspace anymore, but could be processed directly in the kernel. A need for better tracing capabilities of the Linux kernel then 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*, which allows for real-time inspection of running processes, memory- and CPU-usages, network-resources and more [26].

2.2.1 eBPF Hook Points

The Linux kernel offers several hook points to which eBPF programs can be attached. There are two prominent ones that we considered for our suggested setup. The first allows access to the *Traffic Control* (TC) subsystem, while the second allows access to 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. On the other hand, TC offers a more versatile way of packet processing since the used `sk_buff` buffer object containing the packet data provides access to metadata, which is unavailable when using XDP and its `xdp_buff` buffer object. What ultimately led us to choose TC over XDP was, however, the fact that XDP only allows ingress packet processing, while TC allows processing packets at 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 aim for.

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

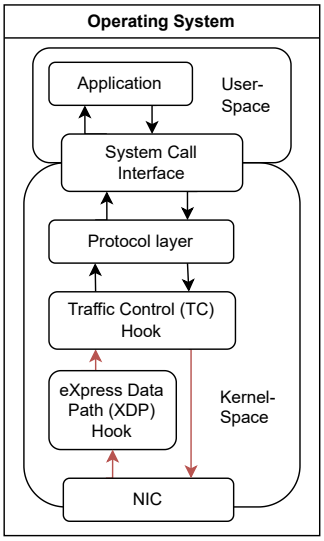


Figure 2.1: TC and XDP hook points in the Linux kernel. The red loop indicates the shortcut, which the fast-relay utilizes.

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 *eBPF verifier*, which inspects every eBPF program for its safety by simulating possible program paths, looking at the graph representation of the program and more [7]. This imposes some restrictions on the complexity of the programs that can be used within the kernel. For our early prototype implementation, this does not cause any issues, since we do not need to rely on very complex control structures. However, if one wants to extend the prototype to support a more complicated packet structure (e.g. composition of multiple frames per packet), the verifier might become a limiting factor.

2.2.3 eBPF Maps

One of the most important concepts in eBPF, which we 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 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.

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);         // Hash map
    __type(key, struct client_info_key_t);   // Specific client key
    __type(value, uint32_t);                 // 32 bit id
    __uint(max_entries, MAX_CLIENTS);        // Maximum number of clients
    __uint(pinning, LIBBPF_PIN_BY_NAME);     // Pin by name to the tc filesystem
} client_id SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);      // Ring buffer
    __uint(max_entries, MAX_PACKET_EVENTS); // Maximum number of packet events
    __uint(pinning, LIBBPF_PIN_BY_NAME);     // Pin by name to the tc filesystem
} packet_events SEC(".maps");
```

Listing 2.1: Examplary eBPF map definitions.

When we define an eBPF map, we can choose between different types and configure size, key type, value type, and how the map is stored. An example of two eBPF map definitions can be seen in Listing 2.1. It shows two different types of map definitions, a hash map, and a ring buffer, that are part of our fast-relay setup. Those and other relevant map types are listed in Table 2.1.

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 in the process of standardization by the IETF [9]. MoQ targets live-streaming applications like Twitch or YouTube Live 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 main system requirements with regard to media-data-transmission-protocols and -setups. One is low latency, while the other is high scalability [13]. Systems of the former include real-time collaboration tools like Zoom, Teams, or Meet. The latter are often needed by huge platforms like Twitch, YouTube, or Netflix, which aim to reach millions of users simultaneously. The one thing both have in common is that it turns out to be difficult to incorporate both low latency and high scalability into a system at the same time [13]. The MoQ protocol tries to solve this by providing a low-latency and highly scalable setup. To achieve this, it supports performance-enhancing approaches like relay caching or support for adaptive rate mechanisms.

2.3.2 Design of a MoQ Relay

The charter of the IETF working group [9] 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.

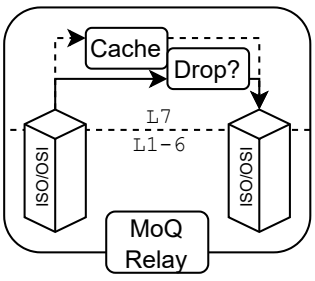


Figure 2.2: MoQ relay architecture.

Figure 2.2 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.

Considering the caching mechanism, the relay can cache media data to reduce internal traffic and improve user experience by lowering the latency of a request. Figure 2.3 and Figure 2.4 show example traffic for the two different approaches. Comparing the traditional non-caching relay and a caching MoQ-relay, we can see the latter lowers the traffic between server and relay in case the same data is requested by multiple clients. Cached data can also be sent out faster since the relay does not require any communication with the server to retrieve the data.

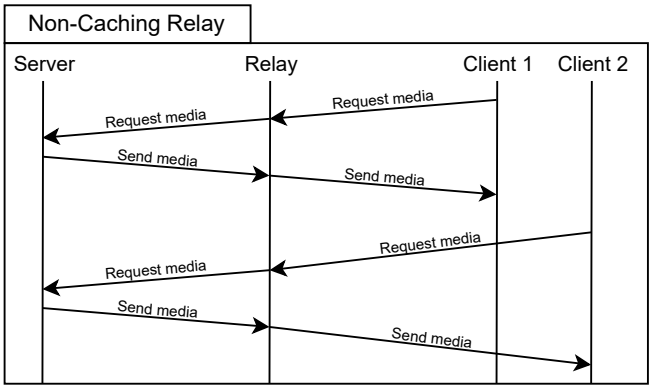


Figure 2.3: Multiple clients requesting data using a non-caching relay.

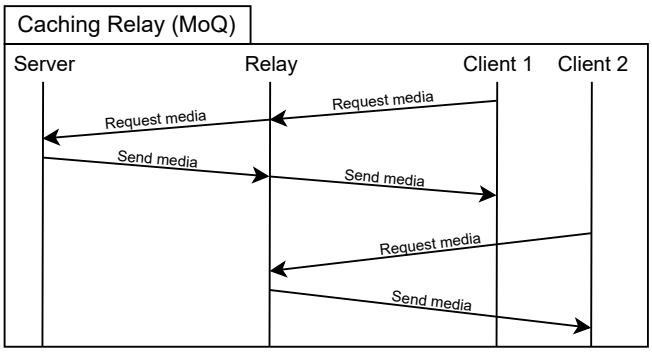


Figure 2.4: Multiple clients requesting data using a caching relay.

This caching mechanism fits quite naturally into our proposed eBPF setup since we will need to communicate packet data between kernel- and userspace 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 easily 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, we will go into detail on how the eBPF setup actually uses mechanisms like priority fields to achieve this.

2.3.3 moqtransport

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 [22]. 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, the 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 [10].

2.4 Real Time Communication and Adaptive Bitrate Streaming

Whenever we look at different types of streaming, it is important to consider the different requirements that come with them. We differentiate between a one-sided communication, where an increase in delay in order to gain a higher quality is acceptable, and a two-sided communication, where a low delay is crucial for a satisfactory user experience. The former allows for more complex setups like the one considered in subsection 2.4.2 while the latter requires a more direct approach which will be discussed in the following subsection.

2.4.1 Real Time Communication

When looking at streaming setups that consider real-time constraints, such as conferencing, where interhuman communication is involved, the objectives of the connection tend towards low latency. In order to provide natural-feeling human communication, the delay between the sender and the receiver has to be minimal. This type of communication between devices is often referred to as *Real-Time Communication* (RTC) and used in many applications, such as Zoom, Skype, or Discord.

Mechanisms and Ideas

The main goal of such systems is to minimize the delay between the sender and the receiver, while additionally allowing all parties to communicate with each other. A lot of design decisions are made with this in mind. One design decision for a large conferencing system like Zoom might consider the usage of peer-to-peer (P2P) connections versus a server-based approach. While P2P might allow for lower latencies

in case of a small number of participants, a server-based approach scales a lot better and can therefore provide a better resource utilization.

Implications on Streaming Setup

The need for low latency also has implications for the fundamental setup. An example of such an implication is the connection design between the real-time connection network (RTCN) and a client. Since retransmissions are expensive in terms of latency, a setup should strive to minimize the distance a packet has to travel through an unreliable network. This could be achieved by having multiple, strategically placed edge-servers. Those would internally use a connection, which is more reliable compared to the internet. Such a reliable connection would then be used for a large part of long-distance transmissions (e.g. intercontinental ones). This way, packet loss occurring during a client-to-edge-server connection would not have a big impact on the overall latency of a connection. A visualization of such an approach can be seen in Figure 2.5.

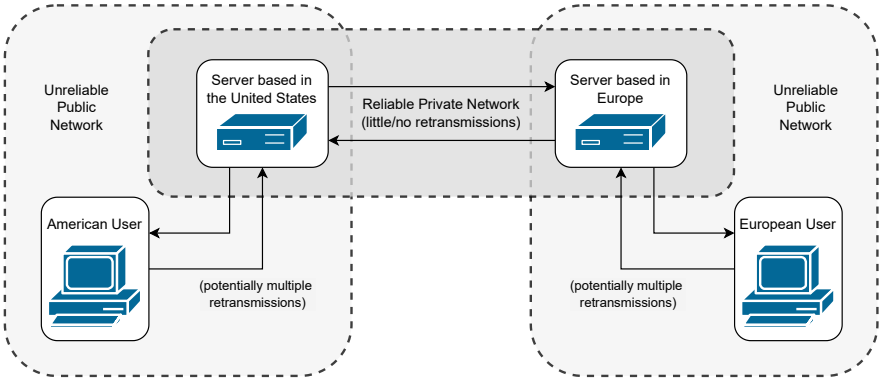


Figure 2.5: A setup with multiple edge servers worldwide can help minimize a connection's latency.

2.4.2 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 a client's connection state, such as available bandwidth, and adapting the rate at which a server sends data [5, 8]. Such a process is called *Adaptive Bitrate Streaming* (ABS) and is employed in many of today's streaming setups [1].

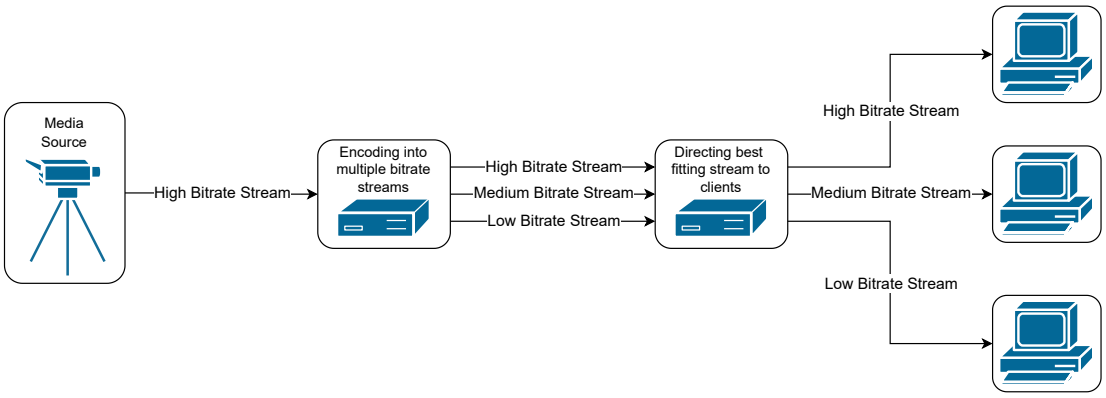


Figure 2.6: Behind the scenes, multiple streams with different resolutions exist to allow adapting to a user's bitrate.

Mechanisms and Ideas

ABS implements the simple idea of changing the amount of data sent to a client based on the client's connection quality. This means a client with a good connection gets a higher-quality stream than one with a bad connection. For this, the internal setup needs to be able to provide multiple streams with different resolutions and bitrates. An example setup can be seen in Figure 2.6 where an encoder creates streams with different resolutions. This allows an edge server, which manages the connections to the clients, to switch between those streams based on the connection state of a specific client. Twitch and YouTube Live are examples where, although more complex, similar setups are used to provide a better user experience. The premise of such a setup is that in case of a bad connection, it is still preferable to be able to watch a stream continuously, even if that means cutting down on quality. However, one must keep the differences between Video-on-Demand (VoD) and live streaming in mind when looking at such real-world examples. The latter might require more computation before a video is sent since the media has to be processed on the fly since it is created in real time. For VoD, on the other hand, the media already exists in its final form.

Implications on Streaming Setup

ABS forces a few changes to the way servers and relays are set up. First, as Figure 2.6 shows, there needs to be an encoding component that turns the single, high-quality stream received from the streaming source into multiple streams with different resolutions. In some cases, the relay also needs to keep track of the client's connection quality, e.g. using measurements like RTT, packet loss, et cetera, and decide which stream to

forward to the client. In other cases, this can also be done by the client itself. Since the connection quality can change dynamically, the one responsible for the measurements must continuously monitor the connection and potentially trigger a change in the used stream. The MoQ standard does not yet specify if the decision to adapt the bitrate lies with the publisher or the subscriber. Besides encoding and monitoring aspects, the system also has increased storage requirements since stream data will be multiplied.

2.5 Related Work

Our approach combines ideas from multiple different related publications. This section will give a brief overview of some related work that uses similar concepts such as packet handling using a BPF program or bypassing the Linux network stack to improve performance.

2.5.1 eQUIC Gateway

There have been previous publications on making QUIC more efficient by using BPF programs such as [19], 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 the current infrastructure.

2.5.2 Kernel Bypass

Another interesting approach that follows a similar idea of speeding up packet processing by avoiding the Linux network stack is [27]. The difference in this work is that DPDK is used to bypass the network stack to then process packets in userspace instead of directly using eBPF programs for processing, as we do. This, for example, offers more flexibility as the userspace program is not as limited (e.g. by the eBPF verifier) as the eBPF 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, thus increasing latency.

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. This is explored further in [16]. There the authors discuss a more tailorable congestion handling compared to video quality discretization as well as an improvement to, potentially randomized, frame dropping. This thesis, similar to [16], will not focus on how the priority for a specific

packet is determined but rather on how packets are handled in general. 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.

2.6 Summary

In this chapter, we have introduced some prerequisites for the upcoming one. We have looked at the QUIC transport protocol and the MoQ protocol, which is based on top of QUIC. We have also mentioned eBPF and presented two different concepts in the realm of streaming, namely ABS and RTC. In the next chapter, we will combine these concepts to create a relay design that is capable of traffic forwarding. We will do so by first mentioning necessary changes to the library implementing the QUIC standard, then we will explain our proposed eBPF setup and finally, we will go into detail on special considerations like userspace synchronization or congestion-related topics.

3 Fast-Relays

In this chapter, we will dive into the specifics of our proposed fast-relay setup. We will cover necessary library adaptations as well as setup specifics of the used eBPF programs like userspace synchronization and congestion considerations.

Starting off high-level, Figure 3.1 shows an overview of a basic server-relay-client setup with its respective networking layers. A conventional setup would have the packet travel into userspace at the relay, while our setup aims to shorten the critical path, as indicated by the red arrows.

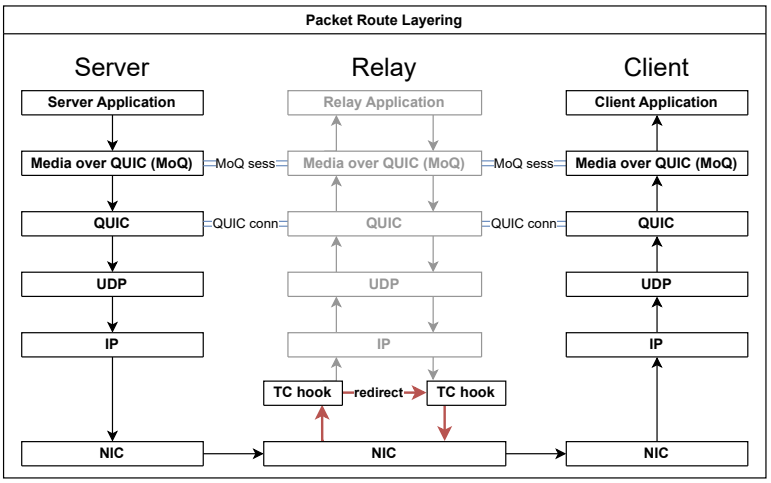


Figure 3.1: Packet path schematic including fast-relay-shortcut (eBPF packet-forwarding removing the need to go up to userspace).

3.1 QUIC Adaptions

As was already mentioned in the previous chapter, our setup requires some adaptations to the quic-go library. One initial change necessary was to turn off packet en- and decryption 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 QUIC header protection. For obvious reasons sending unencrypted packets is not something that would be wanted in a production environment, but for our exemplary setup, it is

unproblematic. An alternative would be to "push down" en- and decryption onto a smartNIC using a hardware offload, but at the time of writing, there was no such offload available for QUIC. Given that such a hardware offload is added in the future, the en- 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.

We added conditional function calls like the one depicted in Listing 3.1, at multiple locations within the quic-go library. 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.ConnectionIdUpdateBPFHandler != nil /* && potentially other
      conditions */ {
3     packet_setting.ConnectionIdUpdateBPFHandler(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 ConnectionIdUpdateBPFHandler func(id []byte, l uint8, conn QuicConnection) =
      nil
```

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

The function that the developer of the relay wishes to execute at specific points will be defined locally in the relay code and provided to the configuration of the quic-go library. An example illustrating how this could look is shown in Listing 3.3.

```go
/* Definition of the function within the local relay code */
func localUpdateConnectionId(id []byte, l uint8, conn
    packet_setting.QuicConnection) {
    /* handle the connection update by interacting with the eBPF program */
}

/* Providing the function to the quic-go library */
func main() {
    /* ... */
    packet_setting.ConnectionIdUpdateBPFHandler = localUpdateConnectionId
    /* ... */
}
```

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 would be the case if the library would access the eBPF maps directly.

3.1.2 Direct Changes to the Library

Besides the simple function-pointer style additions, we also had to make some direct changes to the library. These include simplifications of the packet structure to make the implementation of a prototype easier but also changes that are necessary for the whole approach to work. The necessary state changes mainly required internal state adjustments that would not be possible from outside the library because of missing access/visibility. The optional turn-off for the packet en- and decryption based on the value of the `CRYPTO_TURNED_OFF` flag also required some direct changes to the library, but these will not be mentioned further since they are temporary and not a focus of this section.

Simplifications of Packet Structure

We added some changes to the quic-go library to make a prototype implementation easier. The first one is that we fixed the sizes of some variable length fields like the connection-id or the stream-id. This was mainly to avoid the need to figure out the correct sizes within the eBPF program, as this would have resulted in approaches that would have to be very carefully turned into eBPF-compatible code due to verifier restrictions. Besides fixing the length of some fields, we also limit the number of frames

per packet to one. Normally a QUIC packet can contain multiple frames, especially stream-frames, but this would require a packet traversal within the eBPF program that, again, would have been harder to implement considering all verification constraints. Besides this additional complexity, for our use case, sending a single stream frame per packet is not changing a lot since the payload of one media packet will be split into multiple packets due to its size. This means a single stream frame is likely already using up a whole packet, and multiple frames per packet would not be happening often.

Stream Priorities

Our approach relies on the fact that every packet has a priority value assigned to it. In our case this priority value is stored within the connection-id. Based on this, the first change, which is not only simplifying our prototype implementation but is actually needed for our approach to work, is the addition of opening a stream with a specific priority. Our assumptions define that the server is the one marking the packets with the correct priority, which in our case is realized by sending them over a specific stream. Every stream is bound to a specific priority value, and our code changes will make sure that the connection-id that is used for any packet sent on this stream will contain the correct priority value. Figure 3.2 visualizes the setup as well as our rudimentary approach of saving the priority value as the first byte of the connection-id.

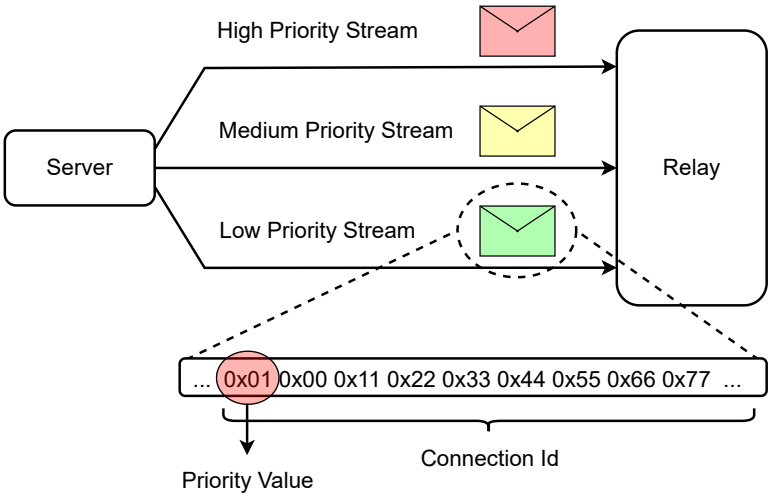


Figure 3.2: The server opens streams with specific priorities, which are then used to send packets with the corresponding priority.

This approach of having the priority value saved within the connection-id caused the need for another change in the library. Due to the periodic change/retirement of

connection-ids, we need to make sure that at any point in time, there exists at least one connection-id for each priority within the connection. Otherwise, we might be unable to mark a packet with the correct priority. This led us to introduce some additional logic to the code that is executing the retirement of connection-ids. There, we will ensure that before a connection-id is retired, either one with the same priority value already exists or is newly created. This solves the problem of not being able to mark a packet with the correct priority.

Retransmission

Another direct change that is needed is that of a specific `OnLost` function for packets that have been forwarded by the eBPF program. Since the relay state is not necessarily aligned with the server state, and the relay does not know about the stream a packet was sent on, it is not possible to reuse the plain `OnLost` function of the quic-go library. This is because the plain `OnLost` function would look up the stream corresponding to the provided stream-id and retransmit the packet on there, but the stream for the given id might not exist within the relay stream-pool. Instead, our new function needs to look up the stream-id of the lost packet and potentially open a new stream with the same stream-id, ensuring that the client correctly receives the retransmission. Besides the stream-id, the created stream-object should have the same remaining stream-state as the original stream (e.g. offset-information). When a retransmission happens, the relay also needs to tell the underlying eBPF program that a packet is part of a retransmission and that, even if a stream was actually created by the relay, the egress program should treat it like it was created by the server. That last part is necessary for the stream-id translation part of the egress program, which is explained further in subsubsection 3.2.1, to work correctly. This functionality of informing the eBPF program about retransmissions, however, is again realized by a function-pointer style addition and thus designed by the relay developer.

Visible Endpoint for Packet Registration

The last direct change that we added was the introduction of an additional function `RegisterBPFPacket` on a quic-go connection object that allows the relay to register a packet. Since the packet registration requires access to the internal state of the connection, this also needed to be done as an actual change to the library. Now the relay can just read the necessary information of a packet that needs to be registered from the eBPF maps and then pass it on to this function, which will handle the registration. This also provides a good separation between the Go code that handles eBPF communication and the actual QUIC connection handling.

3.2 eBPF Setup

This section will discuss which eBPF programs are needed to achieve the desired relay functionality within the Linux kernel, as well as their individual responsibilities.

3.2.1 Different eBPF Programs

In order to allow the relay to forward packets independently of the userspace, we need to equip the relay with three eBPF programs, as seen in Figure 3.3. Those three programs are:

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

Their responsibilities then are:

- handling the initial registration of new clients and storing their information, such as MAC addresses, in an eBPF 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 packet 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.

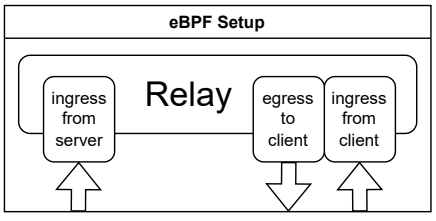


Figure 3.3: The relay has to be equipped with three eBPF programs.

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

Client Ingress

This ingress eBPF program initially does some simple packet inspection on every incoming packet, looking at whether 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 one 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, since we use QUIC, the program will check for QUIC long header packets that set up an initial connection. It will then save the transmitted state information such as connection-id, stream-related states, et cetera, in an eBPF map together with information that is not directly known by the userspace, such as the client's MAC address. Saving data like the MAC address directly once the connection is set up allows us 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, multiply the packets accordingly. The counter indicating the number 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 minuscule 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. Figure 3.4 shows the high-level packet duplication and forwarding process for an example with three clients who want to receive the video data.

In this ingress program, we need to consider a few things to ensure the correctness of our approach. These are:

1. The program can **only** forward packets that contain video data and must not forward any other packets that contain e.g. control data.

   This is fairly easy to achieve by doing some header inspection of the QUIC header, which contains the packet type. Also, since the payload is generally sent using short-headers, there is no need to consider long-header packets.

2. The program should pass an unaltered copy of each packet up to userspace to allow the QUIC library to gain knowledge of the packet and handle any state changes accordingly.

   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.

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.

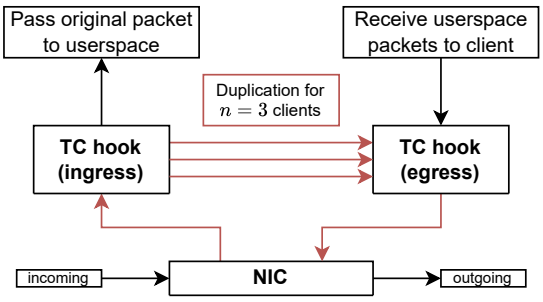


Figure 3.4: Duplication and forwarding of video packets to egress directly from ingress.

Duplicating and forwarding incoming packets that were identified as containing 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 with the index that is provided as the second argument. The last argument allows for the specification of 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 [6, 2]. Table 3.1 shows them together with a brief description of their behavior.

Helper	Description
`bpf_clone_redirect`	Clones and redirects a packet to the interface associated with the provided index.
`bpf_redirect`	Redirects a packet to the interface associated with the provided index. The packet is not cloned (no underlying call of `skb_clone()`) so it is slightly more efficient. The packet is also not redirected immediately but only after the function finishes.
`bpf_redirect_peer`	Similar to `bpf_redirect`, 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.
`bpf_redirect_neigh`	Again similar to `bpf_redirect` but allows to redirect a packet to another net device. This helper also fills 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. There are more than that, but those are the ones we identified as possibly useful initially.

Based on the descriptions of the redirection helper functions mentioned in Table 3.1 it becomes clear that `bpf_clone_redirect` is the one suitable for our use case. This is because we need the cloning aspect, as 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 everything, from state-management to packet forwarding, 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. The second and third requirements are directly caused by the interleaving of packets, while the other points are general requirements for the egress program.

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 the userspace nor the server knows 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 an eBPF program, changing it right before sending it out is the best way of keeping them coherent.

3. In case the packet (in addition to being QUIC for a client connection) contains stream frames, we need to do a similar translation of the stream-id. The reasons and methods for this are the same as for the packet-number translation. Figure 3.5 shows a flow diagram of the stream-id translation process. Important steps include checking if the stream-id translation already exists, as well as figuring out where a stream was created. The former is necessary in case a payload is split into multiple packets, and the latter is necessary since the relay essentially has a fan in of two (relay-originated packets and server-originated packets). This fan-in might cause a situation where both the relay and the server open a stream, which initially has the same id (since the server and relay states are not synchronized). In such a case, the two streams must not be mapped to the same outgoing stream-id. This is also why a retransmission coming from the relay needs to be treated as if it came from the server. For that, the eBPF program will be notified by the userspace regarding which packets are such retransmissions.

4. Again, if the packet is QUIC and considering the right connection, the program needs to read the packet's priority and decide via a map-lookup if the connection state allows sending packets with the given priority.

5. If the packet is a redirected media stream data packet, the program must determine which client it should be sent to. 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.) using the saved id and overwrite the respective fields in the buffer before sending it out.

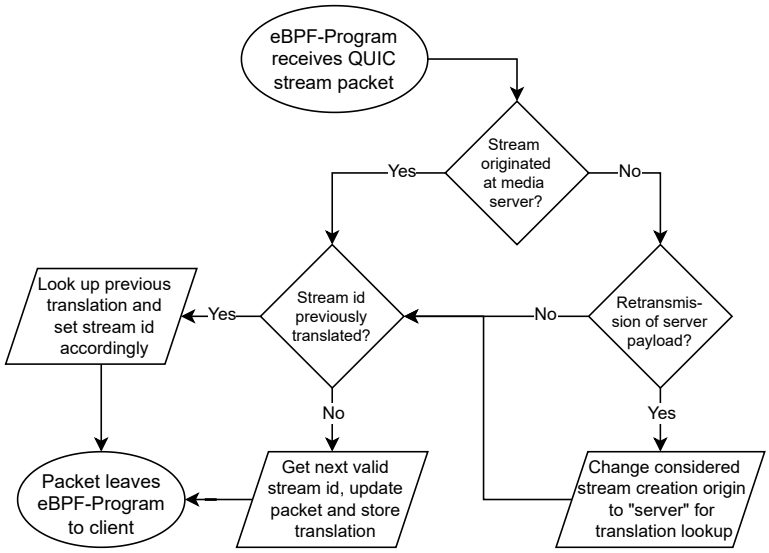


Figure 3.5: Flow diagram of stream-id translation process.

The aforementioned packet-number translation means that the packets sent by the userspace will likely have a different packet-number i.e. not the one chosen by the QUIC library. This might lead to inconsistencies again when receiving ACKs, but can be avoided by remembering the translation in a map, as well as by only storing packet objects in the Go library that have the correct state. This technique of "storing" a packet that the userspace 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. 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 into more detail on how the packet registration works. Later, we will also look at the implications of this on retransmissions. Even though the stream-id translation works very similarly 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, which is running in userspace, usable, we need to inform the QUIC library about the forwarded packets. This again happens via eBPF 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 3.6 visualizes the setup for this process.

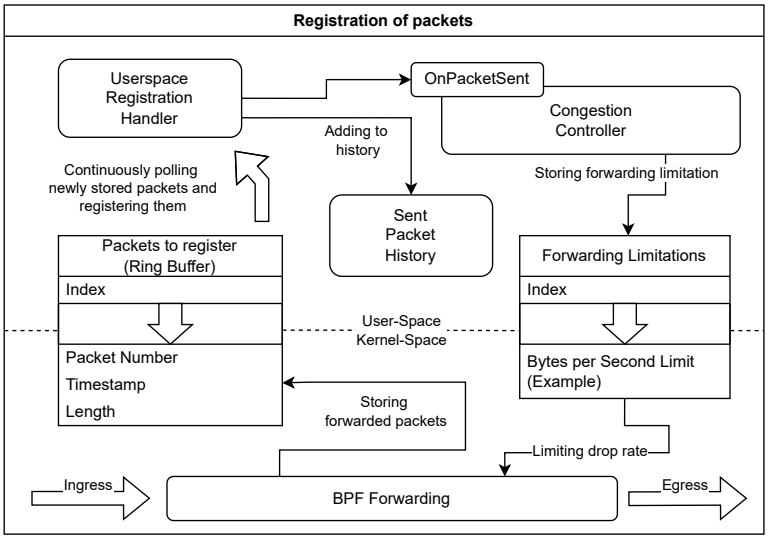


Figure 3.6: Internal setup for registering forwarded packets as well as incorporating forwarding limitations for the BPF program.

3.2.3 Retransmissions of Forwarded Packets

The fact that the relay only knows about the packets after a separate registration call causes another issue when it comes to retransmissions. Retransmissions happen at a stream level, which means that the relay needs to know about the stream in the first place if it wants to retransmit a packet. Given that we actually never create the streams that contain the frames of the payload data at the relay, this is a problem. Our proposed system handles this by manually creating a stream with the correct id, the first time a packet goes missing for that stream. This can be imagined as a “just-in-time” stream creation before the retransmission is sent.

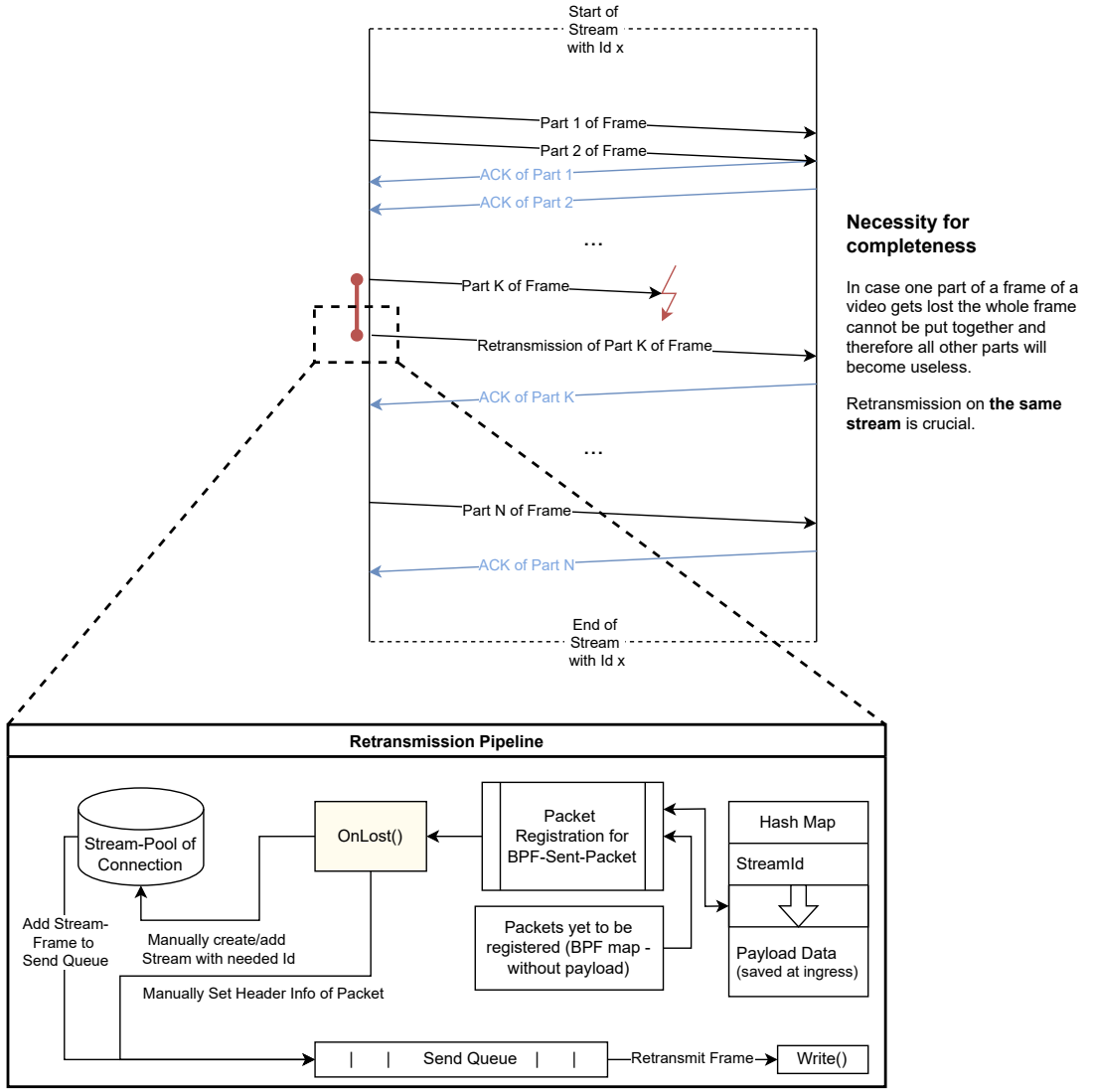


Figure 3.7: Internal setup for retransmitting packets forwarded from server connection.

It is not necessary to create the stream before that because our approach is only working with unidirectional streams so the relay does not get any data back from the client. The only case where it matters that the relay state "knows" about the stream is when a packet gets lost, and the `OnLost` function of the QUIC library is called. This is also the latest point in time when the relay can create the stream with the needed stream-id. Figure 3.7 visualizes the setup for retransmitting a packet that got lost on its way to the client.

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. 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, et cetera, 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 additional communication since the server is expected to use QUIC's unidirectional streams for sending the media data. That means the relay does not need to know about the stream details except if it has to trigger a retransmission.

Since a client can also unsubscribe from a certain media stream, the relay needs to support this as well. This is done by simply decrementing the *number-of-clients* counter and making sure the client ids stay in a usable state (i.e. the relay is not duplicating packets for unsubscribed clients). Our prototype implementation does not consider this yet, because our proof of concept and performance tests do not require it.

3.3.2 Relay Caching

Caching of data within a relay, which is required by the MoQ standard, is something we essentially get for free since we are 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. This means the application is able to receive any data from the server as if this was a normal connection and store, e.g. the last `n` second(s) worth of data in a cache. Then, once a new connection is established, the relay could, in parallel

to incrementing the kernel counter representing the number of clients, also send out cached data so that the client receives it as early as possible.

One aspect that we still left open is the point in time when 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 further experimenting and testing to find a good solution.

3.4 Congestion Considerations

QUIC, like many other modern transport protocols, contains congestion control mechanisms regulating the rate at which data is sent to a client. This is done to avoid the network becoming congested. Simply forwarding all packets the relay receives from the server would cause the relay-client connection to no longer have its own congestion control. 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 3.6, 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 3.6, this is exemplified as "Bytes per Second Limit". The idea is that both the function determining how limits and thresholds are calculated from the information on 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 QUIC internal measurements like 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 showed us 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 situation like the following could happen:

- The traffic contains packets within the priority range 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 and 50% of 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 of the byte limit available so that a high-priority packet can be sent without delay. 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 uncertain 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 period of time. However, this is another heuristic highly dependent on the specific use case and traffic patterns, so 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

To show the feasibility of our approach, we have built a prototype that implements the suggested setup. It is capable of streaming an example video as well as mock-payload data from a server to a client via a relay. This setup will be open source and available on GitHub. Some of the mentioned functionalities, like retransmission, caching, and adaptive bitrate streaming, have not yet been fully implemented but should not require major additions since the infrastructure already considers them. The absence of these functionalities will not have a drastic impact on further testing, however, since the retransmission and the adaptive bitrate streaming will not be needed in our lab-environment, and the caching is not expected to have a big influence either.

3.5.1 Compatibility

The application layer of the prototype is written in Go, but this is not a requirement itself. The eBPF program expects QUIC, but this could also be changed to support other underlying transport protocols. In order to keep the needed changes minimal, we would, however, suggest that a protocol with similar properties to QUIC is chosen. TCP, for example, might not be the best choice as its reliability assurances would not allow for a simple priority drop.

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** [21]: 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** [20]: This repository is a fork of the QUIC library "quic-go" [24] and provides a plain Go implementation of the QUIC protocol. For our thesis, we needed to make some adaptations to the library, such as adding support for hook points for separate functions, which are specifically designed to handle the underlying eBPF setup and its eBPF map usage.
- **MoQ-Transport Adaptation** [22]: This repository is a fork of a repository implementing the MoQ-Transport protocol [10]. It provides some needed adaptations. 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.

3.6 Summary

In this chapter, we have presented our approach to designing a relay based on eBPF forwarding. We examined the details of the necessary changes to the underlying QUIC library and explored the specific requirements and challenges involved in designing such a system. All this led us to a detailed explanation, which provides the basis for a prototype implementation we developed. In the upcoming chapter, we will use this prototype to evaluate the performance of our relay design compared to a more traditional system.

4 Testing

In this chapter, we will be discussing the way we tested our prototype implementation and the results we obtained from these tests. We will mainly consider basic metrics such as delay reduction and CPU utilization to ensure the fundamental impact of our prototype is analyzed. Further testing might be especially interesting after the integration of a working congestion handling mechanism into our prototype.

4.1 Setups

We test the performance of our prototype in a lab-like environment that allows us to maintain a stable network state. We do this by using Linux network namespaces where each namespace represents a different participant in the communication, i.e. the server, the relay, and the client. A schematic overview of the setup is shown in Figure 4.1.

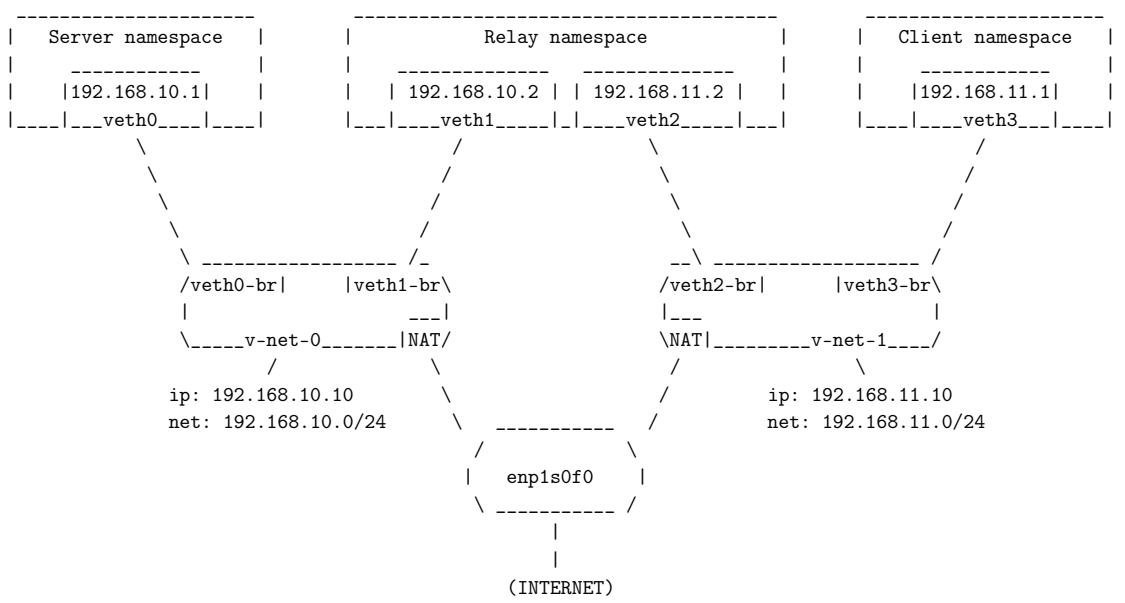


Figure 4.1: Local setup including bridges and namespaces.

4.1.1 Local Environment for Testing and Development

The local environment we use for testing and developing allows us to set up the network in a way that fits the current needs of the prototype. One example of such needs would be that we need a slight delay between sending a packet and receiving its ACK since the registration of a sent packet takes place after a packet is sent out. In a real-world scenario, this would obviously be the case if the distance covered were large enough.

In our setup, we introduce a queueing discipline (qdisc) to an interface of the bridge that connects the relay and the client. This qdisc introduces a delay of 5ms to each packet. Using such qdiscs allows us to simulate a network environment that is more realistic, e.g. by introducing delays, like we did, or by defining thresholds for packet loss. In our setup, however, we only added a delay. The fact that the delay of a bridge can be set to a fixed value allows us to better analyze and understand the impact of different setups, for example on packet jitter. Additionally, we can be confident that any delays or improvements are not attributable to external network conditions but rather to the immediate forwarding introduced by our approach. It also ensures that variations in delay are primarily due to the modifications we made to the relay setup.

4.1.2 Physical Server Setup for Real-World Testing

Our setup for testing is solely based on local namespace environments. This suffices since we only look at delay reduction and CPU utilization, both of which are only local metrics regarding the relay. We left it open for future work to implement a real-world setup that could potentially involve multiple relays and observe their combined improvement in a more sophisticated network setup. This goes hand in hand with a working congestion-control extension to the eBPF and QUIC setup, which reacts to a real-world network environment with ubiquitous changes in congestion.

4.2 Testing and Results

When testing the performance of our prototype, we will focus on showing that the eBPF forwarding is capable of reducing the delay of packets. Delay in our case is measured again using eBPF programs that save the timestamps of a packet leaving and entering the server and client namespace, respectively. Due to the controlled nature of the local environment, we know that the difference between the two timestamps will be mainly due to processing at the relay. Additionally, since we are on a single physical machine, it is clear that the timestamp-differences will be accurate because no clock synchronization is involved. In a real-world setup, different, poorly synchronized clocks could lead to inaccuracies.

Besides delay analysis, we will also show that our approach does not require more CPU resources than the plain userspace forwarding. For that, we will look at profiling data from programs like *pprof* as well as the Linux *process status* (`ps`) command, which

can be used to show the CPU usage of a process. Additionally, we will look at the system calls that are used by both approaches.

4.2.1 Delay Reduction of eBPF Forwarding

When considering the impact of eBPF-Forwarding on the delays of packets, Figure 4.2 visualizes the timestamp and delay data that was the result of a rudimentary test of a single-connection scenario that was run both with and without direct eBPF forwarding. We can see that the delay of a single packet is decreased by around 100 µs when compared to the userspace forwarding. The userspace relay setup for this measurement only considers the simplest case of a single client connection and a direct "passing-through" of the packets without much additional computation. More complex setups might have the relay consider tasks like (de-) multiplexing, encoding changes, error correction, or similar. Given that such complexity can become arbitrarily large, this delay improvement can become even bigger. Important to mention is that en- and decryption should not have an influence on the delay reduction since both setups use the same en- and decryption methods. Any change that might be observable in an extended prototype, which uses hardware offloading, is likely caused by a change in processing time that the offloading itself introduces. This means that in case one uses a fully offloading prototype for testing, the same offloading should be used for the traditional setup it is compared to.

Another result Figure 4.2 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 equivalent structures that can 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 general network jitter, which might outweigh the reduction in jitter that our setup caused. The measurements depicted in Figure 4.2 were taken by sending mock data over the network, with each packet being processed once in userspace and once directly forwarded in the kernel (i.e. each packet is duplicated). Since all packets are considered independent of each other, it suffices to use the data of one test run, given that enough packets were sent, for a statistically significant result.

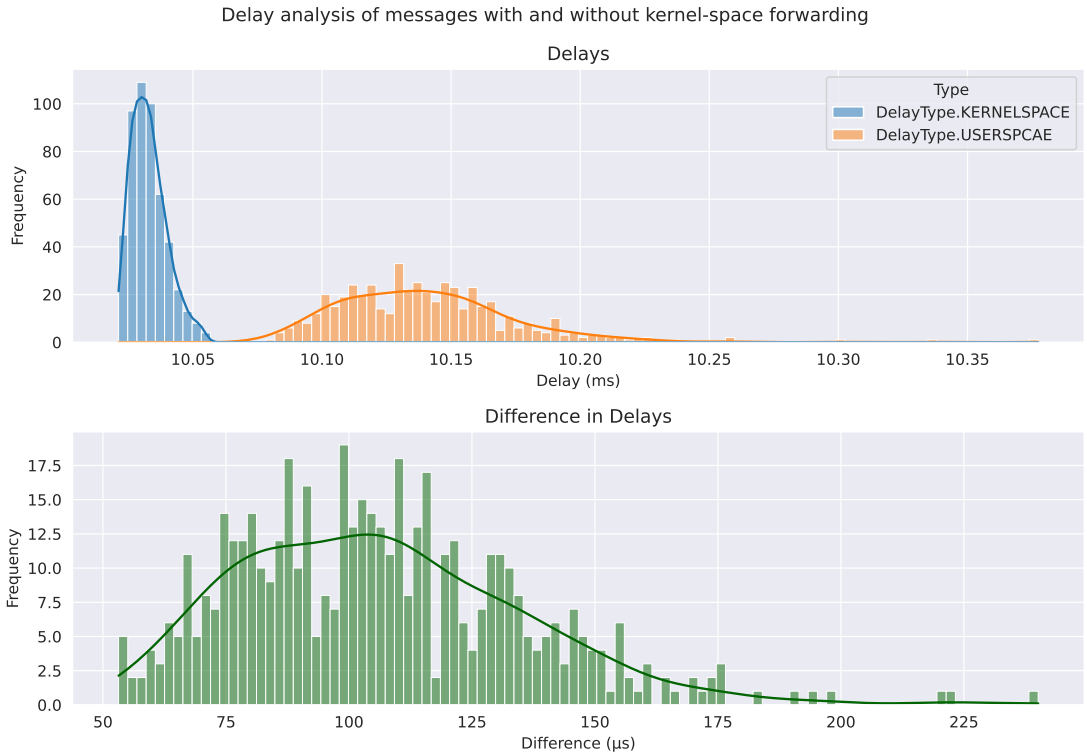


Figure 4.2: The blue and orange parts of the plot show the delay of a kernel- and userspace-forwarded packet, respectively. Delay is measured as time passed between leaving the server eBPF program and entering the client eBPF program. The green part shows the raw difference between the delays of the same packet, which was sent via both kernel- and userspace-forwarding.

4.2.2 CPU Utilization Comparison

Besides showing that our approach reduces packet delay, we also measured that the CPU usage is not negatively impacted by our streaming system. Figure 4.3, 4.4 and 4.5 show the CPU usage of the server, relay, and client processes, respectively, both with and without eBPF forwarding. It is observable that none of the utilizations significantly differs between the two setups. We created these CPU measurements by accumulating the CPU usage of all processes that are related to the respective parts of the setup.

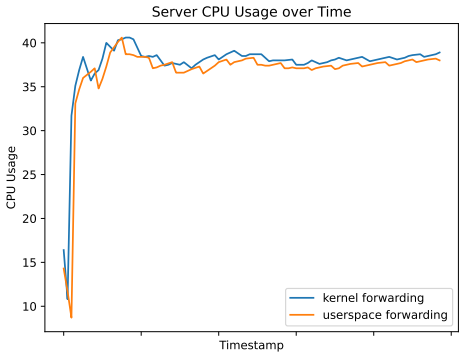


Figure 4.3: Accumulated server CPU usage comparison.

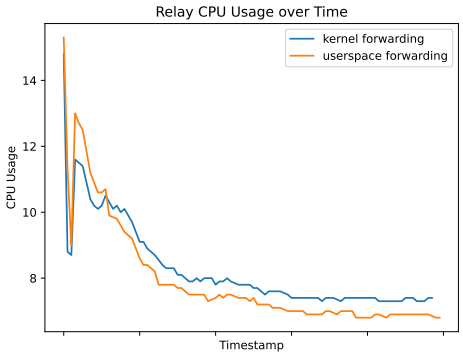


Figure 4.4: Accumulated relay CPU usage comparison.

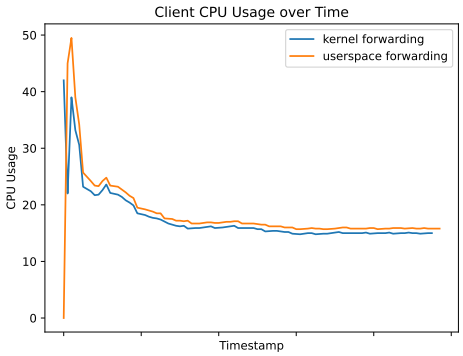


Figure 4.5: Accumulated client CPU usage comparison.

Syscall Usage

Appendix A and B show the syscall usage of a 30-second video streaming example considering a setup with and without eBPF forwarding, respectively. We trace this using *strace* while considering the main process as well as all child processes it creates and interacts with. When comparing the two tables, it is evident that our approach uses significantly fewer system calls. This is partially due to a reduced need for userspace synchronization, decreasing the number of `futex` syscalls. Other syscalls like `epoll_pwait` or `nanosleep` are also used significantly less.

The concrete impact of our setup on system calls can be seen at the bottom of Appendix A and B where total syscall counts are shown. The setup using eBPF forwarding uses 225674 syscalls during our example transmission, while the traditional setup uses 296132. This means our setup uses roughly 24% fewer syscalls than the traditional one. Regarding `futex` calls, our setup uses a little more than 34% fewer than the traditional setup, with 21666 calls instead of 32940. Even bigger percentage differences can be seen

for `nanosleep` calls with a reduction of approximately 42% (14293 instead of 24716) and `epoll_pwait` calls with an improvement of around 67% (11289 instead of 34149).

Another observation we made was that the eBPF setup uses fewer `bpf` syscalls than we would have assumed, given the amount of traffic that occurred during the test. This might be caused by optimizations happening in the eBPF library used by our Go application. Such optimizations might include reducing the number of `bpf` syscalls by combining multiple map reads into one syscall. We did, however, not investigate how this underlying library works and how our eBPF communication might be optimized.

4.3 Summary

We have presented the results of comparing our newly developed relay design to a traditional relay system without eBPF usage. We were able to show that the idea of delaying any non-essential application-layer processing until after the packet was forwarded does provide a performance benefit, while utilizing a similar amount of CPU resources. The fact that our system works also shows that eBPF is capable of providing basic relay functionality despite possible limitations due to its verifier. In the following chapter, we will conclude our work and provide an outlook on future work that can be done in this area.

5 Conclusion

Closing this thesis, in this chapter, we will summarize and conclude our findings and give a brief outlook on what one could do to advance the work we have done. Many of the initial questions we posed in the introduction have been answered but some areas, even though briefly touched upon, will require from further research.

5.1 Conclusion

In this thesis, we have shown how one can improve the performance of a relay in a media streaming scenario by using eBPF technology. We explained needed high-level concepts and provided insights into implementation details of a prototype showing the feasibility of our approach. Concluding, we can say that leveraging the high-performance capabilities of eBPF programs and forfeiting some universality of a relay design can lead to faster and more deterministic packet processing with a similar CPU load. We thereby answered the initial research questions regarding the possibility of a performance improvement and the avoidance of userspace processing. Our prototype showed the needed communication between the userspace and the eBPF program to keep coherency and, by doing so, answered another research subquestion regarding the necessary communication. Lastly, throughout our work, we also hinted at how to expand our setup to support other protocols and standards, covering the last of our initial questions regarding generalization.

5.2 Future Work

Here we will mention some parts that still require additional work to be done. The areas that will be mentioned will provide a good starting point in case one wants to build upon or extend our work.

5.2.1 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. However, to be of any use in a real-world scenario, the decryption of incoming and the encryption

of outgoing packets would need to be handled below the lowest used BPF hook point in the stack.

Future work could focus on developing a hardware offload setup similar to those already available for TCP/IP checksum offloading [3]. We looked for possible smartNICs and libraries that would allow us to offload en- and decryption. However, at the time of writing, we could not come up with any viable solution that works with our current setup. We believe one could possibly even put the eBPF program itself on a smartNIC as another approach suggests [15]. The cited work, however, only considers eBPF/XDP offloading, so further research would be required to see if this could be adapted to eBPF/TC programs as well.

Despite the current lack of options, previous work in this direction has already been done since at least 2019 [29], which makes us hopeful that integrating smartNIC support into our prototype can be achieved in the near future.

5.2.2 Compatibility Expansion

This thesis used the QUIC protocol together with MoQ to demonstrate how fast-relays, which circumvent userspace by utilizing eBPF 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), the protocol provides a possibility to drop packets (i.e. not like TCP, but similar to QUIC), and there is a way to accommodate the priority of a packet in the packet itself. The final point could always be realized by using part of the payload which forces a deeper packet inspection within the eBPF program but avoids the need to fit the priority into the header of an existing protocol.

Expanding our approach to other protocols could be interesting for many areas outside of media streaming, such as gaming, telesurgery, or financial services, where a delay-decrease of microseconds would be particularly beneficial.

5.2.3 Prototype Completion

As mentioned already during the setup description, our prototype does leave out some parts of the proposed design. Despite them not being essential for our proof of concept, both in functionality and performance, they are still important for a final product that is intended to be used in a real-world scenario.

Future work could focus on finalizing the prototype by researching and implementing best practices for congestion handling, retransmission management, and other related aspects. A fully functional prototype could then be tested in a real network environment to see how common network conditions, that were not considered within our namespace environment, affect the performance of the system.

Abbreviations

ABS	Adaptive Bitrate Streaming
ACK	Acknowledgment
ARP	Address Resolution Protocol
BPF	Berkeley Packet Filter
CDN	Content Delivery Network
CPU	Central Processing Unit
DPDK	Data Plane Development Kit
eBPF	extended Berkeley Packet Filter
ETH	Ethernet
HLS	HTTP Live Streaming
DASH	Dynamic Adaptive Streaming over HTTP
HTTP	Hypertext Transfer Protocol
HTTP/3	Hypertext Transfer Protocol version 3
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISO	International Organization for Standardization
OSI	Open Systems Interconnection

MoQ	Media over QUIC
MoQT	Media over QUIC Transport
NIC	Network Interface Card
P2P	Peer-to-Peer
QDISC	Queueing Discipline
RFC	Request for Comments
RTC	Real-Time Communication
RTCN	Real-Time Communication Network
RTP	Real-Time Transport Protocol
RTT	Round-Trip Time
TC	Traffic Control
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VoD	Video-on-Demand
XDP	eXpress Data Path

List of Figures

List of Tables

Bibliography

[1] N. T. Blog. *Optimizing the Netflix Streaming Experience with Data Science*. Accessed: August 9th, 2024. June 2014. URL: https://netflixtechblog.com/optimizing-the-netflix-streaming-experience-with-data-science-725f04c3e834.

[2] A. Chiao. *Differentiate three types of eBPF redirects (2022)*. Accessed: August 9th, 2024. July 2022. URL: http://arthurchiao.art/blog/differentiate-bpf-redirects/.

[3] S.-M. Chung, C.-Y. Li, H.-H. Lee, J.-H. Li, Y.-C. Tsai, and C.-C. Chen. "Design and implementation of the high speed TCP/IP Offload Engine." In: *2007 International Symposium on Communications and Information Technologies*. 2007, pp. 574–579. DOI: 10.1109/ISCIT.2007.4392084.

[4] Cloudflare. *Cloudflare Radar: Adoption and Usage*. Accessed: July 30th, 2024. July 2024. URL: https://radar.cloudflare.com/adoption-and-usage.

[5] Cloudflare. *What is Adaptive Bitrate Streaming?* Accessed: August 9th, 2024. URL: https://www.cloudflare.com/learning/video/what-is-adaptive-bitrate-streaming/.

[6] T. L. Foundation. *bpf-helpers(7) - Linux manual page*. Accessed: August 9th, 2024. URL: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[7] T. L. Foundation. *eBPF Verifier*. Accessed: July 13th, 2024. URL: https://docs.kernel.org/bpf/verifier.html.

[8] ImageKit. *Adaptive Bitrate Streaming*. Accessed: August 9th, 2024. URL: https://docs.imagekit.io/features/video-transformation/adaptive-bitrate-streaming.

[9] Internet-Engineering-Task-Force. *Media over QUIC (moq)*. Accessed: July 18th, 2024. 2023. URL: https://datatracker.ietf.org/wg/moq/about/.

[10] Internet-Engineering-Task-Force. *Media over QUIC Transport*. Accessed: July 19th, 2024. 2024. URL: https://datatracker.ietf.org/doc/draft-ietf-moq-transport/.

[11] Internet-Engineering-Task-Force. *Media over QUIC Transport*. Accessed: July 30th, 2024. 2024. URL: https://datatracker.ietf.org/doc/html/rfc9221.

[12] 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.

[13] Internet-Engineering-Task-Force. *What's the deal with Media Over QUIC?* Accessed: July 18th, 2024. Jan. 2024. URL: https://www.ietf.org/blog/moq-overview/.

[14] 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/.

[15] J. Kicinski and N. Viljoen. "eBPF Hardware Offload to SmartNICs: cls bpf and XDP." In: *Proceedings of netdev* 1 (2016).

[16] C. Krasic, J. Walpole, and W.-c. Feng. *Quality-Adaptive Media Streaming by Priority Drop*. June 2003. DOI: 10.1145/776322.776341.

[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.

[18] 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/.

[19] 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.

[20] D. Pfeifer. *A QUIC implementation in pure Go*. Accessed: May 26th, 2024. 2024. URL: https://github.com/danielpfeifer02/quic-go-prio-packs.git.

[21] D. Pfeifer. *eBPF Assisted Fast Relays*. Accessed: May 26th, 2024. 2024. URL: https://github.com/danielpfeifer02/ebpf-fast-relays.git.

[22] D. Pfeifer and M. Engelbart. *Media over QUIC Transport*. Accessed: May 26th, 2024. 2024. URL: https://github.com/danielpfeifer02/priority-moqtransport.git.

[23] 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.

[24] M. Seemann. *A QUIC implementation in pure Go*. Accessed: May 26th, 2024. 2024. URL: https://github.com/quic-go/quic-go.git.

[25] 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.

[26] Tigera. *eBPF Explained: Use Cases, Concepts, and Architecture*. Accessed: July 19th, 2024. URL: https://www.tigera.io/learn/guides/ebpf/.

[27] N. Tyunyayev. *Improving the performance of picoquic by bypassing the Linux Kernel with DPDK*. Master's Thesis. 2022.

[28] W3Techs.com. *Usage statistics of QUIC for websites*. Accessed: May 26th, 2024. May 2024. URL: https://w3techs.com/technologies/details/ce-quic.

[29] 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.

Syscalls with eBPF Forwarding

% time	seconds	usecs/call	calls	errors	syscall
59,95	17,114866	789	21666	1663	futex
11,06	3,156126	52602	60		wait4
7,58	2,163202	151	14293	18	nanosleep
5,66	1,616939	143	11289	6	epoll_pwait
3,53	1,006913	33	30236	107	read
3,28	0,937777	29	32194		fcntl
1,56	0,444417	22	19860	1005	newfstatat
1,32	0,376735	33	11354	253	openat
1,02	0,289889	25	11417		close
0,90	0,256689	46	5580	5239	epoll_ctl
0,81	0,230937	4528	51	5	waitid
0,37	0,104574	21	4898	1812	recvmmsg
0,36	0,103789	52	1990		sched_yield
0,32	0,090941	2	32329		lseek
0,28	0,081038	413	196		clone
0,25	0,072563	62	1160		getpid
0,25	0,072532	64	1133		sendmsg
0,18	0,051164	37	1381	40	rt_sigreturn
0,18	0,050788	276	184		copy_file_range
0,17	0,049578	13	3697		getrandom
0,15	0,043913	39	1114		tgkill
0,12	0,035065	56	622		fstat
0,12	0,033083	79	416		getdents64
0,11	0,031193	14	2178		mmap
0,06	0,018254	13	1314	22	bpf
0,05	0,015350	17	875		pread64
0,05	0,012890	1171	11		restart_syscall
0,04	0,011800	15	782		rt_sigprocmask
0,04	0,011269	54	206	12	unlinkat
0,03	0,009583	7	1262		write
0,03	0,009384	32	285		madvise
0,03	0,007470	3	2481	2456	readlink
...	...	...	...	...	...
100,00	28,548177	126	225674	12923	total

Syscalls without eBPF Forwarding

% time	seconds	usecs/call	calls	errors	syscall
64,71	29,089191	883	32940	3320	futex
9,53	4,282788	101971	42		wait4
7,21	3,242993	94	34149	18	epoll_pwait
6,64	2,982646	120	24716	35	nanosleep
2,70	1,212353	39	30601	49	read
2,01	0,901806	28	31863		fcntl
0,97	0,437349	22	19794	1005	newfstatat
0,89	0,401939	33	11862		sendmsg
0,71	0,319470	18	17115		getrandom
0,68	0,307659	27	11274	253	openat
0,54	0,243391	21	11215		close
0,49	0,219378	40	5414	5221	epoll_ctl
0,46	0,206849	6894	30	2	waitid
0,38	0,172377	22	7719	2739	recvmmsg
0,36	0,162222	1763	92		pipe2
0,29	0,131692	68	1912		getpid
0,25	0,112894	40	2798		sched_yield
0,20	0,090007	2	32309		lseek
0,17	0,075008	39	1884		tgkill
0,13	0,058566	26	2237	200	rt_sigreturn
0,10	0,045475	247	184		copy_file_range
0,09	0,040321	268	150		clone
0,08	0,034742	83	415		getdents64
0,06	0,028940	46	622		fstat
0,06	0,028247	15	1791		mmap
0,06	0,026298	15	1751		write
0,03	0,013730	39	349		madvise
0,03	0,013598	755	18	2	restart_syscall
0,03	0,012830	15	831		pread64
0,02	0,010814	52	206	12	unlinkat
0,02	0,008929	14	612		rt_sigprocmask
0,02	0,007546	3	2481	2456	readlink
...	...	...	...	...	...
100,00	44,950890	151	296132	15575	total