SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Congestion Control and Scheduling for Application Flows with Varying Delay Sensitivities on Shared QUIC Connections

Jasper Karl Ottomar Rühl

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Congestion Control and Scheduling for Application Flows with Varying Delay Sensitivities on Shared QUIC Connections

Staukontrolle und Scheduling für Anwendungen mit unterschiedlichen Echtzeit Anforderungen über geteilte QUIC Verbindungen

Author:	Jasper Karl Ottomar Rühl
Supervisor:	Prof. Jörg Ott
Advisor:	Mathis Engelbart
Submission Date:	16.9.2024

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

Munich, 16.9.2024 Jasper Karl Ottomar Rühl

Acknowledgments

I want to thank my advisor Mathis Engelbart for patiently supporting me with his supervison and inspiring advices.

I want to thank my supervisor Professor Ott for giving me the opportunity to write this thesis at his chair.

I want to thank my parents for enabling my privilege to be a student following my passion at the TU Munich.

I want to thank all the people I met throughout my educational career at the TU Munich that brightened my day and made my study years to a great, defining phase of my life.

Abstract

Over the past decade, the prevalence of media streaming has seen a significant rise, marking a notable shift in how we consume and interact with digital content. This ranges from videoconferencing in both personal and professional settings to more unidirectional applications such as live streaming or video-on-demand (VOD) services for entertainment. Unlike VOD streaming, the other purposes all seek to operate with low latency, be it to enable a smooth and natural conversation among the conference participants or simply have the viewer of a live stream be up to date. A shared requirement is thus a consistent connection, guaranteeing low latency. While streaming real-time data, users might like to transfer a file to their peers, using the internet to exchange information more traditionally. Current implementations struggle to balance the transfer of data with different properties regarding volume and latency requirements. Our proposed approach, built upon the quic protocol, aims to address these challenges by enabling applications to utilize a multiplexed connection. This allows for the prioritization of single streams for real-time communication, thereby enhancing the overall streaming experience.

We define metrics to determine the quality of a real-time data connection and construct mechanisms to measure them and implement an algorithm that controls the volume of non-prioritized data dependent on these. Finally, we evaluate our contribution to a video streaming experiment.

Contents

1 Introduction

The rapid growth of real-time media streaming, fueled by advancements in high-speed networks and the increasing demand for multimedia applications, has posed significant challenges for maintaining consistent quality of service (quality of service). From live video broadcasts to interactive gaming and virtual reality environments, these applications rely heavily on the real-time transmission of data, where latency, jitter, and packet loss can profoundly impact user experience. Ensuring these services maintain performance, especially under fluctuating network conditions, requires effective congestion control mechanisms that dynamically adapt to network variability.

The complexity of the internet, with its dynamic and heterogeneous nature, exacerbates the challenges of congestion control for real-time media. Different network paths, varying bandwidth availability levels, and competing traffic streams all contribute to unpredictable network performance. This makes it crucial for congestion control algorithms to balance maintaining high throughput and minimizing delay and packet loss, ensuring that media streams remain smooth and responsive under diverse conditions.

While existing congestion control solutions primarily focus on optimizing the real-time delivery of media streams, modern network applications increasingly demand the simultaneous transmission of real-time communications (such as video or voice calls) and non-time-sensitive data (such as file transfers). This dual requirement presents new challenges, balancing the conflicting needs of two distinct traffic types: real-time communication, which prioritizes low latency and consistency, and bulk data transfer, which seeks to maximize throughput without stringent delay constraints.

To address this gap, this thesis proposes a novel congestion control mechanism that enables the simultaneous transmission of real-time communication and data transfer over the same network path on the transport layer. The proposed solution dynamically adjusts bandwidth allocation and prioritizes traffic in response to network conditions, ensuring that real-time communication experiences minimal latency without severely compromising data transfer throughput.

This contribution is particularly relevant for applications such as video conferencing systems, online collaborative platforms, and live-streaming services, where users often exchange media while simultaneously sharing files, documents, or other forms of data. By developing a congestion control strategy that balances the needs of both traffic types, this work aims to enhance the overall quality of service for end users in environments where network resources are limited or fluctuating.

By focusing on methods that adapt to network conditions in real-time and optimize

media delivery, this work seeks to improve the quality of a connection under constrained network environments.

In the following chapters, we will first briefly overview the underlying concepts, techniques, and prior work relevant to congestion control for real-time media streaming. Then, we present the high-level design of our proposed mechanism, elaborating on the ideas and reasonings behind our design choices. After that, we delve into the technical details of our implementation, covering the specific algorithm. In the evaluation chapter, we will evaluate the performance of our solution by comparing it to quic-go, the QUIC implementation in Golang on which we based our approach, using a video streaming experiment. Finally, we present an outlook on possible improvements in the future work section.

2 Background

This chapter gives a small synopsis of essential concepts and the preceding work on which this thesis builds.

2.1 TCP

Transmission Control Protocol (TCP) is one of the core protocols of the Internet protocol suite, playing a crucial role in ensuring reliable data transmission across networks. It operates at the transport layer of the TCP/IP model and is designed to provide connection-oriented communication[Edd22].

One of TCP's primary functions is to guarantee that data packets are delivered sequentially and without loss. This is achieved through flow control, error detection, and congestion control. Flow control prevents the sender from overwhelming the receiver with too much data at once, using a sliding window mechanism to manage the amount of data sent. Error detection ensures that corrupted or lost packets are retransmitted, using acknowledgments and timeouts to monitor the delivery status.

While TCP ensures reliable transmission, its inherent overhead (due to error checking and retransmissions) may not be ideal for applications requiring low latency, such as real-time communications, where protocols like User Datagram Protocol (UDP) are preferred. TCP remains the dominant protocol for web browsing, file transfers, and email applications, where data integrity and reliability are paramount.

2.2 UDP

User Datagram Protocol (UDP) is a core communication protocol in the Internet protocol suite, designed for applications that require fast, connectionless data transmission with minimal overhead[Pos80]. Operating at the transport layer of the TCP/IP model, UDP is often contrasted with TCP due to its focus on speed rather than reliability. Unlike TCP, UDP does not establish a connection before data is sent nor guarantees the delivery or ordering of data packets.

UDP is considered connectionless and stateless, meaning each packet (a datagram) is sent independently without acknowledgment or retransmission. This makes UDP an ideal choice for applications where low latency is more critical than perfect reliability, such as real-time streaming, online gaming, Voice over IP, and DNS (Domain Name

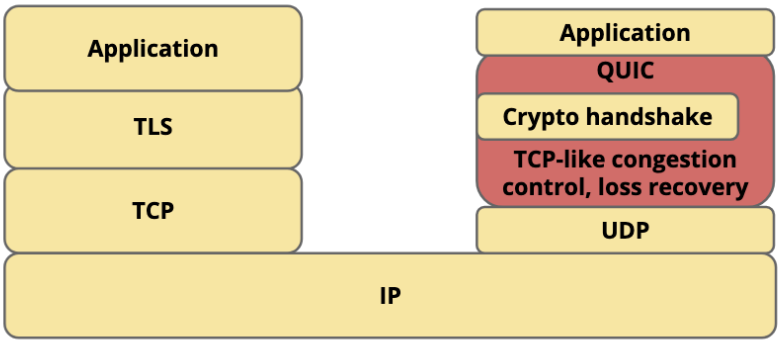

Figure 2.1: Picture illustrating the relationship of QUIC, UDP and TCP
source: datatracker.ietf.org/meeting/98/materials/slides-98-edu-sessf-quic-tutorial-00.pdf

System) queries. In these scenarios, slight packet loss or out-of-order delivery may be acceptable, or the application can handle retransmissions or corrections if needed.

2.3 QUIC

QUIC is a transport protocol designed to enhance the efficiency and security of Internet communications. Developed by Google, QUIC was first introduced in 2012 to overcome the limitations inherent in TCP. Unlike TCP, which is connection-oriented and relies on a multiple roundtrip handshake for establishing a connection, QUIC utilizes UDP as its underlying transport mechanism.

Initially deployed to address latency and performance issues in web services, QUIC has undergone significant development and standardization. Google's implementation of QUIC demonstrated improvements in connection establishment times and data transfer efficiency, which led the Internet Engineering Task Force (IETF) to take an interest in standardizing the protocol. In 2018, the IETF began formalizing QUIC through the QUIC Working Group. The standardization process led to the release of RFC 9000, which outlines the formal specifications for QUIC[IT21].

QUIC integrates several innovative features that distinguish it from traditional transport protocols:

- Reduced Latency: QUIC significantly reduces latency by streamlining the connection establishment process. Combining the handshake and encryption setup into a single operation, QUIC minimizes the time required to establish a secure connection to a single round trip.
- Stream Multiplexing: Multiplexed data streams are supported within a single connection, allowing multiple requests and responses to be handled concurrently

without interference, which improves data transmission efficiency and reduces the overhead associated with multiple connections.

- Built-in Encryption: Unlike TCP, QUIC incorporates encryption, namely the TLS 1.3 (Transport Layer Security) protocol, directly into the protocol. This ensures that all data transmitted over a QUIC connection is encrypted by default.

Major technology companies, including Google, Facebook, and Cloudflare, have adopted QUIC, demonstrating its practical benefits in real-world applications. Further, QUIC forms the foundation for HTTP3. Figure 2.2 depicts the usage of QUIC over the past year, stressing its wide adoption.

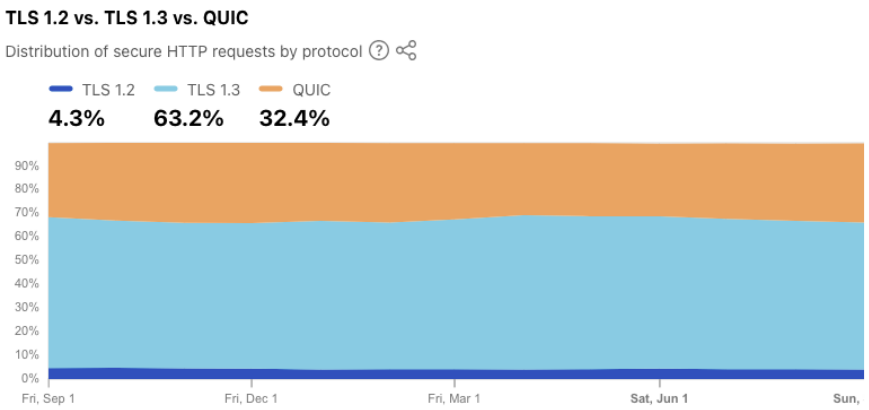

Figure 2.2: Usage diagram over the past year of QUIC, TLS1.2 and TLS1.3 per `radar.cloudflare.com/adoption-and-usage`

2.3.1 Stream Prioritization in QUIC

In late 2023, Fatima et al. presented a priority-based stream scheduler implemented in quic-go[Fer+23]. It enables the prioritization of single streams and claims to reduce delays under unreliable conditions by up to 36%. The approach was tested using a drone, with its control information serving the prioritized data stream. A Weighted Fair Queuing (WFQ) and an absolute strategy are implemented. Note that this algorithm addresses the order of messages being sent but not the volume at which they are.

2.4 RTP

The Real-Time Transport Protocol (RTP) is designed to deliver audio and video over IP networks in real-time. RTP provides end-to-end transport services suitable for

applications transmitting live media, such as voice-over IP, video conferencing, and streaming media[Sch+03].

RTP is typically paired with the Real-Time Control Protocol (RTCP), which monitors transmission statistics, network quality, and synchronization between media streams. Together, these protocols ensure the effective delivery and playback of real-time data, making them critical for maintaining the quality of service (quality of service) in multimedia applications.

2.5 WebRTC

Web Real-Time Communication is an open-source standard enabling peer-to-peer connections between browsers. It allows data transfer and video streaming inside webpages without installing additional plugins. Initially open-sourced by Google, it was finally standardized in 2021 by the W3C and IETF[Wor21]. In WebRTC, data can be transmitted either by media streams or data channels.

WebRTC is widely known for its ability to transmit real-time audio and video streams between peers. This is achieved through RTCPeerConnection and associated protocols like RTP[PWO21].

WebRTC Data Channels use the Stream Control Transmission Protocol (SCTP) to transport arbitrary data between peers. SCTP is the underlying protocol that ensures reliable, ordered, and error-checked data delivery. It also supports partial reliability, allowing specific data to be delivered without strict reliability constraints, which can be useful for time-sensitive data where occasional loss is acceptable[Ste07].

2.6 HTTP

The Hyper Text Transfer Protocol (HTTP) is an application layer protocol and arguably the foundation of our modern World Wide Web. The initial specification was published in 1996[NFB96].

2.6.1 Stream Prioritization in HTTP/2

Stream prioritization is a crucial feature of HTTP/2 that optimizes the delivery of resources by allowing clients to indicate the relative importance of various streams within a single connection[TB22]. This mechanism helps ensure critical resources, such as visible content on a webpage, are delivered more quickly. In contrast, less important resources, like background images or non-essential scripts, are deprioritized. Stream prioritization improves overall performance and user experience, particularly in scenarios with limited bandwidth or high network latency.

HTTP/2 introduces the concept of multiplexing, where multiple streams (requests and responses) are sent concurrently over a single TCP connection. To manage these

streams effectively, HTTP/2 allows clients to assign a priority to each stream using two key mechanisms.

Stream Dependency

Each stream can be assigned a dependency on another stream, forming a tree of streams. This dependency tree defines how streams are related in terms of their priority. A stream that depends on another stream is considered of lower priority, meaning that the dependent stream will not be fully processed until the stream it depends on is sufficiently handled.

Stream Weights

In addition to dependencies, each stream can be assigned a weight between 1 and 256. This weight determines how much available bandwidth should be allocated to a particular stream relative to others with the same parent stream.

By combining dependencies and weights, HTTP/2 enables fine-grained control over how resources are delivered. For example, a client can specify that the main HTML document (critical for rendering the page) has the highest priority, while images and stylesheets are given lower priority. Among these lower-priority resources, the client might still allocate more weight to CSS files over images, ensuring the page's layout is rendered before the decorative content.

While this feature is powerful, it is also quite complex to configure and implement. Even if you were to perfectly set the dependencies and weights, many applications would still need to correctly handle the given information. For that reason, the mechanism was simplified in HTTP/3.

2.6.2 Stream Prioritization in HTTP/3

HTTP/3, the latest version of the Hypertext Transfer Protocol, builds upon the foundation laid by HTTP/2 but operates over a different transport protocol, namely QUIC[Bis22]. While HTTP/3 retains many of the key features of HTTP/2, including multiplexing and stream prioritization, the underlying differences in transport protocols introduce changes in how stream prioritization is implemented and managed. Building upon QUIC, each stream in HTTP/3 is independent, meaning that loss or delay in one stream does not affect the others.

Priority Signaling

HTTP/3 inherits the concept of stream prioritization from HTTP/2 but uses a simplified mechanism for indicating priorities. Instead of the dependency tree model of HTTP/2, HTTP/3 utilizes a simpler priority model based on explicit priority signaling, namely

the Extensible Prioritization Scheme (EPS). This priority signaling uses HTTP headers (Priority header), where each stream can be assigned a priority value. This value indicates how important the stream is relative to others. For streams of equal priority, one can specify if those are meant to be served simultaneously (Round Robin style) or consecutively.

Adoption

Recently, Herbots et al. [Her+24] measured how HTTP/3's mechanism is supported. They found considerable differences in how content providers implement and adhere to ESP, often ignoring parts of it.

2.7 Congestion Control

Congestion control is a critical aspect of network management that ensures efficient utilization of network resources and maintains performance quality in high traffic volumes. Congestion occurs when the demand for network resources exceeds the available capacity, leading to packet loss, increased latency, and reduced throughput. Congestion control algorithms are designed to manage and mitigate these issues by dynamically adjusting the data transmission rate based on current network conditions.

The challenges a Congestion Controller needs to address are to ensure fairness among multiple connections, keep latency low, and maintain high adaptability to different and changing network environments.

Many congestion control algorithms have been developed, each with its approach to managing network congestion. Those can be categorized into two classes: loss-based and delay-based.

2.7.1 Loss based algorithms

The core principle of loss-based congestion control algorithms is to treat packet loss as a congestion signal. Packet loss typically happens when a router's buffer is full, which means the network is congested, and the sender must slow down. Conversely, if no packet loss is observed, the sender assumes the network has more capacity and can increase its transmission rate. Packet loss is usually detected using the receiver's acknowledgments.

TCP Reno

TCP Reno employs a congestion control strategy based on three main phases: slow start, congestion avoidance, and fast retransmit/fast recovery. During slow start, the congestion window increases exponentially until a loss is detected, at which point

the algorithm transitions to congestion avoidance, where the window size increases linearly.

TCP NewReno

An improvement over TCP Reno, TCP NewReno enhances handling multiple packet losses within a single data window. It improves the recovery process by addressing issues related to partial acknowledgments and multiple lost packets.

QUIC Congestion Control

QUIC's default congestion control algorithm is similar to TCP NewReno. However, users can replace this algorithm with an implementation of their own. Thanks to QUIC running in userspace, this is relatively easy to do.

2.7.2 Delay Based Algorithms

Delay-based congestion control is a technique that relies on network delay as a key metric to adjust the sending rate of data packets, making it a popular approach for mitigating network congestion. These controllers seek to balance network load by detecting early signs of congestion through variations in delay rather than waiting for packet loss.

For QUIC, there have been efforts to add measurements for a one-way delay, which would serve as an ideal metric for such algorithms. [Hui22]

The following is a small selection of existing delay-based approaches.

LEDBAT

LEDBAT (Low Extra Delay Background Transport) is a congestion control algorithm designed to be used "by background bulk-transfer applications" to minimize the impact on foreground, interactive applications[Sha+12]. It aims to ensure that those transfers avoid congestion and thus do not interfere with real-time applications like video calls or online gaming, which are sensitive to latency and jitter.

It measures the one-way delay within a TCP connection by adding a timestamp to each sent packet. Upon receipt, the sender then substracts this timestamp from the current time and sends this difference - the resulting delay - back to the sender. If the sender notices this delay is increasing, it deduces that the TCP packets have to be buffered for a longer time at some point in transmission, reducing the data-sending rate. If the measured delay is sufficiently small, the number of bytes that may be sent within a given time window is increased.

SCReAM

SCReAM (Self-clocked Rate Adaptation for Multimedia) congestion control is a Congestion Control algorithm developed at the Ericsson Company[JS17]. It is based on LEDBAT but designed explicitly for the RTP protocol. Since RTP aims to transmit real-time data such as voice or video, often over mobile networks where conditions fluctuate, SCReAM seeks to adapt the quality of the sent data to the available bandwidth.

GCC

Like the approach mentioned above, the Google Congestion Controller (GCC) is also a congestion controller for real-time communication over WebRTC[Car+16]. And just like the approach mentioned above, it measures the one-way delay to decide on the sending rate. As the data being transferred is not meant to avoid congestion but rather maximize the quality of communication, packet loss is likely to occur and thus used as an additional metric. When packet loss is detected, the media's bitrate is immediately reduced.

2.7.3 Coupled Congestion Control

Raiciu et al. propose a multipath algorithm for the scenario of multiple TCP connections for the same combination of sender and receiver. Where the congestion controllers of the respective connections would normally lead to unnecessary congestion among each other, Their approach aims to enable cooperative behavior among them by linking their increase functions and controlling the overall aggressiveness of the controllers to increase overall efficiency[RHW11].

3 Motivation

To efficiently transmit data for various purposes within a single connection, considering differing real-time constraints, priorities, and data volumes is essential. Utilizing separate connections for each data stream could lead to the congestion controllers of these connections competing against one another for the available network capacity. This scenario is particularly problematic because a connection with a higher data volume would consume a disproportionate amount of the available bandwidth. Consequently, this would severely hinder a less intensive connection that is responsible for transmitting higher-priority data.

To mitigate such congestion issues, we propose enabling these data streams to collaborate within a single connection. This approach would allow for the prioritization of certain data transmissions over others, ensuring that high-priority data can be sent efficiently even if the overall network load is high. By implementing a cooperative model for data streams, we can achieve a more balanced and effective utilization of network resources, thereby improving the performance and reliability of data transmission for applications with varying demands. More specifically, if we are able to merge different streams with different uses into one application, we are able to use a single congestion controller for these, greatly reducing congestion among the same machine.

3.1 Stream Prioritization

Network stream prioritization is a fundamental concept in network management, aimed at optimising the delivery of data across a network by assigning different levels of priority to various types of traffic. As networks support a range of applications and services, prioritization ensures critical traffic receives the necessary resources to function effectively.

3.1.1 Challenges

Stream prioritization can significantly improve the performance of a program and consequentially user experience. Implementing effective prioritization comes with several challenges, however. As networks grow in size and complexity, effective prioritization becomes more challenging as such large-scale networks require sophisticated management tools and techniques to ensure consistent performance across diverse traffic types.

Furthermore, network traffic patterns can be highly dynamic, being subject to changes in traffic volume and types over time. Effective prioritisation strategies must be adaptable to accomodate these fluctuations and ensure continuous performance.

3.1.2 Application-Level Prioritization

Some applications include mechanisms to request higher priority for their traffic. For example, modern streaming services and real-time communication applications may use application-layer protocols to signal their priority requirements.

3.1.3 HTTP/2

HTTP/2 introduced stream multiplexing and prioritization, which for example enables web browsers to load contents of a website in parallel. The specification for signaling priorities is complicated, and many implementations still have issues and are not complete.

If you want to leverage HTTP's mechanism, you naturally need to use the protocol. With HTTP being a layer 7 protocol, this poses a restriction for constrained environments who do not want to use a high level, intense protocol.

3.2 Contribution

Thus, we introduce such a stream prioritization mechanism to the transmission protocol QUIC. With QUIC being a layer 4 protocol, practically all applications sending data over the internet can make use of it. Given a connection, QUIC enables you to open up multiple streams over which data is sent. Using the quic-go implementation, we add the possibility to easily prioritize single streams. The throughput of these streams is monitored, and the throughput of other streams is adjusted to account for changes in latency or throughput. We enable simultaneously having both streams for real time communication as well as streams for high bandwidth data transfer.

The innovation of this work lies in designing a congestion controller that can be used for two types of data at once (real time communication and data transfer) and optimizes the sending process with regard to the performance of the real time data. This is achieved without adapting the QUIC protocol.

4 Design

This chapter will outline the techniques used and deduced to implement the real-time media prioritization scheme.

4.1 Linear Regression

Linear regression is a statistical method widely used to model and analyze the relationships between a dependent variable and one (or more) independent variables. It is one of the simplest and most commonly applied forms of predictive analysis in various fields. The primary objective of linear regression is to establish a linear relationship of the form $y = mx + b$, where y represents the dependent variable, x is the independent variables and m determines the slope of the regression line.

Given a set of observations of random variables, the regression line is calculated to best fit the observations. The estimation of regression coefficients typically utilizes the least squares criterion, which aims to minimize the sum of the squared differences between observed and predicted values.

4.2 Overview of quic-go

Quic-go allows the overlying application to initiate a connection to another party. Given a connection, the application can create streams that serve as uni- or bidirectional lossless data channels. Data can then be written to these channels, and quic-go handles the transmission to the other party.

As QUIC utilizes UDP, the data written to these various streams must be transferred to UDP packets to be sent over the network. The *Framer* is responsible for assembling data from different streams and packing it into a sequence of bytes that ends up as a packet, which is then sent. The *Framer* will be called repeatedly as continuous data is available. Keeping a queue of active streams, the *Framer* builds a frame with data from all streams having bytes they seek to transfer, trying to exhaust the limit of available bytes.

The *Framer* is where we will inject our mechanisms, as we want to control what is being sent at a stream-granularity.

4.3 Idea

Suppose we have a video call and decide to transfer a file with our peers. If we do so with a separate process, this process might start hogging much of our bandwidth. Ideally, we send the file without a noticeable drop in the quality of our video call because only the excess bandwidth is utilized.

4.3.1 Properties of data streams

However, there is a difference between data from a file and data from a real-time video source: While the data from the real-time source is limited - a camera can only record with a certain framerate - data from a file is practically unlimited, as the speed with which it can be read from storage is vastly greater than the rate at which it can be sent over the internet. Additionally, this little amount of data is only available at discreet points in time, which would be every 33 milliseconds for a 30FPS camera. Thus, a non-prioritizing algorithm will end up sending vastly more file data than video, likely causing the video stream quality to deteriorate.

Hence, the need to prioritize streams and some "reservation of bandwidth" mechanism arises.

4.4 Earlier approaches

As a first experiment, we used a probabilistic approach. In the framer, we skipped adding data from the framer with a 50% chance. This way, we hoped to prevent the framer from using all available network capacity for the low-priority stream.

The probabilistic approach suffered from this "over availability" of file data. Even if you skipped data from the file stream, it would still dominate the available bandwidth and cause the video stream to deteriorate, as the function was called a multitude more often for file data than video data.

Thus, we find a way to track how much data is being sent so that we can make a qualified decision instead. The congestion controller already tracks the number of sent bytes. Still, we need a more fine-grained separation at the stream level to see if we are sending too much low-priority or too little high-priority data.

4.5 Balancer

The *Balancer* is the core class of our addition. As a member of the *Framer*, it is queried during frame assembling. The *Balancer* then decides whether a given stream can be read from at the given point in time or not. An overview of its structure is given in Figure 4.1.

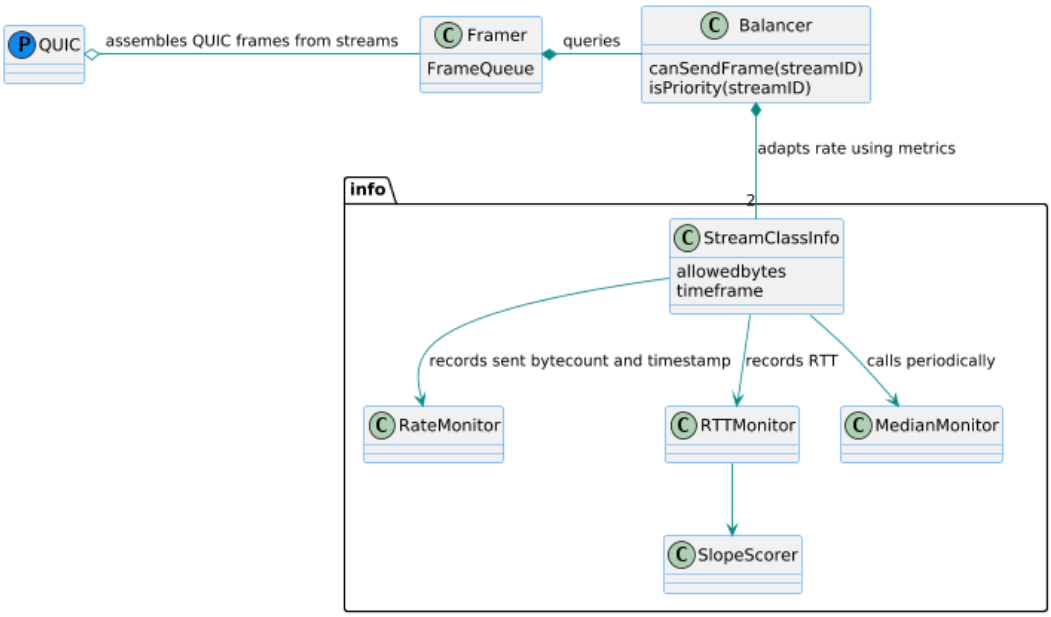

Figure 4.1: UML Diagram depicting the structure of our addition

In the *Balancer*, we log the number of bytes sent for two classes of streams: For the streams prioritized by the application (high-priority data) and for the rest (low-priority data).

At first, we tracked how many low-priority bytes we had been sending over a certain period. If too many bytes were sent within this last timeframe, no more low-priority data could be sent. As time advances, this count would naturally decrease, and the window would be open again. By hardcoding a threshold, we could ensure that there was always enough bandwidth for the priority data.

The limitation of this hardcoded threshold is twofold: We need to know about the capacity of the link, If the volume of priority data increases and we already use the total capacity, congestion will occur. If the link's capacity increases, we will no longer utilize the total bandwidth. If it decreases, congestion will occur. Therefore, we want to monitor the throughput of low-priority and high-priority data. If we notice a drop in the high-priority throughput, we can infer a drop in the connection quality and react accordingly.

4.5.1 Dynamic quality metrics

Since the criteria for connection quality are high-priority data, we can now monitor its throughput and respond to its development. We no longer need to resort to a hardcoded limit for the low-priority data. Still, we can adapt this limit to the overall available bandwidth and the amount of high-priority data being sent.

We ended up with many features on which to base our decision based on low-priority data throughput.

How do we notice a drop in the quality of priority data? We want an application and bandwidth-agnostic implementation, meaning our idea should work in both network environments with low or higher available link capacity.

Rate

If we assume the data rate will be somewhat uniform, we can measure the relative development of the rate. When we notice a sudden drop, we can quickly adapt the amount of other data sent to ensure enough bandwidth for the high-priority connection.

To approximate the rate of data sent, we record the number of bytes packed into a frame by the *Framer* in conjunction with the time of sending. Using linear regression, we can obtain a slope proportional to the rate. By applying this technique over timeframes of different durations (e.g., over the last second and in the previous five seconds) and comparing the resulting slopes, we can deduce if the rate of sent bytes is changing or has been steady.

Round Trip Time

We also had the idea of using the round trip time (RTT) as an indicator of the connection quality. At first, we wanted to detect a change of quality simply by comparing the RTT to previous values and trigger on, e.g., a 20% increase. Unfortunately, this approach suffered from a few things: Firstly, small increases in RTT, such as from 10ms to 20ms, are a relative increase of 100%. This means that simply using this percentage-based detection, reasonable variations would already be *an anomaly*. Secondly, at startup, the RTT naturally increases. To an extent, this is also true when you increase the volume without reaching the link capacity. Overall, hardcoded values regarding the increase of RTT thus quickly reached the limit of their usefulness.

Slopescorer

To detect substantial changes in RTT, we again used linear regression. If there suddenly was a positive slope, we infer the RTT has increased. Again, we use linear regression. This time, the line we are *drawing* using linear regression is - under stable conditions - even, as the RTT is expected not to drastically vary under a stable connection. More mathematically, the slope will be close to zero under stable conditions and very large during sudden changes in RTT.

The problem of this enormous increase was tackled using a *Slopescorer*. The *Slopescorer* keeps track of the highest encountered slope value and normalizes a given value w.r.t. this maximum. The maximum is updated if the given value is larger than the maximum.

To account for outliers and changing circumstances, the saved maximum is shrunk over time.

Median Throughput Monitor

Sometimes, there is a decrease in quality, for example, because another process on the machine starts sending data. This will lead to less available bandwidth. Before the change is detected, much non-priority data is still being sent, causing a drop in high-priority throughput. This will be detected, and the low-priority throughput will be adjusted. The algorithm will notice if the tendency of the high-priority throughput is no longer decreasing and assume stability. This approach bears an issue: if the timeframe over which said tendency is analyzed is too short to include the time the decrease started, we will not come back to the previous rate. Instead, we will have forgotten the better yet achievable level.

To adjust for this, we introduce another feature capable of keeping track of the bandwidth over a longer time: The *Median Througput Monitor*. It periodically records the current bitrate and saves the ten largest recorded values. To account for outliers, the absolute maximum is not returned. Again, values are slowly reduced over time to account for outliers and changes in environment. This throughput monitor serves as another metric to which the current bitrate can be compared to decide if the high-priority data's throughput is adequate or not back to a normal level yet.

4.6 Adaption Algorithm

The algorithm adapts the *allowedbytes* of low-priority data. If the metrics indicate a stable connection, the limit is raised. Thus, even under no changes in the environment, the limit will eventually run into the maximum link capacity, causing the high-priority data throughput to drop. Once this drop is detected, the limit is adjusted, and the high-priority data returns to normal. To reduce the excess of low-priority data sent, we want to reduce the excess increase of *allowedbytes*. To do so, we remember the last time we exceeded the link capacity. If we approach that value, we significantly reduce the speed at which we increase the low-priority window. We then have a longer timeframe until this limit is inevitably exceeded again.

This exceeding is inevitable because we want to be able to use more of the link capacity incase it has increased. If we were to never exceed a determined limit, this potential could not be detected and would go unused.

The algorithm for determining the new low-priority window is run periodically. Using the metrics above, a $growthfactor$ is calculated. Then, the window is multiplied by that $growthfactor$.

4.6.1 Growth phases

Similar to TCPReno, we introduce separate phases to avoid repeatedly causing congestion. The phases are dependent on the current *allowedbytes* value and the largest value before it was decreased. Essentially, if the link capacity is stable, congestion will be caused at a similar value as last time. By remembering this value, we grow at a slower rate when it is soon to be reached. If congestion is indeed caused with a similar value, the *allowedbytes* value is decreased gently.

5 Implementation

This chapter explains the concrete implementation of our approach in detail.

5.1 Gstreamer

GStreamer is a powerful and versatile multimedia framework designed to facilitate the development of complex audio and video processing applications. Initially developed by the GNOME project, GStreamer is an open-source framework that provides a comprehensive set of tools for building multimedia pipelines. It supports a wide range of audio and video formats and codecs, making it a popular choice for desktop and embedded systems.

GStreamers architecture is based on a pipeline model, where multimedia data flows through processing elements. The core components of GStreamer include:

- Elements: The basic building blocks of a GStreamer pipeline are elements that perform specific operations on multimedia data. Elements can be sources, sinks, or filters. Sources generate data, sinks consume data, and filters process data in various ways.
- Pipelines: A pipeline is a container for a series of connected elements. It manages the flow of data through the elements, ensuring that data is processed in the correct order. Pipelines can be dynamically constructed and modified at runtime.

GStreamer offers several features that contribute to its flexibility and power:

GStreamer's design allows the creation of custom elements and plugins, extending the framework's capabilities. This modularity supports a wide variety of multimedia formats and processing tasks. The framework uses a plugin-based architecture, where functionality is provided through dynamically loadable modules. GStreamer includes many built-in plugins for handling different media types, and additional plugins can be created or obtained from the community. GStreamer supports streaming protocols and technologies, allowing for the development of applications that handle live streaming, broadcasting, and networked multimedia. It includes support for RTSP (Real-Time Streaming Protocol) and RTP (Real-Time Transport Protocol). Finally, mechanisms for synchronizing audio and video playback, as well as buffering to manage data flow and handle variations in processing speeds, are provided.

5.2 QUIC-GO

quic-go is the first QUIC implementation in Golang. It was presented by Lucas Clemente in 2016 [1] and is open-sourced. [2] The congestion control algorithm implemented is *CUBIC*.

5.3 Golang

Go, also known as Golang, is an open-source programming language designed by Google. Launched in 2009, Go was created to address the challenges of building software with high degrees of parallelism.

Go is distinguished by several key features that contribute to its effectiveness and popularity:

- Performance: As a statically typed, compiled language, Go delivers high performance comparable to languages like C and C++. The Go compiler produces efficient machine code, and the language's design ensures that programs execute quickly and with minimal runtime overhead.
- Concurrency: Go provides built-in support for concurrent programming through its goroutines and channels. Goroutines are lightweight threads managed by the Go runtime, allowing developers to write concurrent code easily. Channels facilitate communication between goroutines, enabling efficient data exchange and synchronization.
- Standard Library: Go comes with a comprehensive standard library that provides various packages for various tasks, including networking, cryptography, and data manipulation. The standard library's breadth and consistency contribute to rapid development and code reuse.

The language has also gained recognition for its role in modernizing systems programming and contributing to developing scalable and high-performance applications.

5.4 Quality measurement features

This subsection addresses the implementation of various components responsible for measuring the quality of the connection.

[1] https://www.youtube.com/watch?v=BvpRBdWwecU
[2] https://github.com/quic-go/quic-go

5.4.1 RateMonitor

As explained in the Design chapter, we are able to measure the respective throughput for the classes of high and low-priority data. Additionally, we can quantify the trend of throughput, meaning we can tell if the throughput of data has been increasing, decreasing, or steady.

To measure the relative change in throughput, we record the total number of bytes sent whenever a new packet is recorded and the time it is sent. The size of this packet is naturally added to the total number of bytes sent. These tuples can be interpreted as coordinates on a two-dimensional plane.

With these tuples of timestamps and total bytes sent, we can use linear regression to estimate the rate of data that was sent. With this linear regression, we try to fit a straight line onto these coordinates. The slope of the resulting line is the rate at which data has been sent and recorded.

By employing this linear regression over different timeframes, e.g., over the last second and the last 5 seconds, we can detect if the rate has decreased within the last five seconds: The slope of the resulting lines can be compared by simple division: By dividing $\frac{slope(1s)}{slope(5s)}$ we have a value indicating the development. This value is called the *RateStatus*.

This process is computationally intense because several steps are done whenever the rate is estimated: For each timeframe specified, all *totalbytessent* samples within that timeframe are converted to input for the linear regression model. Then, the actual regression is performed.

Alternatives

Alternatively, we could have calculated the rate in a simpler way: To determine how many bytes were sent in the last second, we can take the totalbytessent sample closest to exactly 1 second ago and subtract that value from the current totalbytessent value. This would favor some executions over others, as sometimes the sample would be older than at other times. Overall, this approach would not be as computationally intense.

An earlier design tried to estimate the amount of sent bytes by using the $bytesinflight$ value provided by the congestion controller. The advantage of this module is that it is both more fine-grained (per class) and also easily extendable in case one would like to introduce even more classes of data streams.

5.4.2 Round Trip Time Monitor

Similar to the Rate Monitor, we also monitor the Round Trip Time (RTT) development. Whenever a new RTT sample is sent and recorded, the last observed RTT and the current time are stored, forming a cloud of points on a two-dimensional plane on which linear regression is performed. Different from the rate monitor, however, is the

behavior of the resulting line under stable conditions: For the sent bytes, the lines for two timeframes would both be going steadily upwards with the same slope, as the rate has been steady and data has been sent continuously. On the other hand, the line for the RTT Samples Regression would be horizontal because the RTT was the same throughout the observation timeframe. This means that instead of comparing the slopes of two timeframes, we can directly use the resulting slope, as any non-zero slope implies a change in conditions.

Slopescorer

At the start of transmission, an increase from, for example, 10ms to 20ms is a 100% increase but still completely tolerable. The normalization works by employing a *Slopescorer*. The idea is to be variable in scoring the magnitude of the RTT to account for different network environments, with the distance between the client and server being a commonly varying variable significantly affecting the RTT.

This *Slopescorer*keeps track of the maximum slope value observed and normalizes a given value to it. If there suddenly was a significant positive slope, we infer the RTT has increased. The maximum is updated if the given value is larger than the maximum. To account for outliers and changing circumstances, the saved maximum is shrunk over time.

Consider this table 5.1 as an example of such a slopescorer output. As in our final implementation, the maximum is shrunk by 1% each timestep.

Time	Input	Maximum	scored output
0	100	100	1
1	22	99	$\frac{2}{9}$
2	-1	98.01	$\frac{1}{98.01}$
3	4000	4000	1
4	100	3960	$\frac{100}{3960}$

Table 5.1: Periods of the experiment

5.4.3 Median Throughput Monitor

Every 100ms, this monitor calculates the current bitrate and saves it. This is then compared to previous samples; the ten largest recorded bitrate values are stored. Further, the stored values are shrunk by 1% each timestep.

5.5 Algorithm

With the features for measurement of quality defined, it is time to delve into the algorithm plugging these features together. As mentioned before, a $growthfactor$ is calculated with which the allowedbytes of the rest class is multiplied.

The features are designed to be a value indicating neutral development, as in no change, at 1. If the quality of, e.g., the round trip time has worsened (increased), the according value of the feature is smaller than 0.

Naively, we can calculate the $growthfactor$ by forming the product of these three features:

1. RateStatus
2. RTTStatus
3. CurrentToMaximum

However, experiments show that a slightly more sophisticated approach yields better results.

The following will describe how each feature is manipulated and eventually added to the $growthfactor$. Since we are forming the product, the order in which these features are applied is irrelevant.

RateStatus

Arguably, the most crucial feature, the RateStatus, illustrates the rate of recently sent bytes compared to a larger timeframe. To avoid small fluctuations negatively influencing the transfer (e.g., a RateStatus of 0.95), we do not shrink the $growthfactor$ if the RateStatus is still larger than 0.9. Further, we square the value to give more weight to extreme throughput changes. The steeper the dropoff, the faster we want to react. Since a substantial dropoff in link capacity has likely led to some buildup of data that has been sent but has yet to arrive, we want to change the capacity disproportionally. With consistency being our goal, we accept the risk of not utilizing the bandwidth optimally.

For the case of the RateStatus being positive, indicating the rate of bytes has been increasing, we use the RateStatus value. To account for extreme outliers that might happen after a period of very little data being sent, maybe because there was simply no data available, the value is capped at 1.5.

An advantage of this: By not relying on the bytes in flight metric, our implementation will work the same way for setups where server and client are further apart, where multiple hops in between the sender and receiver might lead to a considerably larger amount of bytes in flight under otherwise very similar conditions.

Finally, the value that ends up being multiplied with the growthfactor is given in Figure 5.1:

$$factor(RateStatus) = \begin{cases} rateStatus^2 & \text{if } rateStatus < 0.9 \\ 1 & \text{if } 0.9 < rateStatus < 1 \\ \min(rateStatus, 1.5) & \text{if } rateStatus > 1 \end{cases}$$

Figure 5.1: Formula for the RateStatus

RTTStatus

The RTTStatus indicates how the RoundTripTime has been developed recently. More precisely, it indicates how the latest RTT compares to the maximum. If the RoundTripTime is at least 80% of the all time maximum, we deem this as problematic and reduce the growthfactor by 30%.

CurrentToMaximum

This feature is the current throughput of data compared to the maximum throughput observed in the session, recorded by the *ThroughputMonitor*. It is meant to discover a stable but worse condition, where the RTT is stable, and the SendingRate has been sub-par for a time long enough that it is no longer noticeable by the Rate monitor.

Again, we square the value to better react to lower values. If the current throughput is bigger than the last observed maximum, we do not modify the growthfactor. The complete formula is thus given by Figure 5.2.

$$x = \frac{current\_throughput}{max\_throughput} \qquad f(x) = \begin{cases} x^2 & \text{if } x < 1 \\ 1 & \text{if } x >= 1 \end{cases}$$

Figure 5.2: Formula for factoring in the current throughput

5.5.1 Capping

After determining the new growthfactor using the features above, we limit the growthfactor to 1.5 to avoid a growth that is too explosive. Outlier values could influence the calculation too much otherwise. Because this process is repeated periodically, a shortcoming, meaning an allowedbytes correction is too small, will simply be corrected in a subsequent call. Further, it is much easier to exponentially grow to a large, adequate value than to shrink back from it.

Under stable conditions where all requirements for the prioritized data are met, we will allow more low-priority data to be sent, thus increasing the allowedbytes value. However, if the low-priority data are not exhausting the allowed bandwidth, we must prevent further growing the allowedbytes. If we do not, a suddenly worsening situation

will take more time to be adapted to, as the algorithm will need to start reducing the allowedbytes from a much higher, inflated value.

Thus, as a final condition we check if the rest of the streams are utilizing at least 90% of the allowed bandwidth, and set the growthfactor to 0.99 if they are not.

5.5.2 Growth Phases

We introduce several growth phases. With the current state, we are always going to run into a limit because we keep on increasing the allowedbytes of the rest streams, causing an overshoot, a drop in throughput for the high-priority data, leading to our algorithm significantly reducing the allowedbytes for the rest streams. To limit this drawback, we want to gently increase the rest streams allowedbytes when we know we are approaching the limit, similar to the congestion avoidance phase of TCP NewReno. Likewise, we avoid radically shrinking the limit if it has led to a shrinking at roughly the same value it did the last time. This would indicate that we again have reached the same network capacity limit, but the network environment has not changed.

We thus arrive at four different growing stages:

1. INCREASING
2. DECREASING
3. INCREASING_SLOWLY
4. DECREASING_GENTLY

This state serves as a memory for what happened in the last invocation of the algorithm. Another variable we introduce is the *lastmax* value, which saves the last value of allowedbytes before it has to be reduced.

When there is a downward growth change, when the growing state is at either INCREASING or INCREASING_SLOWLY, the current bytesallowed is compared to it. If the values are somewhat similar, we assume to have hit the same limit as last time and that the network environment has not changed. Thus, to not unnecessarily shrink the allowed bandwidth excessively, we set the new state to DECREASING_GENTLY. If the downward change happens at another capacity, we assume the network environment has changed because link capacity has reduced or other senders are causing congestion. In this case, we set the state to DECREASING.

If we are not at a downward change but continuing to grow, we either grow normally or, if we detect to be close to the last limit that caused failure, grow gently, where we significantly soften the growthfactor before applying it to the allowedbytes value. If we exceed the *lastmax* value, the network environment has changed, and we resort to growing normally again. Naturally, the *lastmax* value is updated.

If we are continuing to shrink, we simply stay in the same type of DECREASING state, with one exception: Suppose we decrease gently and circumstances change,

$$allowedbytesfactor(stage, gf) = \begin{cases} gf & \text{if } stage == (INC, DEC) \\ 0.95 & \text{if } stage == DEC\_GENTLY \\ \frac{10+gf}{11} & \text{if } stage == INC\_SLOWLY \end{cases}$$

Figure 5.3: Final smoothing of the growthfactor depending on the growing stage

which will be reflected in a very low growth factor. In that case, we resort to the regular DECREASING state to reach a low rest stream bandwidth utilization faster.

Depending on the growing stage, the growthfactor is factored in differently into the *allowedbytes* value.

Growing Stages Effect

Now that the four stages have been showcased, we finally explain how this affects the *allowedbytes* value and form the final $allowedbytesfactor$.

If the growing stage is INCREASING or DECREASING, the $growthfactor$ remains unmodified. In case it is DECREASING_GENTLY, we set it to 0.95. Lastly, if it is INCREASING_SLOWLY, the $growthfactor$ is smoothed to be closer to 1.

Finally, the new *allowedbytes* value is set by multiplying the old value with the derived $allowedbytesfactor$.

6 Evaluation

In this chapter, we finally seek to test and evaluate our supposed improvements and compare them to the existing solution.

6.1 Experimental Setup

First, we describe the setup used to evaluate and benchmark our modified QUIC implementation. The tests were run on a Debian 11 system, virtualized on QEMU.

6.1.1 Application

Since the goal to improve the performance of high-priority data was set with media transfer in mind, we constructed a small program that uses GStreamer to send a video from a server to a client. The GStreamer pipeline streams the popular BigBuckBunny[1] and encodes it to the H.264 codec. A new QUIC stream is opened for each video frame, over which it is sent using the RTP protocol. The server is configured to send RTP packets at a target bitrate of 2000 Kilobit per second. The time of sending/receiving each RTP packet is logged at both the server's and client's application layer. In the analysis, we can thus calculate the one-way delay for the packet to be transmitted — naturally, the lower the value, the better. In the following graphs, this value is labeled as "delta_to_client."

After a startup time of 10 seconds, the server starts sending additional random data. This data is meant to emulate the transfer of an arbitrarily large file and stress the implementation.

6.1.2 Testbed

We artificially impose a limit on the connection to better understand the algorithm's behavior and see how long it takes for the network capacity to be fully utilized again after an increase or decrease in capacity. We use Mininet [2] to construct a virtual connection to connect the server to the client. Mininet is widely used to test and develop new network protocols, experiment with Software-Defined Networking (SDN),

[1] https://peach.blender.org/
[2] http://mininet.org/

and provide hands-on learning for network architecture and operations. The sender and client are arranged using a classic dumbbell topology.

We have several periods in which we vary the bandwidth of the connection. The table 6.1 shows the different periods and their capacities.

Period #	Duration (seconds)	Capacity
1	120	2 Mb/s
2	60	4 Mb/s
3	60	0.6 Mb/s
4	60	1 Mb/s
5	60	3 Mb/s
6	60	1 Mb/s

Table 6.1: Periods of the experiment

6.2 Criteria

When we look at the following graphs, two features are of critical importance:

1. bandwidth utilization,
2. recovery after restriction and
3. latency of the real-time media

Bandwidth utilization means we want the total sending rate to be as high as possible throughout the run. Recovery after restriction refers to the sending rate of the media stream - our high-priority real-time data - remaining stable when the link capacity is suddenly reduced. The latency of the real-time media is to be kept low consistently to enable smooth communication throughout all conditions.

6.3 Results

We realize the experiment with four different random bitrate values, namely with 0 kbit/s, 500 kbit/s, 2 Mbit/s, and 6 Mbit/s. For each of these values, we ran the experiment with the quic-go version, on which we based our implementation as well as our final version. The direct comparison will emphasize the proposed mechanisms' advantages and drawbacks.

6.3.1 Baseline

To establish a baseline, the first run has no random data.

As there is no random data to balance, the throughput of both implementations is identical. The measured RTP delays are of the same magnitude, too. With no additional data being transferred, this is to be expected.

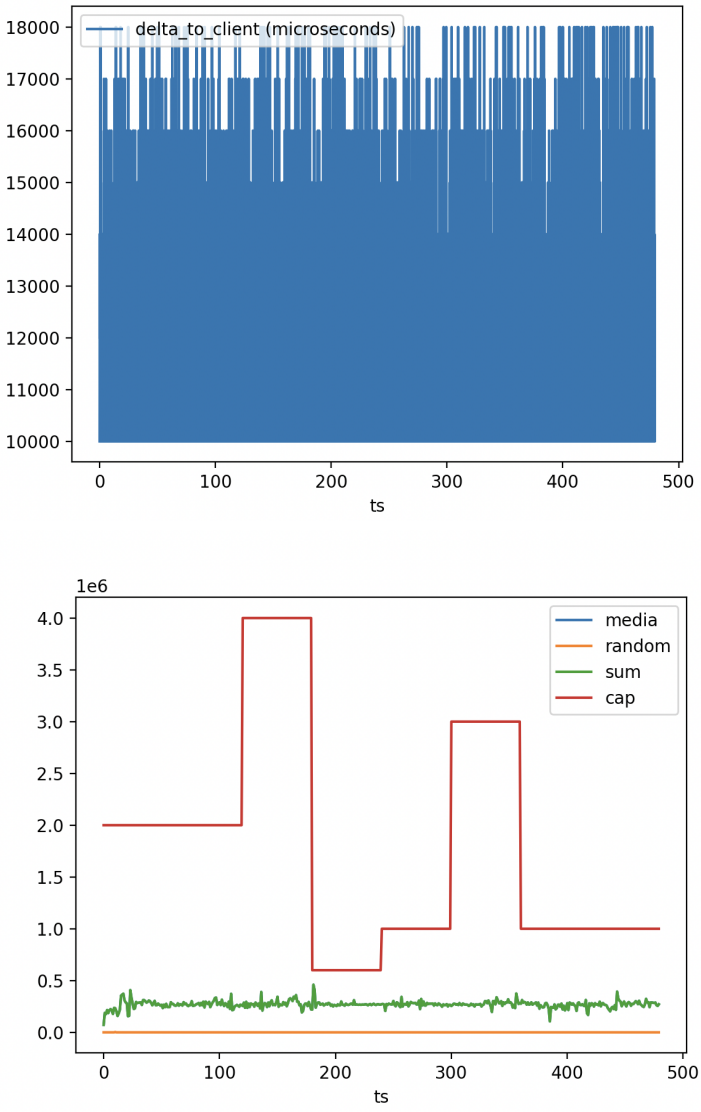

Figure 6.1: Metrics for modified quic-go without random data

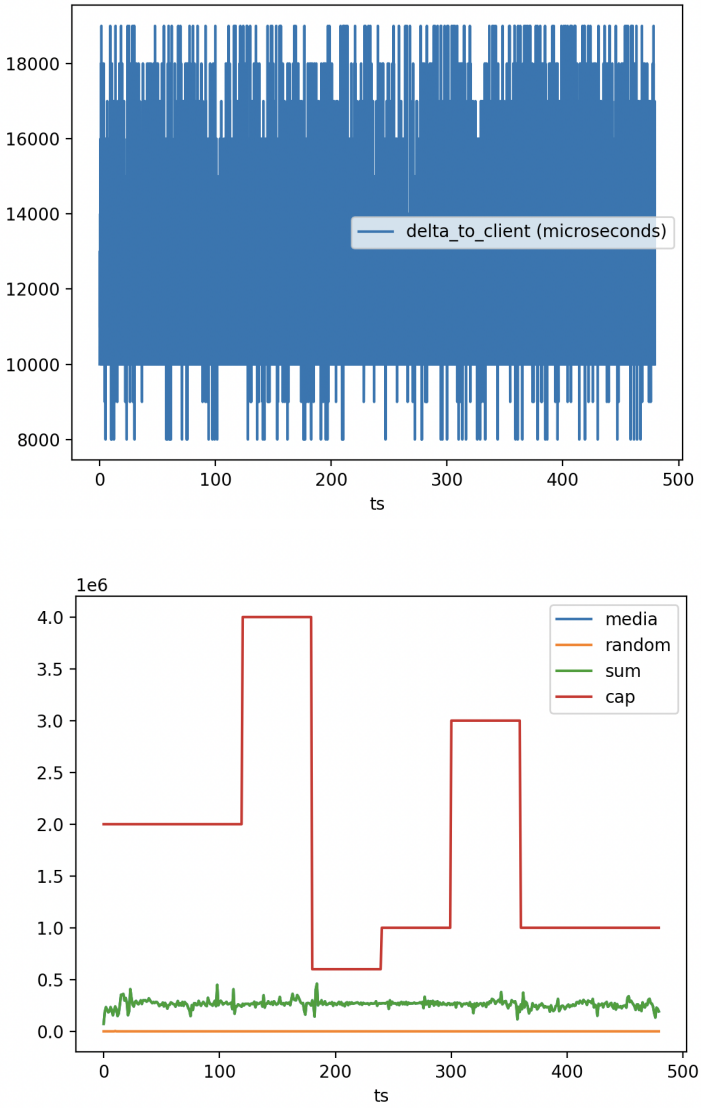

Figure 6.2: Metrics for original quic-go without random data

6.3.2 500 kbit/s

As a next step, we send 500kbit/s of random data. The network capacity is sufficient at this rate during all phases except the third one.

For our modified version, the transmission of the media data remains stable, and the spike of the RTP delta is very similar to the baseline behavior. The original implementation without prioritization has a better throughput during the phases with sufficient bandwidth. Where our version detects some drop in media throughput (e.g.,

around the 90-second mark) and subsequently reduces the random rate, the unmodified version remains at a very stable 500kbit/s of random data throughout the run, which comes at the cost of the media data reduced to near zero. While both RTP delays spike during the third phase, the spike is much smaller for our modified version. Thus, we can already see the advantage of our mechanism.

However, the disadvantage also becomes prevalent: due to the StreamBalancer noticing a drop in media throughput, the random media rate is throttled multiple times with enough bandwidth still available. In all cases, it takes more than ten seconds before random data is again sent in larger quantities, wasting potential bandwidth. Compare this to the total throughput of the original version, which is always near the maximum (at the cost of the media quality).

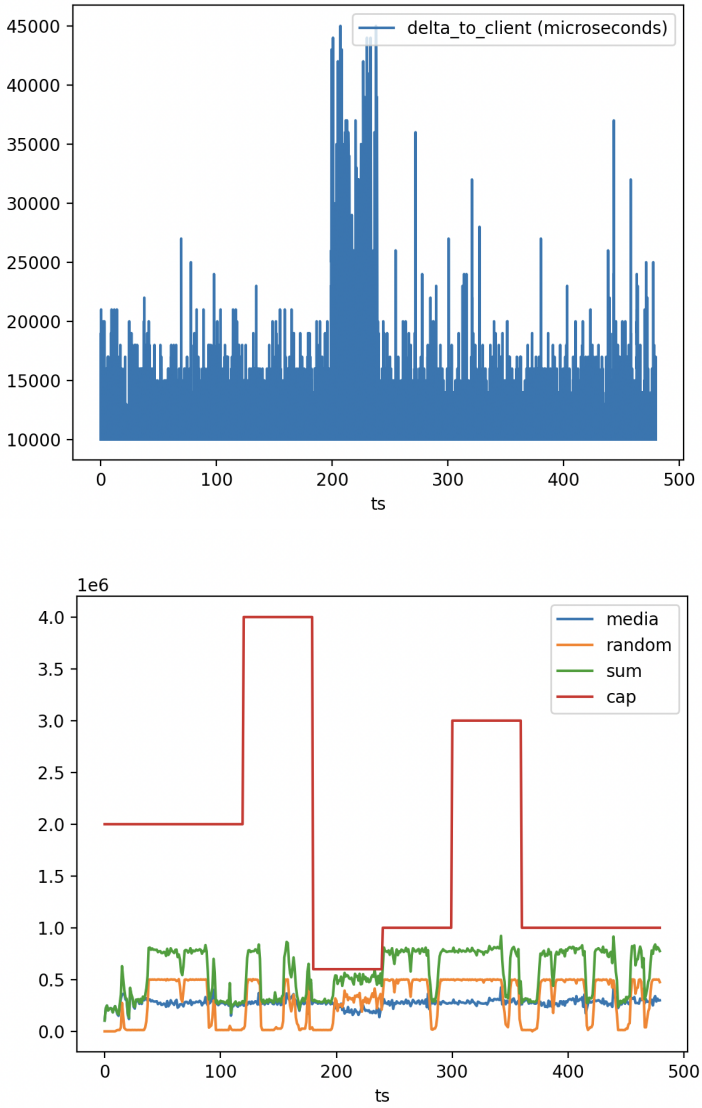

Figure 6.3: Metrics for modified quic-go with 500kbit/s random data

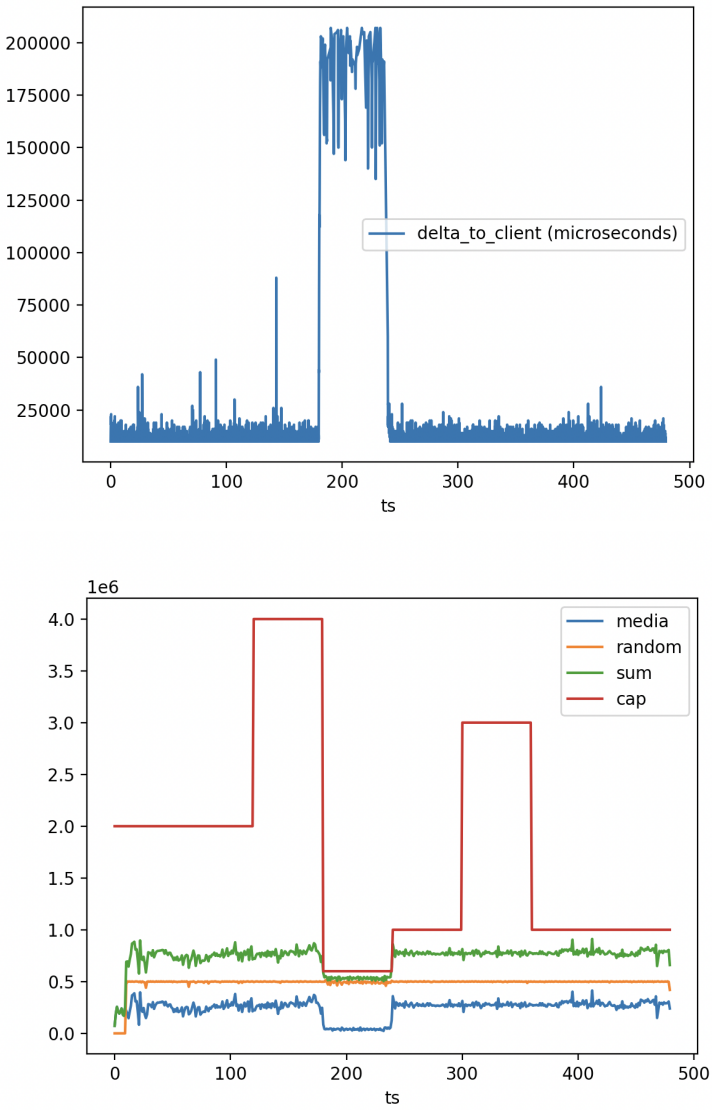

Figure 6.4: Metrics for original quic-go with 500kbit/s random data

6.3.3 2 Mbit/s

Next, we set the random data rate to 2 Mbit/s, leading to the sum of data exceeding most limits set.

For the original version, the media throughput suffers noticeably during all phases where the bandwidth is insufficient, similar to before. For the modified version, the media throughput stays stable while still achieving high rates for the random data, even reaching the maximum twice (around the 60 and 180-second mark, respectively).

Again, some dips in media throughput lead to prematurely reducing the random data rate.

The RTP delay measurements for the original version continue to be very high during phases of limited bandwidth. For the modified version, some spikes in delay are visible whenever there is an increase in random data rate, even when the sum of sent bytes does not approach the current bandwidth limit (e.g., at the 50-second mark).

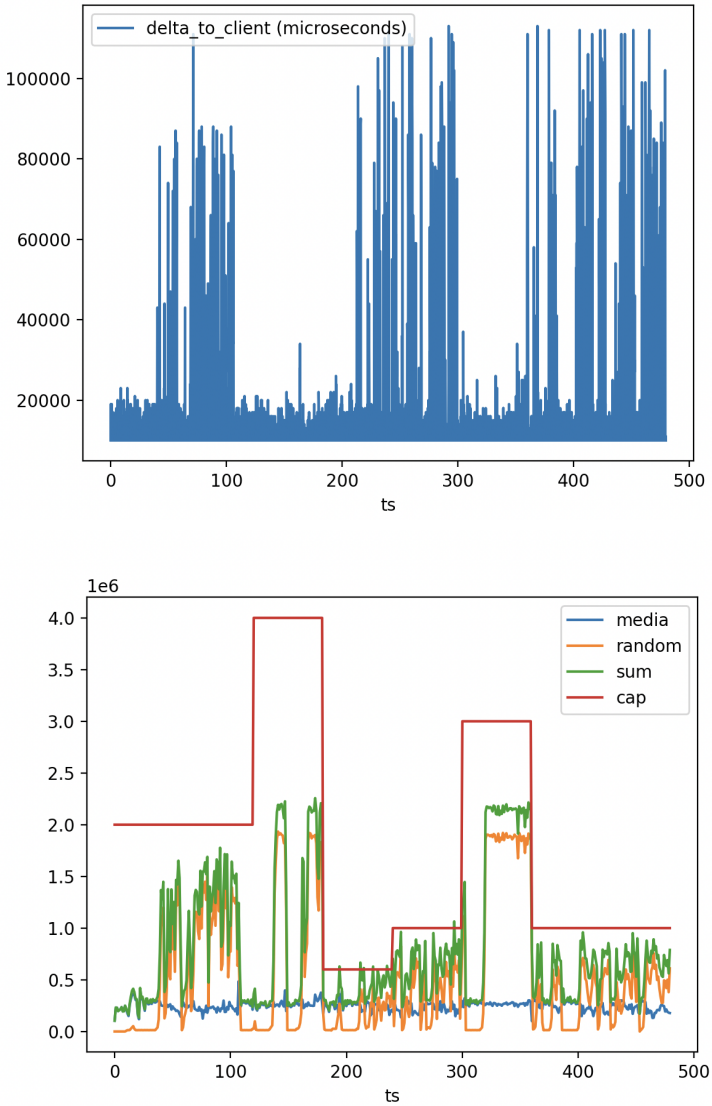

Figure 6.5: Metrics for modified quic-go with 2 Mbit/s random data

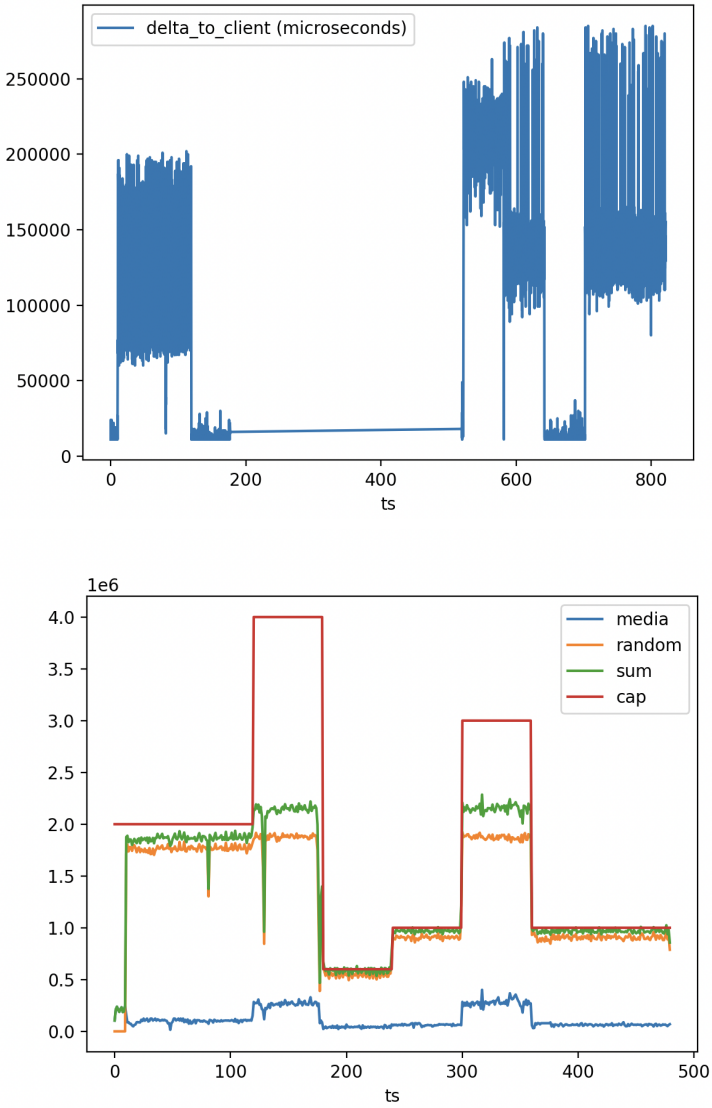

Figure 6.6: Metrics for original quic-go with 2 Mbit/s random data

6.3.4 6 Mbit/s

Finally, we set the random data rate to 6 Mbit/s, a value that will exceed the limits for all phases, meaning that for our modified version, a stable transfer rate for random data can never be achieved. Because the StreamBalancer tries to accommodate the high demand for random data, it sends too much data and detects a slight dip in the media transfer rate. This triggers a replacement of the *allowedbytes* value for the random data, leading to the zigzag pattern of the curve between seconds 40 to 90 in Figure 6.7.

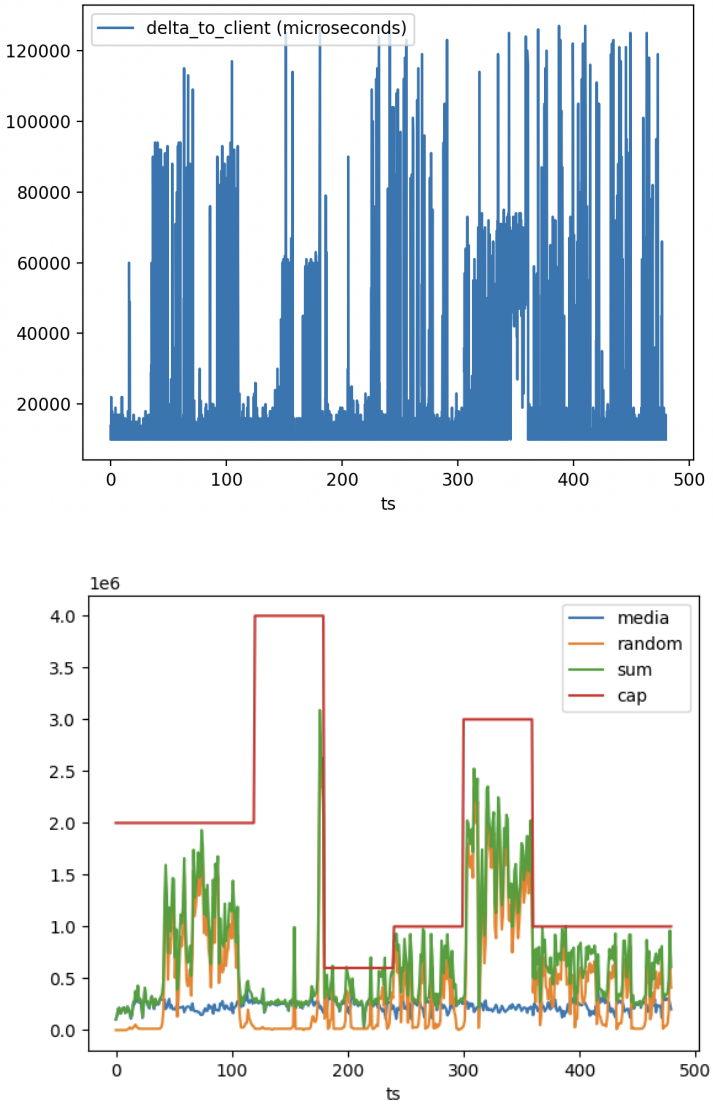

Figure 6.7: Metrics for modified quic-go with 6 Mbit/s random data

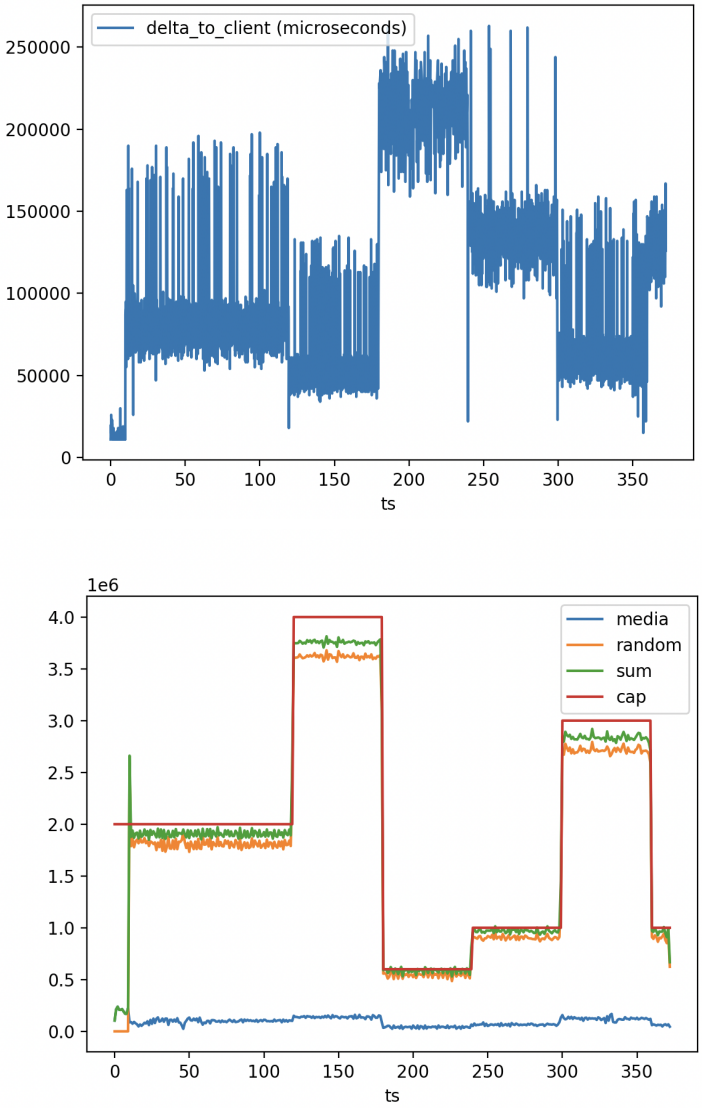

Figure 6.8: Metrics for original quic-go with 6 Mbit/s random data

6.4 Only Prioritization

In reference to the stream prioritization work by Fatima et al. [Fer+23], we conduct the same experiment with another modified version. We are left with strict prioritization of the streams by removing the call that queries the *Balancer* if a low-priority stream can be sent. As seen in Figure 6.9, more than such an approach is needed with such a disbalance between the high and low-priority data.

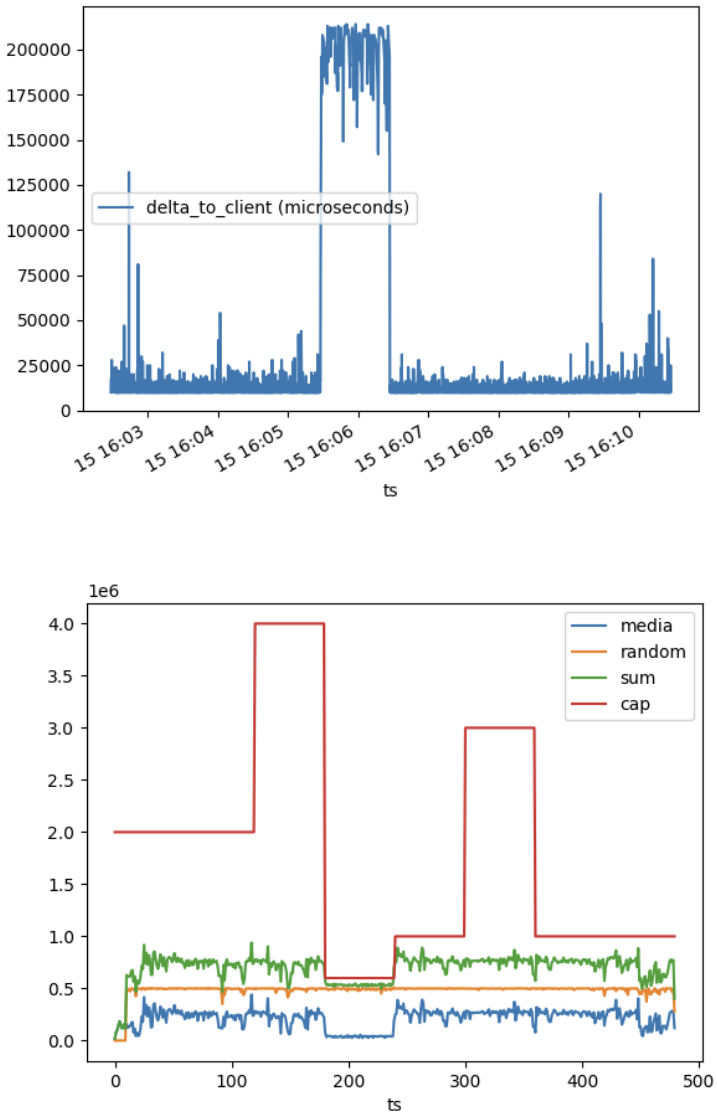

Figure 6.9: Metrics for modified quic-go only implementing strict prioritization with 500 kbit/s random data

6.5 Concurrent IPerf

As an additional test, we create a very large amount of congestion by running the popular speed test tool IPerf[3] over the same link. With our modified version, IPerf ends

[3]https://iperf.fr/

up completely dominating the link, leaving close to no bandwidth for our program to operate with. The resulting metrics are given in Figure 6.10.

Interestingly, the original version performs better under this scenario (Figure 6.11). While the media connection still starves, random data can still be transfered. This is likely related to the congestion controller, unmodified in both versions. Where our modified version would send such a small amount causing the congestion window to decrease, the high availability and sending rate of the random data causes the congestion controller to maintain the size.

To conclude, this hints that further expeditions should work in conjunction with the congestion controller.

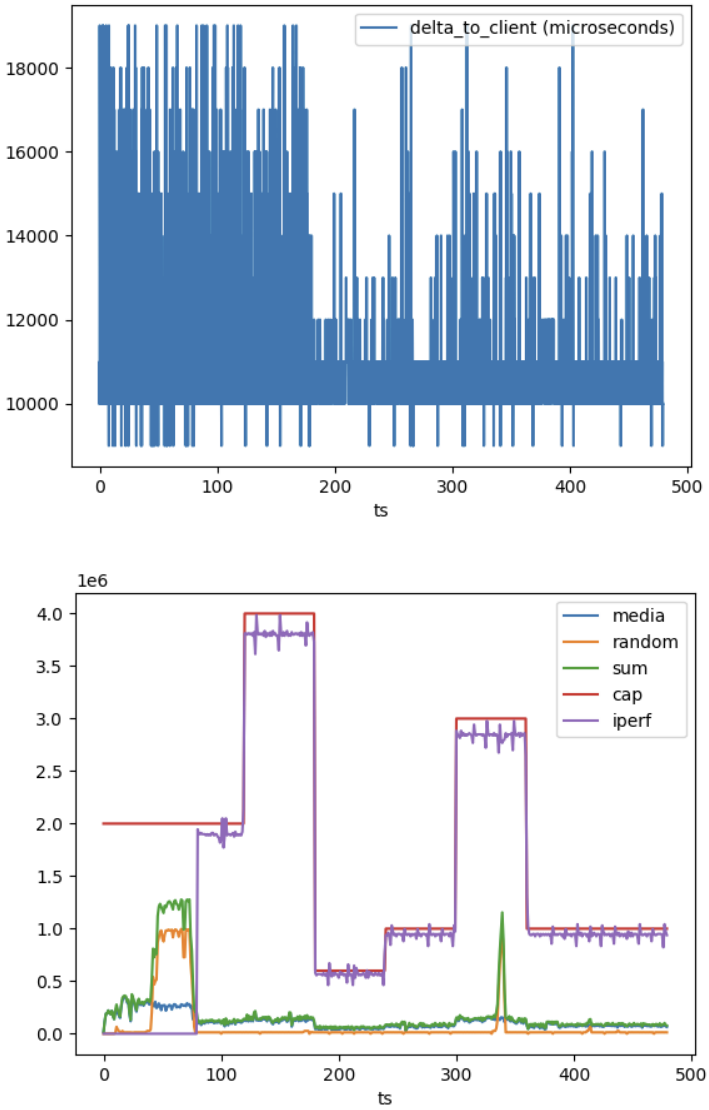

Figure 6.10: Metrics for modified version with IPerf, 1 Kbit/s of random data

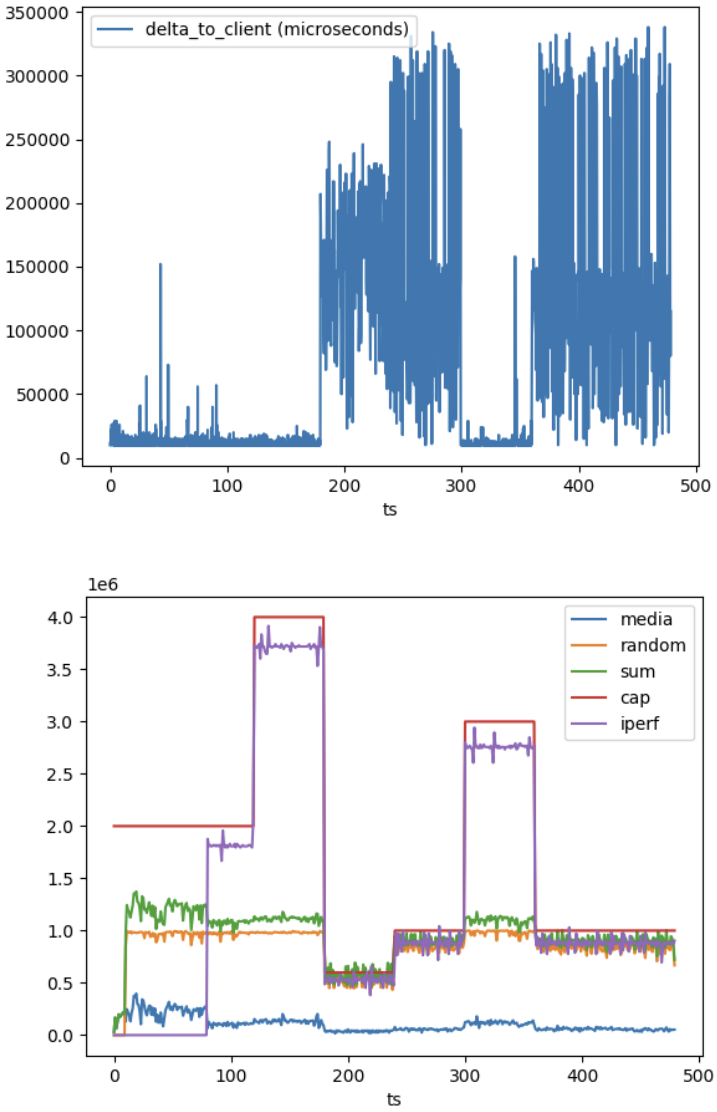

Figure 6.11: Metrics for original version with IPerf, 1 Kbit/s of random data

6.6 Findings

We see clear advantages in direct comparison to the naive quic-go implementation without prioritization. While the media rate under the original version plummets as soon as the limit is insufficient, the modified version quickly detects when less bandwidth is available and promptly reduces the throughput for random data.

6.6.1 Utilization

The bandwidth utilization still needs some improvement. Due to the nature of the current algorithm, which always seeks to increase the amount of data being sent, a limit in bandwidth is constantly hit, leading to a temporal decrease in media data. This issue of recognizing when an optimum state is reached remains a problem that needs to be researched. While one could stay at a steady bitrate once a bottleneck is detected to avoid sending too much (zig-zag pattern), this would close the connection to new bandwidth capacities opening up. Inspired by other congestion control techniques, it could be beneficial to introduce a second mode where the throughput is no longer increased, or at least not in the multiplicative way present.

6.6.2 Periodicity of Data

When looking at the performance of the unmodified quic-go stack, it is apparent that the random data was over-proportionally favored during periods of insufficient bandwidth. We believe the difference in nature of timings to be the cause for this: While a new media frame only periodically becomes available (every 33ms in the case of a 30FPS video), the writer writing the random data practically always has more data to write. Thus, the write calls to the quic-go stack are called – and capacity is used up again – before the new frame can be written. This highlights another vital difference in these types of data: The periodicity with which they are made available. Our experiment with the quic-go version only implementing strict prioritization further stressed this, with the media data rate decreasing similarly to the unmodified version once bandwidth was limited.

The finding regarding this difference suggests that a fair scheduler should not just consider the order of streams when choosing what to send next but also take the volume of recently sent data into account.

7 Future Work

In this chapter, we give an outlook on how our approach could be improved and what use cases might be applicable.

7.1 Improvements to our Implementation

Several improvements come to mind as our framework remains a mere proof of concept.

7.1.1 Introduction of several Classes

Right now, the framework allows adding a stream to the high-priority class, making sure this stream's data is sent as fast as possible. With the RateMonitor tracking the steadiness of the data, it is tailored to a steady stream of media. For broader applicability, one could add the option to specify certain goals that shall be met, such as latency or throughput. A plausible scenario for such a demand would be the presence of several real-time media streams, e.g., one for video and audio each.

Throughput should be straightforward, as the code to measure the amount of sent bytes over a certain timeframe already exists. A periodically called goroutine would query the amount of sent bytes per stream class and reduce a limit for other stream classes if it found a particular class that did not reach its goal.

The goal of measuring latency for a specific stream class is more challenging to achieve. Currently, latency is only measured using the RTT, which indicates how long it takes for a QUIC packet to be sent, acquired and receive the acknowledgment. The RTT samples provided by QUIC are for the whole connection but not at stream-level granularity. See Section 7.2 for more precise latency metrics considerations.

7.1.2 Combination of Metrics

Our modified version of the *Framer* serves as a congestion controller without replacing the actual congestion controller of quic-go. Besides the number of bytes used by the *Framer*, its only other information is the RTT. A combination with other metrics, such as the congestion window or occurring packet loss, is likely to improve the performance further.

7.1.3 Sending Rate Adaption

The rate adaption uses the available metrics to decide on the amount of low-priority data that may be sent.

Hardcoded Values

While we tried to keep the metric derivers agnostic and avoid hardcoded values, this philosophy has not been adapted to the rate adaption algorithm where those metrics are combined. More comprehensive testing might reveal weaknesses in the current implementation that could be tackled with a more flexible adaptation policy.

Growth Phases

For example, this idea of removing hardcoded values can be applied to the growth phases. For example, instead of reducing the $growthfactor$ by a fixed scheme when it is approaching the $lastmax$ value, this could be a dynamic factor dependent on the remaining difference of these two values. Similarly, the *gentle decrease* should be redesigned to decrease slower, reducing the zig-zag patterns observed in Chapter 6 and improving bandwidth utilization.

7.2 One-way delay in QUIC

The RTP one-way delay presented in the evaluation was measured by the application instead of QUIC itself. For a more precise latency measurement, one would need to start at the point in time where the specific piece of data is passed to the underlying QUIC stack. In practice, this means extending the QUIC specification and including a timestamp in the sent packets to later match those packets to a specific stream. Note that such experiments have already been conducted but have not made their way into the QUIC specification yet[Hui22].

7.3 Adaptive Bitrate

This work focused on deriving a congestion control algorithm to simultaneously transfer two different types of data. The quality of the media data was never adapted. Leaning on the idea of SCReAM, one could introduce a mechanism that calculates the allowable bitrate for the real-time data (or the respective class in case of multi-class prioritization). Querying this mechanism periodically, the application would then adapt the quality of the media encoder.

This approach's primary challenge is not determining when to decrease the quality but when to increase it again. When congestion is detected, encoder quality is lowered,

and less data is generated. When more capacity becomes available, the adaption algorithm might allocate more bandwidth to the low-priority data as the metrics for the high-priority data have been stable. To avoid this, an application should, therefore, indicate whether it is satisfied with the current allowed rate or wants to increase the quality of media sent.

7.4 Combination with existing Congestion Control Algorithms

Solutions exist for sending real-time media over various connections and transferring data without increasing delays in SCREaM and LEDBAT, respectively. With both being delay-based, combining their approaches seems promising. As neither is based on QUIC, some work to convert those would be required. Naturally, the application would require an interface to specify how it wishes the respective streams to be treated.

7.5 Proxy for Constrained Environments

With the possibility of implementing separate classes of prioritized streams, the approach could be used to balance multiple clients over a shared connection. For example, a group of machines in a remote location might be sharing internet access via satellite link. If the bandwidth supplied by this connection is subject to change, or the machines stress its' capacity, a proxy tunnel reaching over this satellite link could balance these connections according to preconfigured priorities. Such quality of service measures exist already, however.

7.6 Evaluation

While showing promising results, our implementation can be evaluated more thoroughly.

7.6.1 Performance

The computational complexity has not yet been analyzed. Considering several periodically called goroutines deriving metrics, some even performing linear regressions, we can expect a relatively high computational overhead compared to more straightforward solutions. This overhead could prove to be detrimental for embedded devices or servers dealing with large amounts of simultaneous connections.

7.6.2 Variance in Environment

We only changed the link capacity during our experiment in quite a harsh fashion. A curve with more frequent, more minor changes in capacity could depict actual conditions more accurately.

Further, the algorithm's behavior during packet loss should be examined.

8 Conclusion

In this thesis, we embarked on a novel exploration of data transfer, simultaneous data streaming, and real-time communication. We aimed to maintain a stable, low-latency connection, a challenge that has not been fully addressed in existing research.

Using the quic-go implementation, we introduced a novel component called the StreamBalancer. This component intercepts the frame assembling process, effectively acting as an additional congestion controller. Its role is to dynamically adjust the priority of data streams based on the current network conditions, thereby optimizing the data transfer process. This approach allowed us to derive metrics for the connection at a per-stream granularity. This granularity is crucial as it enables us to better account for the inherent disbalance in availability between a file in memory, which is readily accessible, and media data generated in real-time, which is constantly changing and may not be immediately available for transmission.

While this approach introduced a significant amount of complexity and parallelism, our experiments demonstrated that a simple, or 'naive ', prioritization approach is insufficient. This is primarily due to the aforementioned disbalance in availability, which a simple prioritization scheme may not adequately address, leading to suboptimal performance in terms of connection stability and latency. We believe the identification of this issue to be a key takeaway from this work.

As mentioned, a finer-grained delay measurement requires the QUIC specification. While not altering the protocol, our current solution has demonstrated promising results, providing a practical and effective approach. Thus, it could serve as a starting point for stream prioritization for the QUIC layer and enable prioritization on the transport layer.

Further, we effectively use linear regression in the networking context to swiftly recognize changes in the network environment in an agnostic way.

List of Figures

List of Tables

Bibliography

[Bis22] M. Bishop. *HTTP/3*. RFC 9114. June 2022. DOI: 10.17487/RFC9114.

[Car+16] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo. "Analysis and design of the google congestion control for web real-time communication (WebRTC)." In: *Proceedings of the 7th International Conference on Multimedia Systems*. MMSys '16. Klagenfurt, Austria: Association for Computing Machinery, 2016. ISBN: 9781450342971. DOI: 10.1145/2910017.2910605.

[Edd22] W. Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293.

[Fer+23] F. Fernández, M. Zverev, L. Diez, J. R. Juárez, A. Brunstrom, and R. Agüero. "Flexible Priority-based Stream Schedulers in QUIC." In: *Proceedings of the Int'l ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks*. PE-WASUN '23. Montreal, Quebec, Canada: Association for Computing Machinery, 2023, pp. 91–98. ISBN: 9798400703706. DOI: 10.1145/3616394.3618267.

[Her+24] J. Herbots, R. Marx, M. Wijnants, P. Quax, and W. Lamotte. "HTTP/3's Extensible Prioritization Scheme in the Wild." In: *Proceedings of the 2024 Applied Networking Research Workshop*. ANRW '24. Vancouver, AA, Canada: Association for Computing Machinery, 2024, pp. 1–7. ISBN: 9798400707230. DOI: 10.1145/3673422.3674887.

[Hui22] C. Huitema. *Quic Timestamps For Measuring One-Way Delays*. Tech. rep. Internet-Draft, Intended Status: Experimental, Expires: 1 March 2023. Private Octopus Inc., Aug. 2022.

[IT21] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000.

[JS17] I. Johansson and Z. Sarker. *Self-Clocked Rate Adaptation for Multimedia*. RFC 8298. Dec. 2017. DOI: 10.17487/RFC8298.

[NFB96] H. Nielsen, R. T. Fielding, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. May 1996. DOI: 10.17487/RFC1945.

[Pos80] J. Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768.

[PWO21] C. Perkins, M. Westerlund, and J. Ott. *Media Transport and Use of RTP in WebRTC*. RFC 8834. Jan. 2021. DOI: 10.17487/RFC8834.

[RHW11] C. Raiciu, M. J. Handley, and D. Wischik. *Coupled Congestion Control for Multipath Transport Protocols*. RFC 6356. Oct. 2011. DOI: 10.17487/RFC6356.

[Sch+03] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. July 2003. DOI: 10.17487/RFC3550.

[Sha+12] S. Shalunov, G. Hazel, J. Iyengar, and M. Kühlewind. *Low Extra Delay Background Transport (LEDBAT)*. RFC 6817. Dec. 2012. DOI: 10.17487/RFC6817.

[Ste07] R. R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. Sept. 2007. DOI: 10.17487/RFC4960.

[TB22] M. Thomson and C. Benfield. *HTTP/2*. RFC 9113. June 2022. DOI: 10.17487/RFC9113.

[Wor21] World Wide Web Consortium (W3C). *Web Real-Time Communications (WebRTC) transforms the communications landscape; becomes a World Wide Web Consortium (W3C) Recommendation and multiple Internet Engineering Task Force (IETF) standards*. Published as a W3C Recommendation. Jan. 2021.