UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Visualizing Media over QUIC Traffic: Improving Debugging through Log Analysis and Visualization

Danny Grispen

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Peter QUAX

COPROMOTOR :
Prof. dr. Wim LAMOTTE

BEGELEIDER :
De heer Mike VANDERSANDEN

2024
2025

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Visualizing Media over QUIC Traffic: Improving Debugging through Log Analysis and Visualization

Danny Grispen

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Peter QUAX

COPROMOTOR :
Prof. dr. Wim LAMOTTE

BEGELEIDER :
De heer Mike VANDERSANDEN

Universiteit Hasselt

Masterproef voorgedragen tot het behalen van de graad van master in de informatica

Visualizing Media over QUIC Traffic: Improving Debugging through Log Analysis and Visualization

Auteur:

Danny Grispen

Promotor:

Prof. Dr. Peter Quax

Co-promotor:

Prof. Dr. Wim Lamotte

Begeleider(s):

Mike Vandersanden
Joris Herbots

Academiejaar 2024-2025

Acknowledgments

I would like to express my sincere gratitude to my promotor, Prof. Dr. Peter Quax, and co-promotor, Prof. Dr. Wim Lamotte, for giving me the opportunity to conduct research on this subject. I am also deeply thankful to my mentors, Mike Vandersanden and Joris Herbots, for their invaluable guidance and constructive feedback throughout the development of this thesis. My special thanks go to Luke Curley for generously and thoroughly answering all my questions regarding the operation and implementation of Media over QUIC. Finally, I am grateful to my friends and family for their continuous motivation and support throughout this journey.

Abstract

Media streaming has rapidly become the dominant form of internet traffic, with livestreaming emerging as a particularly demanding use case due to its low latency and large-scale distribution requirements. Existing protocols often fail to provide both scalability and low-latency support, motivating the development of the emerging Media over QUIC (MoQ) protocol. However, debugging complex transport protocols such as QUIC and MoQ presents significant challenges.

This thesis investigates how structured logging and visualization can support the analysis and debugging of MoQ. Building on the qlog logging standard, QUIC logging was extended with MoQ-specific events through a custom logging library. To analyze this data, moq-vis was developed, a visualization application inspired by qvis but tailored to MoQ's characteristics. The application includes adaptations of an existing visualization as well as novel approaches designed to highlight MoQ's scalability and low-latency support.

A series of demonstrations shows how the visualizations reveal inefficiencies in packetization, identify packet loss, expose network topology issues, and evaluate relay functionality. The results confirm that structured logging combined with targeted visualizations provides valuable insights into MoQ's operation, enabling the detection of implementation and deployment issues.

The thesis concludes that structured logging and visualization are effective tools for advancing the analysis and debugging of MoQ. Future work could expand logging and visualization capabilities by adding MoQ-specific events and enhancing or creating visualizations.

Samenvatting

Introductie

Mediastreaming is de afgelopen jaren een dominante vorm van internetverkeer geworden, een trend die door de COVID-19-pandemie is versneld. On-demand streaming en livestreaming vormen de grootste categorieën van dataverkeer in zowel vaste als mobiele netwerken, waarbij livestreaming in recente jaren vooral bij sportevenementen sterk is toegenomen. Lage latentie en hoge schaalbaarheid zijn hierbij cruciale eisen, waar bestaande protocollen vaak niet volledig aan voldoen.

Het Media over QUIC (MoQ) protocol is in ontwikkeling om zowel lage latentie als efficiënte schaalbaarheid te bieden, maar het debuggen van dergelijke complexe protocollen is uitdagend. In dit kader kunnen qlog (gestandaardiseerd loggen van netwerk events) en qvis (visualisatie van QUIC-implementaties) waardevolle ondersteuning bieden. Deze thesis richt zich op het toepassen van qlog-gebaseerde logging en visualisaties om de analyse en debugging van MoQ te verbeteren, met nieuwe MoQ-specifieke logevents en visualisaties die zowel schaalbaarheid als latentie visueel duidelijk maken.

De centrale onderzoeksvragen zijn:

1. Hoe kunnen qlog-gebaseerde logging en visualisatietechnieken de analyse en debugging van MoQ ondersteunen, met specifieke aandacht voor:
   (a) Schaalbaarheid van MoQ
   (b) Lage latentie
   (c) Gebruik van QUIC binnen MoQ
2. Op welke wijze kunnen visualisaties problemen in MoQ-implementaties en -deployments blootleggen, zoals:
   (a) Inefficiënte packetization
   (b) Pakketverlies
   (c) Topologieproblemen

Low-Latency Livestreaming

Low-latency livestreaming vereist het minimaliseren van vertragingen die ontstaan tijdens het volledige mediaproces, van opname tot weergave. Latentie wordt ingedeeld in verschillende categorieën, van hoog (> 45 s) tot near-real-time (< 100 ms). Belangrijk is het onderscheid tussen startup delay (vertraging bij het starten van de stream) en end-to-end (E2E) latentie (vertraging vanaf opname tot weergave).

De end-to-end latentie bestaat uit drie hoofdcategorieën:

- Media Content Preparation Latency (MCPL): vertragingen bij media-acquisitie, -encodering, -packaging en -ingest. Belangrijke factoren bevatten de codec-keuze, segmentlengte, en encryptie.
- Media Content Delivery Latency (MCDL): netwerkvertraging, inclusief CDN-distributie en last-mile levering van de CDN edge naar clients. Optimalisatie vereist onder andere snelle caching.
- Media Content Consumption Latency (MCCL): vertragingen door de mediaplayer zelf, zoals buffering, decoderen, licentieophaling en bitrate-aanpassing. Bij slecht geoptimaliseerde spelers kan dit tot 50% van de totale E2E latentie uitmaken.

Bestaande Protocollen

Er bestaan veel protocollen voor livestreaming, maar elk heeft beperkingen die lage latentie en schaalbaarheid bemoeilijken.

- HLS/DASH: hoge latentie door verschillende factoren zoals head-of-line blocking, schaalt heel goed dankzij HTTP.
- LL-HLS/LL-DASH: lagere latentie, hoge request rate.
- WebRTC: zeer lage latentie, moeilijker om te schalen, vaak kwaliteitsverlies.
- WebRTC Data Channels: flexibiliteit in data, complexe implementatie, zelfde schaalbaarheidsproblemen als WebRTC.
- RTMP: lage latentie, niet langer in gebruik voor mediadistributie door andere problemen zoals beveiliging.

Media over QUIC

MoQ is een protocol in ontwikkeling door de MoQ IETF-werkgroep, bedoeld voor lage-latentie mediaoverdracht zoals livestreaming, online gaming en media-conferencing. Het protocol richt zich niet alleen op datatransport, maar ook op encoderen, packaging en mediaplayer-optimalisatie. Deze thesis legt de nadruk op het loggen en visualiseren van het effectieve transportprotocol.

MoQ heeft een gelaagde structuur, van onder naar boven:

- QUIC: onderliggend transportprotocol.
- WebTransport: zorgt voor browsercompatibiliteit.
- MoQ Transfork: eigenlijke transportprotocol.
- Karp: media playlist en container.
- Applicatie: bijvoorbeeld een livestreaming-app.

QUIC

QUIC biedt voordelen ten opzichte van TCP en WebRTC, zoals onafhankelijke streams om buffering en vertraging te verminderen, connectie IDs om roaming en NAT-rebinding te ondersteunen, en flexibele congestion control voor lage-latentie toepassingen. Het protocol is beveiligd met TLS, inclusief encryptie van headers en bescherming tegen spoofing.

WebTransport

WebTransport is vereist voor browserondersteuning en maakt het mogelijk om meerdere uni- of bidirectionele streams te gebruiken en data out-of-order te ontvangen, waardoor polling, zoals in HTTP, overbodig wordt en browserclients efficiënt live media kunnen ontvangen. Een

nadeel is dat WebTransport sessies een QUIC-verbinding delen met andere sessies en HTTP/3-verkeer.

MoQ Transfork

MoQ Transfork is het transportprotocol bovenop QUIC dat is ontworpen voor live media streaming via CDNs naar een divers publiek met verschillende latentie- en kwaliteitsvereisten. Het protocol is media-agnostisch, zodat tussenliggende nodes zoals relays of CDNs belangrijke data kunnen prioriteren en doorsturen zonder kennis van codecs, containers of encryptie. Het werkt volgens een publish/subscribe-principe waarbij endpoints media publiceren op basis van abonnementen.

De structuur van MoQ Transfork is als volgt:

- Session: connectie tussen client en server, verzendt Tracks aan de hand van hun pad.
- Track: serie van Groups, worden elk onafhankelijk geleverd en gedecodeerd.
- Group: serie van Frames, worden elk in volgorde geleverd en gedecodeerd.
- Frame: payload, representeert een moment in de tijd.

Hoe applicatiedata wordt opgesplitst in Tracks, Groups en Frames is de verantwoordelijkheid van de ontwikkelaar.

Prioritering is cruciaal bij netwerkcongestie: belangrijke media worden eerst geleverd, terwijl minder belangrijke data kan worden "uitgehongerd". MoQ Transfork gebruikt bidirectionele streams voor controle (Session, Announce, Subscribe, Fetch, en Info) en unidirectionele streams voor data (Group).

Het protocol definieert specifieke berichten zoals `SESSION_CLIENT`, `SESSION_SERVER`, `ANNOUNCE`, `SUBSCRIBE`, `GROUP` en `FRAME`, die respectievelijk sessies initiëren, Tracks aankondigen, abonnementen starten, Groups ophalen en data overdragen. Relays en CDNs implementeren alleen MoQ Transfork, niet de medialaag. Het uithongeren van lagere-prioriteits streams ondersteunt verschillende latentiedoelen en netwerkcondities zonder dat streams volledig worden gedropt.

Karp

Karp is de medialaag van het protocol, gebaseerd op de WebCodecs API. Het is geoptimaliseerd voor low-overhead livestreaming en bestaat uit een catalogus (JSON-bestand met metadata over tracks en live updates) en een container (header rond codec-data met timestamp). Kijkers lezen eerst de catalogus en abonneren zich vervolgens op de gewenste Tracks.

Applicatie

De applicatie is de bovenste laag en omvat alle toepassingen van live media. MoQ ondersteunt standaard audio- en videodistributie; andere mediavormen kunnen via aangepaste Tracks worden toegevoegd. Dit biedt de voordelen van MoQ Transfork zonder de implementatie helemaal zelf te moeten doen; enkel het delta-encoderen van de media in Groups en Frames moet worden geregeld.

Logging

qlog

qlog is een uitbreidbaar, gestructureerd loggingformaat voor netwerkprotocollen dat standaardisatie biedt en data-uitwisseling vergemakkelijkt. Het ondersteunt verschillende serialisatieformaten zoals JSON en CSV en is ontworpen om logconsumptie efficiënt en uitbreidbaar te maken. qlog is hiërarchisch opgebouwd:

- Logbestand: bestaat uit metadata over het bestand, zoals een titel en een beschrijving, en een aantal Traces.
- Trace: bevatten events van één datastroom vanaf één vantage point, inclusief een lijst van events en gemeenschappelijke velden.
- Event: bevatten minstens de tijd, naam en data specifiek voor het type event, en kunnen extra velden bevatten zoals group ID en systeeminformatie.

QUIC qlog Definities

De IETF werkt naast het hoofdloggingschema van qlog ook aan qlog-eventdefinities voor verschillende protocollen, waaronder QUIC. De "QUIC event definitions for qlog" draft beschrijft deze events en bijbehorende metadata voor het kernprotocol QUIC en enkele extensies.

In de implementatie worden alleen de `packet_sent` en `packet_received` events gebruikt; de overige 32 gedefinieerde events worden niet behandeld. De 34 QUIC qlog-events zijn verdeeld in vier categorieën:

- Connectivity: 8 events over de verbindingstoestand (start, sluiting, updates van connectie ID).
- Transport: 17 events over datatransport (versturen, ontvangen, drop, acknowledgment van pakketten).
- Security: 2 events over updates van cryptografische sleutels.
- Recovery: 7 events over detectie van pakketverlies en congestion control.

Deze qlog events kunnen zeer gedetailleerde en hiërarchische data bevatten. De `packet_sent` en `packet_received` events bevatten vergelijkbare gegevens, zoals de packet header, ingesloten QUIC frames en ondersteunde QUIC-versies.

Media over QUIC qlog Definities

Deze thesis definieert een eigen set qlog-events voor MoQ Transfork, omdat de officiële MoQ Transport qlog-definities nog niet beschikbaar waren tijdens de implementatie. De events zijn geïnspireerd door QUIC- en HTTP/3-events en richten zich alleen op berichten die tussen MoQ-endpoints worden verzonden. Voor elk berichttype is er een created-event (bij het aanmaken) en een parsed-event (na ontvangst).

De belangrijkste events omvatten onder andere:

- `stream`: nieuwe MoQ stream aanmaken/parsen.
- `session_started_client` / `session_started_server`: starten van een MoQ Session.
- `announce` / `announce_please`: bekendmaking van Tracks.
- `subscription_started` / `subscription_update`: beheer van abonnementen op Tracks.
- `info` / `info_please`: Trackinformatie opvragen.
- `group` / `frame`: verzenden van Group- of Framedata.

Logging Implementatie

De thesis implementeert een Rust qlog-library voor MoQ Transfork en QUIC. De library gebruikt een singleton zodat events via statische functies kunnen worden aangemaakt en gelogd, en ondersteunt Rust features per protocol zodat alleen relevante code wordt gecompileerd. Voor QUIC-events is een caching-mechanisme toegevoegd om events over meerdere functies heen correct samen te stellen. Als serialisatieformaat wordt JSON Text Sequences gebruikt, wat streambaar en eenvoudig te verwerken is.

Voor QUIC is ervoor gekozen alleen de `packet_sent` en `packet_received` events te loggen. Deze events bevatten alle verzonden data en geven inzicht in de interactie tussen MoQ en QUIC. Direct loggen bij verzending of ontvangst was niet mogelijk vanwege encryptie en de manier waarop Quinn pakketten opbouwt, wat verspreid is over meerdere functies en bestanden. Om dit op te lossen is caching toegevoegd. Events kunnen worden bijgewerkt naarmate frames aan het pakket worden toegevoegd en worden uiteindelijk gelogd bij verzending of na decryptie.

Om events van dezelfde MoQ Session te koppelen, werd een tracing ID toegevoegd aan het `SESSION_CLIENT`-bericht, zodat beide endpoints een gedeelde waarde hebben. Daarnaast is een extra veld `main_role` toegevoegd aan de Trace, dat aangeeft wat de hoofdrol van elk endpoint is, zoals publisher, subscriber of relay. Door de structuur van de logging library waren er verder weinig aanpassingen nodig. Het resultaat is dat de logs van zowel QUIC als MoQ worden samengevoegd in één bestand per endpoint.

Deze loggingstructuur maakt het mogelijk gedetailleerde qlog-bestanden te produceren, die vervolgens in visualisatietools kunnen worden geanalyseerd.

Visualisaties

qvis

qvis is een toolsuite voor het visualiseren van QUIC en HTTP/3 die qlog-bestanden kan importeren. De toolsuite bevat meerdere pagina's met verschillende visualisaties: een sequentiediagram toont het verkeer tussen client en server met `packet_sent` en `packet_received` events als pijlen. Een congestion pagina visualiseert verzonden, verloren en bevestigde data en geeft informatie over flow control, congestion window en RTT, terwijl een multiplexingpagina laat zien hoe gegevens van meerdere streams over tijd worden verzonden. De packetizationpagina illustreert welke frames en streamgegevens in elk pakket zijn opgenomen, en een statistiekentabel geeft een overzicht van metrieken zoals event-aantallen en frame-types. Veel visualisaties zijn interactief en bieden extra details bij hover. qvis stelt ontwikkelaars in staat om zowel algemene hypotheses als diepgaandere analyses te maken, en de brede adoptie in de QUIC-community laat zien dat qlog en qvis effectief zijn voor debugging en het verbeteren van implementaties.

Visualisatie Implementatie

moq-vis is een webgebaseerde applicatie ontwikkeld voor het visualiseren van qlog-bestanden van meerdere endpoints, gericht op het analyseren van MoQ's schaalbaarheid en low-latency ondersteuning. De visualisaties omvatten een netwerkgraaf voor het overzicht van endpoints en connecties, latentiegrafieken om latentie te analyseren en een sequentiediagram voor endpoint- en protocolinteracties. Alle visualisaties zijn interactief.

De netwerkgraaf biedt een overzicht van endpoints en hun verbindingen, gebaseerd op groep ID's uit de qlog Traces. Nodes worden geïdentificeerd met bestandsnamen en visueel onderscheiden door iconen op basis van hun rol (publisher, subscriber, pubsub, relay). Verbindingen tussen nodes worden weergegeven als interactieve edges die geselecteerd kunnen worden om specifieke connecties te analyseren in andere visualisaties. De grafiek ondersteunt drag, zoom en pan, waardoor zowel kleine als grote netwerken overzichtelijk blijven.

Het sequentiediagram in moq-vis toont chronologisch events binnen geselecteerde verbindingen. Events zijn verbonden om datastromen zichtbaar te maken, en alleen geselecteerde verbindingen worden weergegeven om het diagram leesbaar te houden. Events worden langs verticale assen per endpoint geplaatst. Om overlap tussen MoQ- en QUIC-events te voorkomen, worden MoQ-berichten als semi-transparante blokken weergegeven, terwijl QUIC-events erboven blijven. Alle events zijn interactief met modal windows voor details. Kleuren geven type aan, en toggle knoppen boven het diagram laten toe om QUIC- of MoQ-events aan- of uit te zetten.

De latentiegrafieken geven inzicht in de latentie van QUIC-verbindingen en MoQ Sessions,

belangrijk voor low-latency livestreaming. Voor elke geselecteerde verbinding wordt een grafiek gemaak waarbij de x-as de tijd sinds het begin van de verbinding toont en de y-as de latentie in milliseconden. De grafieken zijn interactief: in- en uitzoomen en pannen is mogelijk, en tooltips bij hover tonen exacte waarden van de geplotte data. Twee toggles laten afzonderlijk QUIC- en MoQ-data aan- of uitzetten.

Evaluatie

moq-vis is getest met drie demo's:

- Eenvoudige topologie: één publisher, één relay, één subscriber.
- Zelfde topologie, maar met packet loss.
- Complexere topologie: twee publishers, drie subscribers en twee relays.

Bij de eerste demo viel op dat MoQ vaak meerdere kleine QUIC-pakketten verzendt in plaats van data te bundelen. Zo worden verschillende MoQ-berichten apart verpakt, terwijl een andere connectie dezelfde data in één pakket verstuurt. Dit gedrag veroorzaakt extra protocol-overhead door de headers van UDP en QUIC. De oorzaak hiervan is nog onduidelijk, mogelijk ligt het aan de Quinn-implementatie of aan hoe MoQ met Quinn samenwerkt. Het sequentiediagram blijkt echter een effectief hulpmiddel om deze inefficiënties te identificeren.

Bij de demo met packet loss wordt duidelijk hoe QUIC verloren pakketten opnieuw verzendt, terwijl MoQ tijdelijk hogere latentie ervaart. Dit veroorzaakt opvallende latentie spikes voor MoQ, terwijl de QUIC-latentie relatief stabiel blijft. Ook bij abrupt beëindigen van verbindingen gaan de laatste Frames soms verloren, waardoor meerdere cycli van probe- en retransmit-pakketten volgen. De visualisatie maakt dit gedrag zichtbaar en benadrukt het effect van packet loss op de vertraging van MoQ-berichten.

De netwerkgraaf biedt een snelle visuele weergave van de netwerkstructuur, wat handig is om de werkelijke topologie te vergelijken met de verwachte. Zo kan een niet-verbonden node in een subnet direct worden opgemerkt, waardoor gericht debuggen mogelijk wordt. Momenteel toont de netwerkgraaf alleen verbindingen tussen nodes en geen individuele events van een node in het sequentiediagram. Ook onderscheidt de graaf geen meerdere connecties tussen dezelfde twee nodes: alle verbindingen worden samengevoegd tot één edge. Dit kan later verbeterd worden.

Bij het inspecteren van het sequentiediagram tussen twee relays in de derde demo bleek dat events van drie verschillende MoQ Sessions (drie QUIC-connecties) overlappen in tijd. De huidige implementatie houdt geen rekening met meerdere gelijktijdige sessies, waardoor sommige blokken slecht worden weergegeven en QUIC-events onder MoQ-events verdwijnen. Verbeteringen in de rendering-logica van het sequentiediagram zijn nodig om dit probleem op te lossen.

Conclusie

Ten slotte kunnen we concluderen dat qlog-gebaseerde logging en visualisaties effectief zijn voor het analyseren en debuggen van MoQ-verkeer. De netwerkgraaf, latentiegrafieken en sequentiediagram bieden samen zowel een overzicht van de topologie als gedetailleerde inzichten in berichtenstromen, latentie en protocolgedrag. Visualisaties maken inefficiënte packetization, pakketverlies en topologieproblemen zichtbaar. Hoewel verbeteringen mogelijk zijn tonen de resultaten aan dat qlog-gebaseerde technieken een waardevol hulpmiddel vormen voor het begrijpen, analyseren en verbeteren van MoQ-implementaties en -deployments.

Contents

Chapter 1

Introduction

Media streaming has become a dominant form of internet traffic in recent years. The COVID-19 pandemic accelerated this trend: in March 2020, global streaming hours increased by almost 21% in just two weeks [Conviva, 2020].

The 2024 Global Internet Phenomena Report (GIPR) highlights the scale of this shift. On-demand streaming accounts for 54% of downstream traffic in fixed networks, such as Fiber to the Home (FTTH), while livestreaming contributes 14%. Together, they represent the two largest categories of downstream data. Similar figures are observed in mobile networks such as 4G, where on-demand streaming constitutes 57% and livestreaming 7% of traffic, ranking first and third, respectively. In fixed networks, most traffic originates from platforms such as YouTube and Netflix, while in mobile networks, short-form content from social media platforms like Facebook and TikTok dominates [AppLogic Networks, 2024].

The 2025 GIPR reports an even stronger trend toward livestreaming, particularly for sporting events. Major streaming platforms, including Amazon Prime, Peacock, and Netflix, began broadcasting exclusive sports content in 2024, and investments in livestreaming have continued across the industry. This trend is expected to continue. In Q4 2024, the ten highest-traffic days in the US all coincided with livestreamed sporting events, with traffic spikes of 30-40% observed during broadcasts [AppLogic Networks, 2025]. These findings clarify that streaming is central to today's internet usage, with livestreaming becoming more important.

Low latency is a critical factor for livestreaming. Sports fans demand to see the action as close to real time as possible, and interactive livestreams, such as Twitch streams, depend on fast responses for better viewer interactions. Scalability is equally important, particularly during major events attracting millions of viewers. During the live broadcast of the Mike Tyson vs. Jake Paul boxing match on Netflix, watched by 60 million households, viewers reported freezing, buffering, and resolution issues [West, 2024]. These complaints were also visible in the traffic of X, formerly Twitter, which surged when viewers encountered the streaming issues during the event [AppLogic Networks, 2025].

Existing livestreaming protocols often fall short, struggling to combine low latency with efficient scalability or lacking standardization. This led to the creation of a new protocol, named Media over QUIC (MoQ) [Curley, 2025a]. MoQ, still under development, aims to provide both low latency and efficient scalability for livestreaming of media by building directly on QUIC [Iyengar and Thomson, 2021] as its transport protocol. Creating a new protocol is challenging because network protocols, particularly complex ones like QUIC, are difficult to debug.

This is where qlog [Marx et al., 2025a] and qvis [Marx, 2024] become relevant. qlog is an emerging standard that defines a structured logging schema for network protocols, while qvis is a visualization toolsuite developed to aid in debugging QUIC and HTTP/3 implementations. By logging standardized events and representing them visually, qlog and qvis have already been

used to identify and fix issues across multiple QUIC stacks. Their success demonstrates the potential of structured logging and visualization for debugging complex protocols.

This thesis aims to improve MoQ debugging using structured logging and visualizations inspired by qlog and qvis. It introduces qlog-based logging for QUIC and MoQ by using predefined QUIC events [Marx et al., 2025b] and defining new MoQ-specific events. These logs are then used to create visualizations. Initial inspiration for the visualizations came from qvis. One of the visualizations is also present in qvis but has been adapted to fit MoQ better. The others are novel visualizations introduced to highlight MoQ's scalability and low-latency support.

The research questions this thesis seeks to answer are:

1. How can qlog-based logging and visualization techniques support the analysis and debugging of MoQ?
   (a) How can MoQ's scalability be effectively represented in visualizations?
   (b) How can MoQ's low-latency support be captured and illustrated?
   (c) How can MoQ's use of QUIC be visualized to highlight protocol behavior?
2. In what ways can visualizations reveal issues in MoQ implementations and deployments?
   (a) How can inefficient packetization be identified and analyzed through visualizations?
   (b) How can packet loss be seen in the visualizations?
   (c) How can topology issues be identified?

The structure of this thesis is as follows. Chapter 2 introduces the foundations of low-latency livestreaming, explains relevant concepts, reviews existing protocols, and describes MoQ. Chapter 3 discusses the upcoming qlog standard and its integration with QUIC and MoQ using a custom library. Chapter 4 discusses qvis and presents moq-vis, the implementation of custom MoQ visualizations. Chapter 5 evaluates moq-vis through a series of demos. Finally, Chapter 6 summarizes the findings and outlines directions for future work.

Chapter 2

Low-Latency Livestreaming Foundations

Achieving low-latency livestreaming is a complex challenge, as latency can arise from many sources. This chapter lays the groundwork for understanding latency and livestreaming protocols. It begins by introducing key concepts related to streaming and latency, followed by a detailed breakdown of end-to-end (E2E) latency. Next, the capabilities of existing protocols to attain low latency and efficient scalability are examined. Finally, the inner workings of MoQ are described in detail.

2.1 Streaming and Latency Concepts

The term 'low-latency livestreaming' will frequently appear throughout this thesis; therefore, it is important to establish a clear and precise understanding of what it exactly entails. This section will begin by briefly discussing what livestreaming precisely means. It will then examine the notion of latency, discuss the various latency thresholds, and outline what is commonly regarded as 'low latency.' Finally, the section will provide an overview of the full E2E latency, identifying the various delay sources contributing to overall streaming latency.

[Bentaleb et al., 2025] define 'streaming' as "the continuous transmission of media such as video, audio and metadata from a server to a client and its simultaneous consumption by the client". An important word in this definition is "simultaneous"; if the media were downloaded entirely before playing it, it would not be considered streaming.
Livestreaming is the streaming of live content, linear and personal broadcasts, and surveillance videos. Live content refers to events happening in real time, such as live sporting events and live news. Linear broadcasts include television programs consisting of live as well as prerecorded content. TV channels might stream live sports and/or prerecorded movies or shows, but this prerecorded content is still streamed in real time, so it falls under livestreaming. Personal broadcasts refer to content captured live from personal devices, like smartphones and PCs. Surveillance videos include live video from devices such as security cameras [Bentaleb et al., 2025].

Latency is the amount of time between the capture of media and its playback. Latency can range from a handful of milliseconds to a minute or more. The following subdivision of latency, suggested by [Bentaleb et al., 2025], shows five categories, ranging from the highest to the lowest amount of latency:

1. High latency: 45 or more seconds. Such high latency can be obtained when the scalability and robustness of the streaming service are more important.

2. Typical latency: between 10 and 45 seconds. The latency of most services using HTTP adaptive streaming, which is discussed in Section 2.3.1, lies in this range.
3. Low latency: between one and 10 seconds. Broadcast television's latency falls within this range. This level of latency is required to have a similar user experience to broadcast TV.
4. Ultra-low latency: between 100 milliseconds and a second. Interactive experiences need ultra-low latency, including live commentary, online gambling, and auctioning.
5. Near-real-time latency: less than 100 milliseconds. Videoconferencing, cloud gaming, and remote control of hardware are examples of applications requiring near-real-time latency.

Other sources are similar but are stricter on low latency, categorizing it between one and five seconds of latency [Zenoni, 2025, OptiView, 2018].

An important distinction is the difference between startup delay and latency, two independent streaming metrics. Startup delay is defined as "the time lag from when the client joins a live stream (i.e., the viewer presses PLAY) until it begins to play out the media.". At the same time, latency is "the time lag from when a media frame is captured until the same frame is played out" [Bentaleb et al., 2025]. The startup delay only affects the beginning of a media player's playback, while the latency is a persistent element throughout the playback (although it is possible for latency to further increase or decrease during playback) [Bentaleb et al., 2025].

2.2 End-to-End Latency

Latency is not just the network delay; there is more to it than meets the eye. The complete E2E latency has to be considered. The definition given at the beginning of the previous paragraph is that of E2E latency. The whole media delivery process, everything between media capture and playback, must be considered. Every part contributes to the E2E latency, some more than others. There are three main categories, defined by [Bentaleb et al., 2025], into which the E2E latency can be subdivided:

1. Media Content Preparation Latency
2. Media Content Delivery Latency
3. Media Content Consumption Latency

These will be discussed in more detail in the rest of this section. Figure 2.1 shows these categories and the latency sources that form them.

2.2.1 Media Content Preparation Latency

The Media Content Preparation Latency (MCPL) consists of multiple latency sources relating to acquiring and preparing media for distribution. [Bentaleb et al., 2025] subdivides the MCPL into the following latencies:

1. Acquisition latency: usually negligibly small, will not be discussed further.
2. Encoding latency: contribution depends on the media type; in the case of video encoding, this is often the most significant contributor to the MCPL due to video encoding's complexity. Other factors influencing encoding latency include codec choice, the content's complexity, target quality, and encoding parameters.
3. Packaging latency: involves encapsulating the encoded media into one or more delivery formats. A factor that could potentially increase this latency further is the encryption of the encoded content.
4. Ingest latency: latency of uploading the packaged media and potentially extra necessary data (depending on the chosen protocol) to an origin server for distribution.

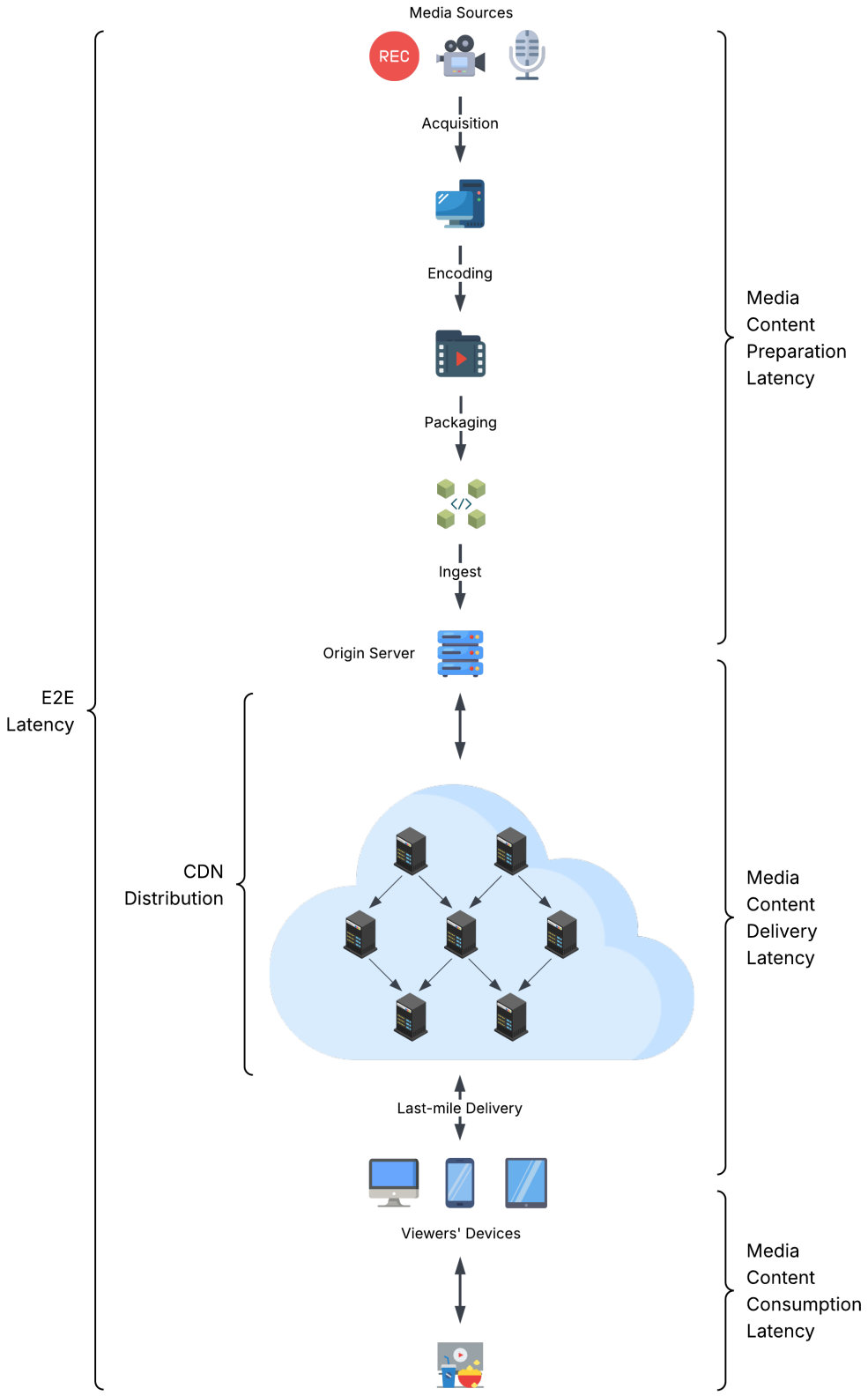


Figure 2.1: Visual representation of the latency sources that contribute to the E2E latency (icons: Flaticon.com)

The following paragraphs will detail these latency sources (apart from the acquisition latency).

Encoding Latency

To improve streaming performance, every codec should have a high compression ratio, which reduces bandwidth and storage, high visual quality, low complexity, and fast processing. Apart from codec selection, suitable parameters must be chosen for the encoding, including the bitrate ladder, and Group of Pictures (GoP) and segment size [Bentaleb et al., 2025].

The bitrate ladder refers to the different bitrate-resolution pairs in which the video is encoded and can be streamed. Table 2.1 displays an example bitrate ladder. The amount and exact choices of bitrate-resolution pairs may increase encoding complexity and, in turn, increase latency. On the other hand, multiple well-chosen bitrate-resolution pairs might lead to better bitrate adaptation in streaming clients, leading to a better user experience (e.g., streaming at a lower resolution on a small phone screen, or being able to switch to a lower bitrate in bad network conditions). These trade-offs make bitrate ladder selection an important but non-trivial task [Bentaleb et al., 2025].

Resolution	Bitrate
416x234	145 kb/s
640x360	365 kb/s
768x432	730 kb/s
768x432	1100 kb/s
960x540	2000 kb/s
1280x720	4500 kb/s
1920x1080	6000 kb/s
1920x1080	7800 kb/s

Table 2.1: Bitrate ladder example [Apple, 2018]

Video encoding works by encoding all the video frames. Some frames are encoded to be independent of other frames, called keyframes. A video decoder can use keyframes as a starting point. Other frames depend on these keyframes but are smaller in size. GoP duration refers to the interval between keyframes.
Large videos are divided into multiple smaller videos that are several seconds long and called segments. In order for rate adaptation to work well so that switching to a different bitrate-resolution pair happens as unnoticeably as possible, every segment needs to begin with a keyframe. This means that every segment consists of at least one GoP.
A low GoP duration means more keyframes, which reduces encoding efficiency and increases the amount of data to be downloaded. A small segment size also means a low GoP duration, this leads to the consequences already discussed. Lengthy segment sizes lead to fewer intervals at which rate adaptation can happen, which increases the likelihood of buffering when network conditions worsen. High GoP durations also mean longer segment sizes, including the ensuing consequences. Again, these parameters must be considered carefully, and trade-offs must be made [Bentaleb et al., 2025].

Packaging Latency

Packaging is the process of encapsulating the media into various delivery formats. Since not all devices and browsers support the same protocols, support for multiple formats is necessary to reach most of the audience. Packaging the content into multiple delivery formats can add a significant amount of latency. In case of a much more widely used format than others, a mitigation can be used that initially packages the content only into the popular format and, when necessary, repackages this media into another format at the network edge. This way, there

is minimal packaging latency. Content Delivery Networks (CDNs) benefit from this since they need to deliver just a single format (better caching and bandwidth utilization might reduce latency further) [Bentaleb et al., 2025].

Another solution would be to combine delivery formats. This is done by the Common Media Application Format (CMAF) standard as an attempt to converge and simplify delivery formats [Bentaleb et al., 2025].

Another step in the packaging process is encryption for content protected by Digital Rights Management (DRM) and/or encryption to protect against fingerprinting or tampering with the data [Bentaleb et al., 2025].

Ingest Latency

Ingest is the transmission of the packaged data to an origin server so that the content is ready to be retrieved. The resulting latency of this step is dependent on the protocol being utilized. Protocols such as WebRTC prioritize low latency, while other protocols, like SRT, tend to have higher latency but better media quality [Bentaleb et al., 2025].

2.2.2 Media Content Delivery Latency

The Media Content Delivery Latency (MCDL) is exclusively comprised of network latency, as it represents the portion of total latency attributable to the transmission of media content from the origin server to the client. [Bentaleb et al., 2025] decompose the MCDL further into the following components:

1. CDN distribution latency: propagation of media through the CDN.
2. Last-mile delivery latency: media propagation from the CDN edge to client devices.

CDN Distribution Latency

CDNs are necessary to handle requests coming from a large number of media players. Several factors can reduce the CDN distribution latency. First, opt for a CDN provider that provides server locations close to clients and supports the necessary technologies for low-latency transport. The CDN provider should also serve live content from memory instead of disk since reading from memory is noticeably faster, and minimize cache misses. In order to do this, effective caching policies and prefetching decisions need to be developed. Caching policies refer to the ruleset applied by CDNs to store data. At the same time, prefetching decisions are the techniques CDNs use to fetch data from the origin server before a client requests that data in order to be prepared and boost cache hits. In the context of low-latency livestreaming, this is incredibly complex due to the time restraints [Bentaleb et al., 2025].

CDNs also need to be able to handle 'flash crowds.' These are defined as "sudden and significant increases in the number of media players" by [Bentaleb et al., 2025]. Flash crowds occur commonly when many clients start watching a livestream at the same time. CDNs should be designed to handle these sudden surges in clients while minimizing the load on the origin server and keeping latency as low as possible [Bentaleb et al., 2025].

Last-Mile Delivery Latency

The last bit of network latency comes from delivering the media from the CDN edge via an access network to clients' devices. These access networks include institution, home, and cellular networks provided by local Internet Service Providers (ISPs) [Bentaleb et al., 2025].

2.2.3 Media Content Consumption Latency

The Media Content Consumption Latency (MCCL) is the product of the media player's tasks. Media players want to ensure smooth playback; they do this by buffering multiple seconds of

media before starting playback. In the context of low-latency livestreaming, this is an undesirable amount of added latency. Thus, adapting the media player for low-latency livestreaming is essential. Otherwise, the media player could be responsible for up to 50% of the E2E latency [Bentaleb et al., 2025].

The media player's functionality is not as simple as downloading the media, decoding it, and rendering the video. Other tasks include rate adaptation, buffering, playhead positioning, license/key fetching, decryption, and decoder priming. Fetching the decryption keys from the licensing server and priming the video and audio decoders are examples of two tasks that need to be done before decoding and rendering can even start. These steps could add even more latency. Furthermore, other modules such as bandwidth measurement, playback speed control, buffer management, and bitrate selection schemes should be redesigned for low-latency livestreaming in order to improve latency further [Bentaleb et al., 2025].

2.3 Existing Protocols

Numerous existing protocols allow livestreaming of data; this raises the question of why these are not used. However, each protocol exhibits one or more limitations that render it suboptimal for achieving low-latency livestreaming with efficient scalability. In this section, we will examine several of these protocols and analyze their respective shortcomings. Non-standardized protocols will not be discussed here. Figure 2.2 visualizes the latency of a number of protocols, including the ones discussed in this section. Table 2.2 at the end of this section provides an overview of the discussed protocols.

Identifying where the latency sources of these protocols contribute to E2E latency is often challenging. Some latency sources vary depending on how the protocol is configured and used, rather than being inherent to the protocol itself. For example, encoding latency is influenced by factors such as the selected bitrate ladder, GoP size, and segment size, which are typically configurable rather than fixed.

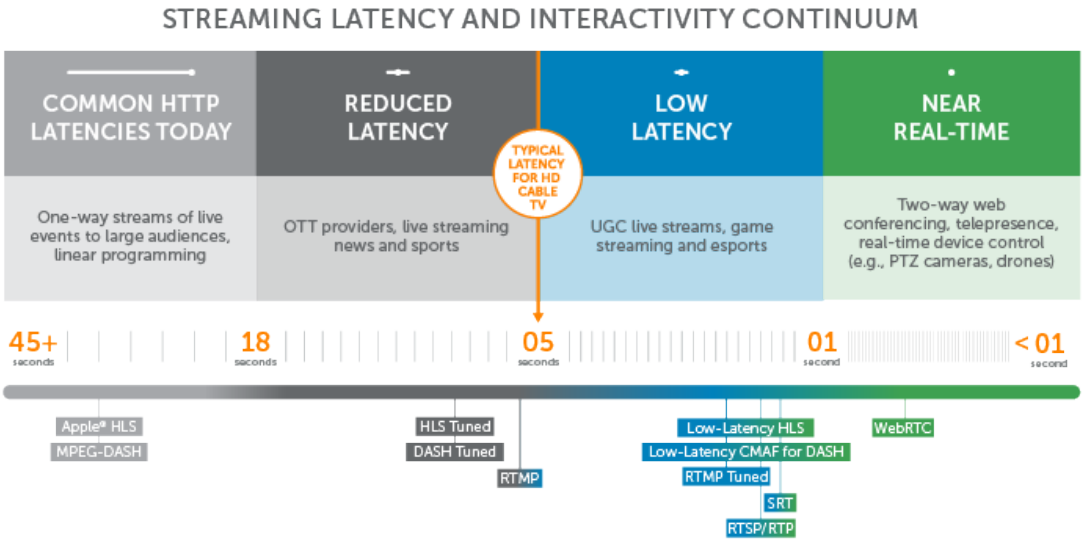


Figure 2.2: Latency of numerous streaming protocols [Zenoni, 2025]

2.3.1 HLS/DASH

HLS (HTTP Live Streaming) and DASH (Dynamic Adaptive Streaming over HTTP) are two different HAS (HTTP Adaptive Streaming) protocols often put together as 'HLS/DASH' be-

cause of their resemblance. They are also the only two remaining HAS protocols in use; others have become obsolete [Bentaleb et al., 2025]. These protocols work by splitting up large video files into smaller videos that are several seconds long and serving these smaller videos over HTTP.
This study will talk more specifically about HLS since this is a publicly available RFC, while DASH is an ISO standard behind a paywall. HLS defines media playlists as text files containing URIs to media segments, the smaller video files that, when played sequentially, will form the larger video. Also defined are master playlists, which include a set of variant streams, which are different versions of the same content, such as versions with different bitrates, different video formats, or different resolutions. Master playlists can also specify a set of renditions or alternate versions, like audio in multiple languages or recordings from various camera angles [Pantos and May, 2017]. A simple schematic representation can be seen in Figure 2.3.
The HLS client will first download the playlist file (using HTTP). Listing 2.1 shows an example of an HLS master playlist file with URIs to different media playlists based on bandwidth. After that, the client will determine which variant to start playing (if there are variants) and which rendition (again, if these exist) based on certain conditions (like network status) and personal preferences (e.g., English audio). The protocol supports encryption, livestreaming, and dynamic adaptation. Livestreaming is supported by adding new media segments to the media playlists; this does require frequent redownloads of the playlist file. Dynamic adaptation is supported by switching to another variant (if available) during playback (e.g., your network conditions worsen, so the client dynamically adjusts by changing to a variant with a lower bitrate) [Pantos and May, 2017].

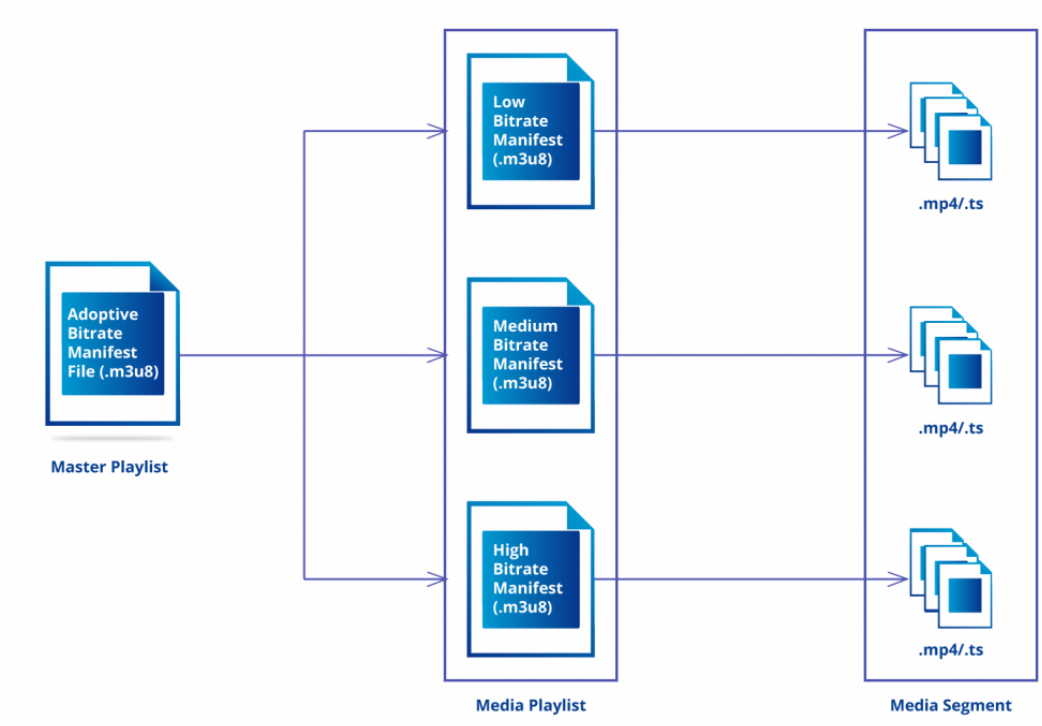


Figure 2.3: Schematic representation of a simple HLS master playlist [Synchronous, 2021]

```
#EXTM3U
#EXT-X-STREAM-INF:BANDWIDTH=1280000,AVERAGE-BANDWIDTH=1000000
http://example.com/low.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=2560000,AVERAGE-BANDWIDTH=2000000
http://example.com/mid.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=7680000,AVERAGE-BANDWIDTH=6000000
http://example.com/hi.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=65000,CODECS="mp4a.40.5"
http://example.com/audio-only.m3u8
```

Listing 2.1: Example of an HLS master playlist file which has multiple variants with different bitrates [Pantos and May, 2017]

HLS/DASH works perfectly for VODs (Video On Demand: pre-existing video like Netflix movies or YouTube videos), but using it for livestreaming raises some issues. The biggest issue is latency. HLS/DASH has many sources of latency. For example, segments must be fully completed before being added to the playlist; the player must poll for playlist updates rather than receiving them directly from the server; and segments are downloaded sequentially instead of in parallel. All the latency sources accumulate to a total latency of around five seconds (in Twitch's case, after optimizations) [Luke Curley, 2022].
One of the most significant latency factors is HTTP's transport protocol: TCP. TCP delivers data reliably and in order, which can lead to head-of-line blocking. During network congestion, the available throughput may fall below the media bitrate. Because the sender continues to transmit at this higher rate, the total bitrate passing through the router exceeds its capacity, which causes packets to build up in queues and ultimately results in buffering. The HLS/DASH player has built-in Adaptive Bit Rate (ABR) algorithms, which can detect when the throughput gets too low or buffering happens, and switch to a lower bitrate. The problem is that this is only possible after the current segment has finished downloading (segments are multiple seconds long); this might take a while with the worsened network conditions [Luke Curley, 2023b].
The latest HTTP version, HTTP/3, replaces TCP with QUIC, which runs on top of UDP. However, it is not as simple as just replacing older HTTP versions with the newest one; this approach will not yield far greater results. Each HLS/DASH segment request will still be sent sequentially. More adjustments will have to be made [Luke Curley, 2023b].

HLS/DASH achieves excellent scalability by leveraging the existing HTTP infrastructure. This ecosystem already includes widespread client and server support, as well as optimized CDNs [Luke Curley, 2023b]. HTTP itself scales efficiently because it is stateless, uses well-established addressing and authentication/authorization schemes, easily traverses middleboxes, and benefits from low-cost caching infrastructure [Bentaleb et al., 2025].

2.3.2 LL-HLS/LL-DASH

LL-HLS and LL-DASH are low-latency variants of their respective protocols. They both try to achieve lower latency by dividing a media segment into smaller chunks and delivering it as soon as the first chunk is available. The client can then begin rendering the segment when it receives the first chunk. Subsequently, more chunks are sent to the client as they are produced [Zhang et al., 2021].
This section will discuss LL-HLS in more depth. LL-HLS still uses its playlist files, which are frequently updated with new partial segments (chunks) as the stream goes on. Partial segments are listed separately in the playlist file, and clients send separate requests for each partial segment. Playlist files are requested only once—at the beginning of the connection. Afterward, the client will receive updates, which are much smaller in size, instead of the whole playlist file(s). These are not the only added features, but the most notable ones [Bentaleb et al., 2025].

The partial segments help reduce latency immensely compared to regular HLS. Latency can be reduced from the length of a segment (in HLS) to the length of a partial segment (in LL-HLS),

from multiple seconds to a few hundred milliseconds [Zhang et al., 2021]. Figure 2.4 shows how these partial segments affect LL-HLS. Traditional HLS has the choice to start playing Segment B when joining the livestream to minimize the startup delay, or to wait for Segment C to finish to minimize the latency. The same applies to LL-HLS, but the usage of the partial segments significantly reduces the latency in the first scenario, and the startup delay in the second scenario, compared to HLS.
However, the partial segments are still served over HTTP, so TCP is still the underlying transport protocol. All the drawbacks of using TCP that were discussed in Section 2.3.1 also apply to the low-latency variant. Lower latency is achievable with LL-HLS/LL-DASH, but head-of-line blocking and video buffering will still happen during congestion [Luke Curley, 2023b].

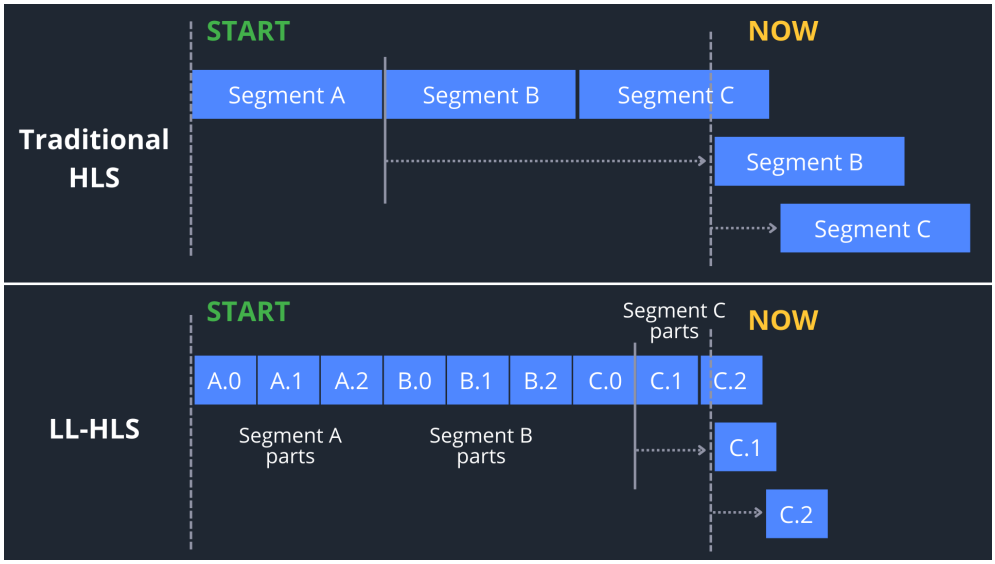


Figure 2.4: Difference between HLS and LL-HLS regarding segments [Karthikeyan, 2022]

Like HLS, LL-HLS can utilize the existing HTTP infrastructure to scale efficiently. However, the approach introduces additional overhead compared to traditional HLS. To achieve low latency, clients must request each partial segment separately. The shorter the length of these partial segments, the more requests must be sent. On top of this, clients will also need to request playlist updates frequently. LL-HLS thus has a much higher request rate than regular HLS. Livestreaming events with many viewers, and thus many concurrent clients with high request rates, can cause significant strain on the CDN and network service providers [Durak et al., 2020].

2.3.3 WebRTC

WebRTC is not a standalone protocol but rather a protocol suite or framework that leverages various protocols under the hood. It provides a set of APIs designed to facilitate the real-time transfer of media and generic application data between devices, using real-time communication protocols [Jennings et al., 2024, Perkins et al., 2021, Alvestrand, 2021]. Figure 2.5 shows an overview of the protocols employed by the WebRTC media stack. Secure Real-time Transport Protocol (SRTP) is used to transport the actual real-time media between the clients, while Datagram Transport Layer Security (DTLS) ensures that the real-time media is encrypted. Interactive Connectivity Establishment (ICE), Session Traversal Utilities for NAT (STUN), and Traversal Using Relays around NAT (TURN) are needed to establish a peer-to-peer connection between WebRTC clients [Alvestrand, 2021]. Lastly, all this communication happens (preferably) on top of UDP since it's faster, and latency is a big concern with real-time communication. TCP can be used in case UDP is unavailable or restricted [MDN web docs, 2025c].

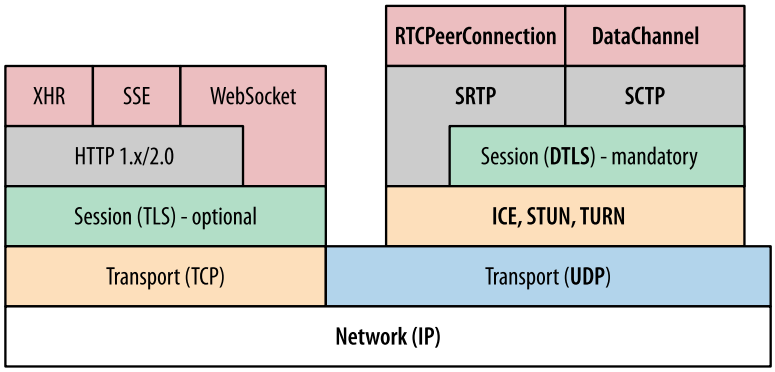


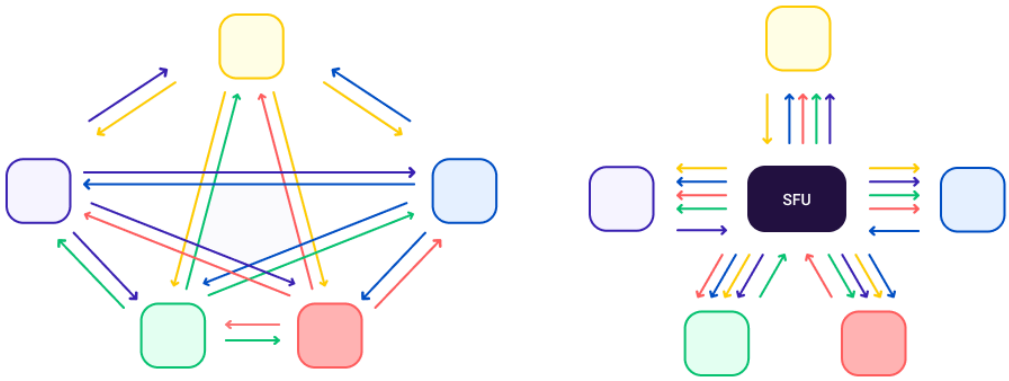
Figure 2.5: WebRTC protocol stack, the part under `RTCPeerConnection` is the media stack [Grigorik, 2013]

WebRTC is primarily designed for video conferencing, with a strong emphasis on minimizing latency. However, this focus often results in significant reductions in media quality. This behavior works very well for its intended purpose, but the hard-coded quality compromise is generally not something most livestreaming media applications desire. Consequently, this behavior can lead to a suboptimal user experience in such scenarios. Adapting WebRTC to suit specific use cases better is challenging, as its configurable modes are limited and may not meet the requirements of all applications [Luke Curley, 2023c].

WebRTC was designed with a peer-to-peer architecture in mind. As a result, each sending peer must encode independent streams for all receiving peers. This approach requires a dedicated encoder for each stream, leading to significant resource consumption. Consequently, the architecture does not scale effectively to accommodate a large number of participants, as is shown in Figure 2.6a [Petrangeli et al., 2018, Petrangeli et al., 2019].
Scalability can be improved through the use of Selective Forwarding Units (SFUs). These components receive streams from all peers and determine which streams should be forwarded to which peers, this is shown in Figure 2.6b. The logic governing these forwarding decisions is left to the application developer using WebRTC [Petrangeli et al., 2018, Petrangeli et al., 2019]. However, this custom functionality also introduces challenges for scalability. Determining where to forward the streams is a complex task. Additionally, a single SFU cannot effectively handle a large number of users. To address this, multiple SFUs must be deployed, each requiring knowledge of the network topology and the locations of all participants. This is another non-trivial problem that adds further complexity to the system [Luke Curley, 2023c], Figure 2.7 shows an example of how such a structure might look.

2.3.4 WebRTC Data Channels

WebRTC data channels work slightly differently than the WebRTC media stack discussed in the previous section (Section 2.3.3). As shown in Figure 2.5, data channels use SCTP instead of SRTP. SCTP (Stream Control Transmission Protocol) is a reliable transport protocol designed to operate on top of connectionless packet networks, like IP. Some of its features include user data transfer with acknowledgments and without duplication, the possibility of multiple streams, and optional bundling of multiple user messages into a single SCTP packet. The protocol lies somewhere between TCP and UDP since it was developed because TCP was considered too limiting, and SCTP has more built-in features than UDP [Stewart et al., 2022].
WebRTC does not run SCTP directly over IP but runs SCTP over DTLS over UDP. SCTP over DTLS provides confidentiality, source authentication, and transfers with integrity protection. Additionally, DTLS over UDP and ICE enables middlebox traversal. Data channels consist of one incoming SCTP stream and one outgoing SCTP stream; this allows for bidirectional communication. Furthermore, data channels have several properties in each direction; the most important ones are whether to use reliable or unreliable transport, whether message delivery is



(a) Direct peer communication **(b)** Peer communication with SFU

Figure 2.6: WebRTC peer communication [Voximplant, 2020]

in-order or out-of-order, and a priority [Jesup et al., 2021].

Data channels are able to improve upon the video quality of the media stack since they allow any data to be sent. They are not bound to the same aggressive quality reduction. However, the logic governing the sending and receiving of data is the application developer's responsibility since the data channel API does not provide this functionality [MDN web docs, 2025a]. The resulting latency is partly dependent on the usage of the data channels.
[Diaz et al., 2024] have attempted to implement a low-latency livestreaming solution using WebRTC data channels. They use unreliable SCTP streams (unordered, without retransmissions) and try to fit one MP4 fragment in each SCTP message (if a fragment is too large to fit, it gets split up) with some metadata. The receiver checks the validity of the MP4 fragment and pushes it to the decoder if an accompanying keyframe is available; otherwise, the receiver waits for the next keyframe whilst dropping other messages. Caching fragments and waiting for past keyframes would have led to an increase in latency. Their paper shows some pros and cons of this approach. Their solution can take advantage of pre-existing WebRTC and media playback APIs already available in the browser. However, SCTP's congestion control makes data channels susceptible to latency if the receive buffer is not large enough for the desired throughput. In order to achieve high throughput and low transmission delay, very low network latency or a large enough receive buffer is required (which is browser-dependent) [Diaz et al., 2024].
[Luke Curley, 2023c] also attempted to use data channels. He tried sending each video frame as an unreliable message. Due to fundamental flaws with SCTP, this did not end up working. Breaking the frames into multiple unreliable messages below the MTU size did work, but this was not an ideal solution. The most important reasons are that every packet gets acknowledged, even though it is unreliable, and a custom SCTP implementation is required, which is unavailable in browsers.

Regarding scalability, data channels suffer from the same issues as the WebRTC media stack since the same underlying peer-to-peer architecture also applies to data channels; this was discussed in Section 2.3.3.

2.3.5 RTMP

RTMP (Real-Time Messaging Protocol) is a protocol developed by Adobe that provides bidirectional message service over a reliable transport protocol, such as TCP. RTMP supports parallel streams of video, audio, and data messages between peers. The protocol also offers the ability to assign priorities to different message classes, which might influence the order in which messages are sent to the underlying transport protocol. RTMP defines multiple types of messages, including data, audio, video, and command messages. Data messages are used

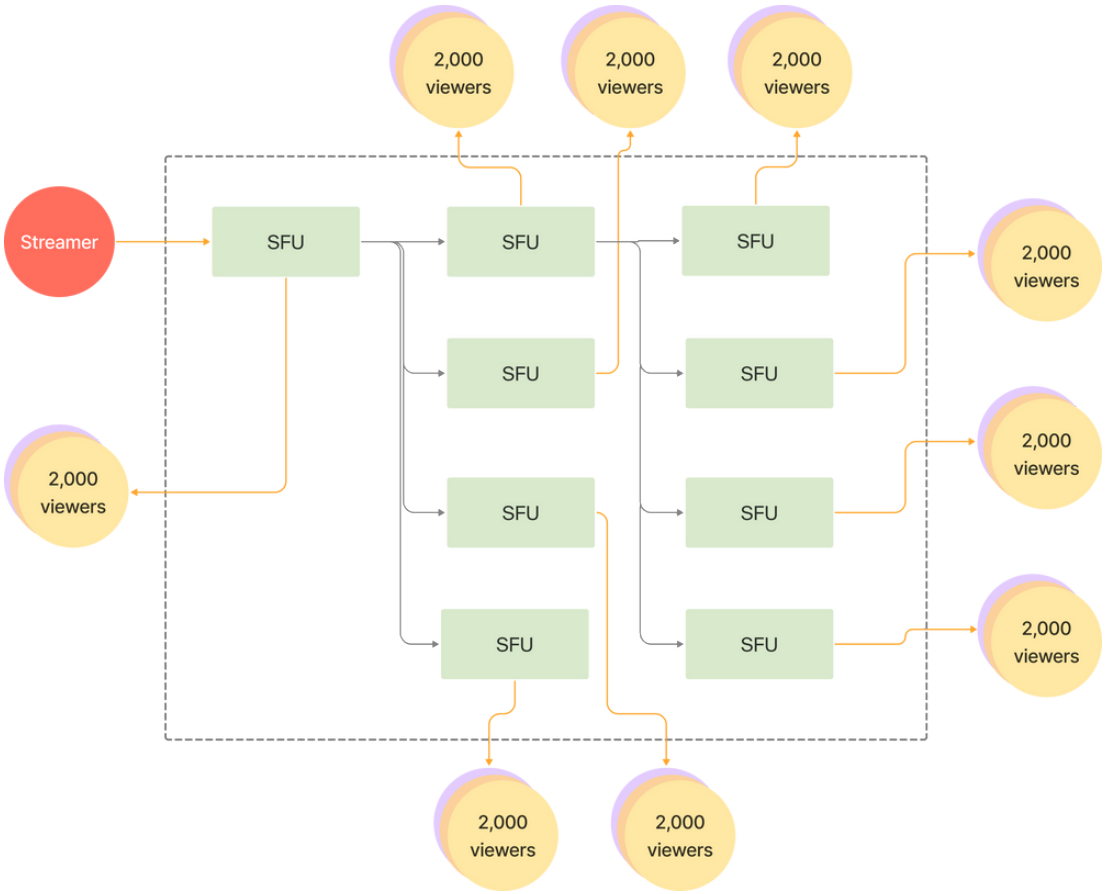


Figure 2.7: WebRTC multiple SFUs example [Zhao, 2022]

to send metadata or user data, and audio and video messages carry audio and video data, respectively. Command messages perform operations when sent to the peer; these operations affect the connection itself or an open stream. Commands on the connection include '`connect`' (requests connection to a server application), '`call`' (executes remote procedure calls at the receiver), and '`createStream`' (creates a stream to publish audio, video, and metadata on). Stream commands include '`play`', '`pause`', and '`publish`' (allows clients to publish a named stream to the server, which any client can play) [Zenoni, 2025].

Since the first two letters of RTMP stand for Real-Time, the protocol is expected to have low latency. This is indeed the case, although its latency of around five seconds is not as low as other real-time protocols, such as WebRTC [Zenoni, 2025]. As with HLS/DASH and LL-HLS/LL-DASH, TCP is the underlying transport protocol, which means this protocol also suffers from the same head-of-line blocking problem already discussed in Section 2.3.1. Then again, these buffering problems are limited since RTMP divides streams into small chunks [Yuzzit, 2025].

RTMP suffers from some other issues. Many endpoints no longer support playing media over RTMP since Flash was proprietary, vulnerable to cyber attacks, and deprecated in modern browsers [Bentaleb et al., 2025]. Furthermore, the protocol has issues passing through firewalls and is not ideal for adaptive bitrate streaming [Yoss, 2022].
However, it is still used for ingest (transport of video from encoder to streaming server) because of the protocol's simplicity, compatibility, and low latency [Zenoni, 2025]. Consequently, RTMP's scalability is no longer an important concern for the protocol since its only use now is as an ingest protocol. The streaming server transcodes the data into another protocol, like HLS or WebRTC, for distribution to clients [Yoss, 2022].

	Latency	Scalability	Use Cases
HLS/DASH	− Has to wait for full segments to complete − Polling: cannot get directly from server − Segments are downloaded sequentially − Head-of-line blocking can cause buffering − Bitrate changes can only happen after finishing a segment	+ Utilizes existing HTTP infrastructure + HTTP scales well	• VOD streaming • Livestreaming (when latency is not a concern) [Digital Samba, 2024]
LL-HLS/LL-DASH	+ Only has to wait for partial segments to complete − Head-of-line blocking can cause buffering	+ Utilizes existing HTTP infrastructure + HTTP scales well − Higher request rate than HLS/DASH	• Low-latency livestreaming
WebRTC	+ Near real-time latency − Aggressive latency minimization causes poor media quality − Limited configuration available	− Peer-to-peer: does not scale well − Scaling using SFUs requires complex custom functionality	• Video conferencing • Low latency streaming • File sharing • Internet of Things [Aboba, 2023]
WebRTC Data Channels	+ Allows any data to be sent, does not have the same media quality reduction − Requires custom sending and receiving logic (which partly affects latency) − Receive buffer size (browser-dependent) can influence latency (depending on desired throughput) − SCTP has fundamental flaws	− Peer-to-peer: does not scale well − Scaling using SFUs requires complex custom functionality	• Video conferencing • Low latency streaming • File sharing • Internet of Things [Aboba, 2023]
RTMP	+ Low latency − Head-of-line blocking can cause buffering	− No longer used for media distribution	• Livestreaming (less since Flash deprecation) • Ingesting streams [Zenoni, 2025]

Table 2.2: Overview of the discussed protocols: latency, efficient scalability, and (a subset of) use cases with associated pros and cons

2.4 Media over QUIC

The Media over QUIC (MoQ) protocol, as proposed by the MoQ IETF working group, aims to enable low-latency media delivery with efficient scalability. It is intended to support a range of use cases, including livestreaming, online gaming, and media conferencing. At the time of writing, MoQ remains under active development, with draft versions of the standard being published periodically. The final specification is anticipated to be completed in the coming years [Internet Engineering Task Force, 2022].

MoQ is not just the data transfer protocol, as discussed earlier in Section 2.1; the whole media delivery process influences the latency. Steps such as encoding and packaging, and components like the media player, are also important to optimize for low latency. The data transfer protocol is just a single part, albeit important. However, since the remainder of this thesis focuses on logging and visualizing the actual transport protocol, this part will be discussed in more detail than the others.

2.4.1 Working Group Goals

The MoQ working group describes MoQ as "a simple low-latency media delivery solution for ingest and distribution of media." It is meant to be efficiently scalable and supported in both browsers and non-browser endpoints, and use cases include livestreaming, gaming, and media conferencing. The focus lies on building protocol mechanisms for media publication and ways to identify and receive the media. This media will be mapped to underlying QUIC mechanisms (streams and/or datagrams) to be sent over the network, hence the name 'Media over QUIC' [Internet Engineering Task Force, 2022].

The goals for the common protocol for publishing media for ingest and distribution to support, as defined by [Internet Engineering Task Force, 2022], are:

- at least one media format,
- an interoperable way to indicate what media, and which format it is in, is being sent,
- strategies for rate adaptation and
- cache-friendly media mechanisms.

The goals for the mechanism that names and receives media to enable are:

- requests for the server to send media related to a given point in the stream and
- the option to select the desired encoding (such as language and bitrate).

Another goal is to specify a simple method for authentication to relays or servers so clients can transmit or receive media. The media will be encrypted at the transport layer through the encryption mechanisms built into QUIC. End-to-end encryption will be possible in certain use cases, but the keys will not be available to relays, only to media sources and consumers. However, the relays can access the necessary metadata for actions such as caching and making forwarding decisions. That metadata will be authenticated and end-to-end integrity-protected [Internet Engineering Task Force, 2022].

2.4.2 Protocol Architecture

MoQ consists of multiple layers that each have their own function. The layering here is important for the protocol to be flexible [Luke Curley, 2024b]. An important thing to note is that two layers have multiple versions, which are shown in the enumeration below. The rest of this thesis and the implementation will not use the IETF version, but will focus on another version. This decision stems from the fact that the IETF working group often engages in extensive debates over protocol details, which can take considerable time to resolve. Moreover, the group tends to prioritize the transport protocol over the media layer [Luke Curley, 2024a]. As a result, it

is more interesting to develop logging and visualization tools for an alternative version of the protocol that evolves more rapidly in order to provide insightful feedback and highlight issues with the protocol so that it can be improved. These findings could then be shared with the IETF working group to inform and potentially influence the development of the official specification [Curley, 2025a]. The rest of this thesis will go into more detail about Luke Curley's fork of MoQ. Ordered from the bottom to the top, as explained by [Luke Curley, 2024b], these layers are:

1. QUIC: underlying transport protocol.
2. WebTransport: needed for browser compatibility.
3. MoQ Transport/MoQ Transfork: actual transport protocol.
4. Warp/Karp: media playlist and container.
5. Application: application that uses MoQ for low-latency livestreaming.

The following paragraphs will detail these layers further.

QUIC

As discussed earlier in Section 2.3.1, TCP is a reliable and ordered transport protocol, which can cause head-of-line blocking and, subsequently, buffering of media. Head-of-line blocking is a problem that has been known for years, and multiple protocols have attempted to fix this problem, including HTTP. QUIC has a solution involving shared state and independent streams. The shared state includes encryption, congestion control, and flow control; this is shared for the entire connection. The independent streams have their own state on top of the shared state and can be created, delivered, and closed in parallel; these operations have minimal overhead [Luke Curley, 2024b].

Another useful QUIC feature is the connection IDs. TCP uses the 4-tuple (source IP, source port, destination IP, destination port) to identify connections so incoming packets can be forwarded correctly to the proper socket/connection. TCP's approach causes some problems nowadays. When roaming and thus switching networks, the source IP and port combination will change. This also causes TCP's 4-tuple to change, meaning the server cannot identify the connection anymore. The server will discard packets since the source is unknown, and the connection will end. The application on the client side needs to detect the severed connection and retry, creating a new connection and resending all lost requests. Applications could proactively reconnect when detecting a switch in the network instead of waiting for the connection to timeout. However, this is not possible with NAT rebinding. NAT can rebind the address transparently. If this happens, the application will not know about it, and the connection will appear to hang [Luke Curley, 2023a].
WebRTC uses the 2-tuple (destination IP, destination port), which means the source IP and source port can change without ending the connection to the WebRTC server. However, this raises another issue. Each connection on the server requires opening a unique port. The problem is that corporate firewalls block almost every port for security reasons, severely limiting the number of connections that can be opened [Luke Curley, 2023a].
UDP is a connectionless protocol that simply sends the data to the specified destination without having to maintain any connection state [Kurose and Ross, 2016].
In order to avoid having the same problems TCP and WebRTC have, QUIC employs connection IDs. Connection IDs are data blobs chosen by the receiver and are sent in the header of all QUIC packets. Multiple connection IDs can be used, and they can be issued and retired at will. In the case of roaming, the source IP and port will change, but the connection ID will not. This change will lead QUIC to validate the new path, which is done to prevent spoofing attacks, before switching to that path. QUIC itself does all of this; that way, the application using QUIC does not need to do anything, unlike TCP applications, which need to develop the measures discussed earlier. The same happens in the case of NAT rebinding. The cost of this is only one RTT, which is needed to verify the new path. QUIC uses the same port for all

connections, port 443 (UDP). Since HTTPS runs on this port, it is one of the few ports often allowed to be open. QUIC clients can even use the same IP and port for all of their outgoing connections [Luke Curley, 2023a].

A secondary usage of connection IDs is in load balancing. Since the receiver chooses the connection IDs, which have unbounded length, any information can be encoded in them. This makes it possible to do a multitude of interesting things with them; there is even a draft dedicated to these possibilities [Duke et al., 2025]. However, attention must be paid to overhead and security. Attaching a connection ID of a kilobyte might not be the best idea, and since the connection ID is plaintext data, caution must be taken so it is encrypted or unguessable [Luke Curley, 2023a].

Furthermore, QUIC takes privacy and security very seriously. Encryption via TLS is required, and even the packet headers are encrypted, so middleboxes cannot inspect or modify packets. The server can sign connection IDs; this way, the client is unable to tamper with them or spoof them. Hence, cooperating routers are able to detect abusive packets and drop them before they reach the server. This action is stateless and can be done in hardware efficiently [Luke Curley, 2023a].

Another advantage of QUIC is its congestion control. This is modeled after TCP but has minor important differences, such as the unique packet numbers in all QUIC packets, even for retransmissions. The most significant difference between TCP and QUIC congestion control is that TCP is implemented in the kernel. This means that the TCP implementation is difficult or impossible to modify. In most (or even all) cases, the default TCP congestion control algorithm is loss-based, which suffers from bufferbloat. These loss-based congestion control algorithms work poorly for low-latency applications. QUIC is not implemented in the kernel, which makes it possible to change between congestion control algorithms; it is even possible to write and use a custom algorithm. This allows existing algorithms to be adapted more effectively to specific use cases [Luke Curley, 2023a].

The choice of building a new protocol on top of QUIC instead of UDP comes from QUIC's features, including those discussed in the previous paragraphs, QUIC's availability in the browser, and QUIC's many implementations. MoQ wants to gain the most out of QUIC's features for delivering live media [Luke Curley, 2024b].

WebTransport

WebTransport is a browser API that can be seen as an updated version of WebSockets. The WebSockets API allows two-way communication between client and server so that polling is not required to receive a response from the server. WebTransport supports multiple streams, unidirectional streams, and out-of-order delivery. Reliable transport is achieved by using streams, while unreliable transport happens via UDP-like datagrams [MDN web docs, 2025d, MDN web docs, 2025b]. WebTransport essentially exposes QUIC in the browser, which means HTTP does not need to be used for browser support. The advantage of this is that the server can send data whenever it is ready, instead of the client constantly having to poll for new data. The content being live media worsens the HTTP problem since the client first needs to know what to request, which is difficult when the content does not exist at that point. Using HTTP also raises the issue of having to handle the contribution and distribution of media separately, since contribution goes from client to server, while distribution goes from server to client. Having QUIC available in the browser eliminates these issues. However, WebTransport is not a perfect solution; one disadvantage is that the WebTransport session shares a QUIC connection with other WebTransport sessions and HTTP/3 requests [Luke Curley, 2024b].

MoQ Transfork

MoQ Transfork is the actual transport protocol built on top of QUIC (MoQ Transport in the IETF version). This layer is central to the thesis since most of the logging happens here.

For that reason, MoQ Transfork will be discussed in great detail in the following paragraphs. However, given the rapid evolution of the protocol, the following explanation may not fully represent the current implementation.

MoQ Transfork is a transport protocol designed to facilitate the delivery of live media streams over CDNs to a diverse audience with varying latency and quality requirements, encompassing the full spectrum from real-time interaction to VOD scenarios. The protocol is media-agnostic, enabling intermediate nodes, such as relays and CDNs, to prioritize and forward critical data even under degraded network conditions, without requiring awareness of the underlying codecs, container formats, or the encryption status of the content. Its flexibility allows it to serve as a foundational layer for higher-level protocols, which define the specific mechanisms for encoding and transmitting various forms of live content, including video, audio, and messaging [Curley, 2025a]. The protocol makes use of the publish/subscribe workflow: endpoints producing media publish this data in response to subscriptions from one or more endpoints [Nandakumar et al., 2025].

MoQ Transfork consists of the following:

- Session: connection between client and server, transmits Tracks by their path.
- Track: a series of Groups, each independently delivered and decoded.
- Group: a series of Frames, each delivered and decoded in order.
- Frame: payload, represents a single moment in time.

Figure 2.8 shows a visual representation of this hierarchical structure. The exact way in which data is split into tracks, groups, and frames is the responsibility of the application developer. In order to provide robust and generic one-to-many data transmission, MoQ Transfork, in turn, manages the caching and delivery of this data by applying rules specified within headers. This architecture supports generic one-to-many distribution while maintaining robustness even in latency-sensitive use cases [Curley, 2025a].

Sessions consist of a QUIC connection between a client and a server and are established after the MoQ Transfork handshake. Sessions are intended to be chained together using relays. This way, broadcasters can establish sessions with the CDN ingest edge, while viewers can establish their own sessions with CDN distribution edges. While MoQ Transfork sessions operate hop-by-hop, application developers should design the applications using MoQ to be end-to-end [Curley, 2025a].

A Track is defined as a sequence of Groups, each uniquely identified by a path. A single MoQ Transfork session can support the publication and subscription of multiple, potentially unrelated, tracks. Track paths are composed of a series of string segments used to route subscriptions to the appropriate publisher. The application is responsible for defining the structure and encoding of these paths. Within a given session, each Track path must be unique [Curley, 2025a].
The publisher may advertise track availability through an `ANNOUNCE` message, enabling subscribers to discover relevant tracks dynamically based on a prefix. Subscribers also define the order and priority of their subscriptions, providing hints to the publisher about which Tracks should be delivered first in the event of network congestion. This prioritization mechanism is essential for maintaining an acceptable quality of experience under degraded network conditions. It is the core reason that QUIC is able to support low-latency, real-time communication [Curley, 2025a].

A Group constitutes an ordered sequence of Frames within a Track. Each Group is transmitted over a dedicated QUIC stream, which may be closed upon completion, reset by the publisher, or cancelled by the subscriber. During an active subscription, multiple Groups may be delivered concurrently, and the Frames within these groups are delivered reliably and in the correct order due to QUIC streams. However, Groups themselves may arrive out of order or may not arrive at all, and the application must be capable of handling such scenarios [Curley, 2025a].

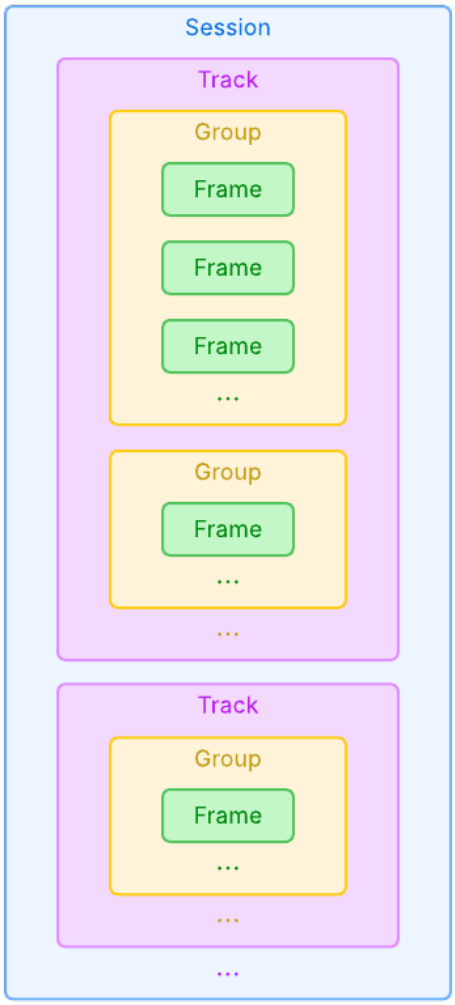


Figure 2.8: Visual representation of the hierarchical structure of MoQ Transfork

Subscribers can issue `FETCH` requests for specific Groups starting at a specified byte offset. This functionality, similar to an HTTP request, can be employed to do things such as recover from partial delivery failures [Curley, 2025a].

A Frame is a data payload within a Group; it represents a data chunk of a known size. Frames should represent a single moment in time and avoid buffering so as not to increase the latency [Curley, 2025a].

Media protocols can only be classified as "live" if they maintain functionality under conditions of degraded network congestion. MoQ Transfork addresses this requirement by prioritizing the most important media, while less important media are starved [Curley, 2025a].
The subscriber signals the relative importance of individual Tracks, Groups, and Frames; the publisher attempts to obey these signals to the extent possible. Subscribers convey priority through two mechanisms: Track Priority and Group Order. Either endpoint may drop any starved data so as not to block the livestream [Curley, 2025a].
In scenarios where a publisher, such as a relay node, serves multiple concurrent sessions, prioritization should be handled per session. For instance, one subscriber may want low-latency livestreaming with a preference for audio, whereas another may prioritize reliable playback while

having muted audio. In such cases, relays may propagate subscriber preferences upstream. However, in the event of conflicts, the publisher's preferences should serve as a tiebreaker [Curley, 2025a].

MoQ Transfork runs on top of WebTransport, so creating a connection involves the QUIC, WebTransport, and MoQ Transfork handshakes. The MoQ Transfork handshake involves opening a Session Stream and sending a `SESSION_CLIENT` message to the server, after which the server replies with a `SESSION_SERVER` message. These messages contain version negotiation and extension negotiation. This session remains active until either endpoint decides to close or reset the Session Stream [Curley, 2025a].
All MoQ Transfork streams are QUIC streams, but contrary to QUIC, MoQ Transfork does not allow half-open bidirectional streams. If one endpoint terminates the send direction of a stream, the peer must also terminate their send direction of that stream. Stream termination can happen in two ways: by closing the send direction gracefully (`STREAM` frame with the `FIN` bit set), or by resetting the send direction (`RESET_STREAM` frame), which immediately terminates the stream [Curley, 2025a].

MoQ Transfork makes extensive use of QUIC's parallel streams feature. Every transactional action gets a dedicated stream. Transactions are terminated if their respective streams are closed. Each stream starts with a `STREAM_TYPE` (1 byte to indicate the stream type) [Curley, 2025a].
The following streams are bidirectional; these are primarily control streams:

- Session: created by the client.
- Announce: created by the subscriber.
- Subscribe: created by the subscriber.
- Fetch: created by the subscriber.
- Info: created by the subscriber.

The only unidirectional stream used for data transmission is the Group Stream, which is created by the publisher [Curley, 2025a].

Session Streams are designated for all session-level messages within a MoQ Transfork connection. Only a single Session stream is opened per WebTransport session, and closing this stream closes the MoQ Transfork session. Upon establishing a QUIC/WebTransport session, the client is required to open a single Session Stream immediately [Curley, 2025a].
At the start of the Session Stream, the client transmits a `SESSION_CLIENT` message, which communicates the supported protocol versions and extensions. If at least one version is supported, the server responds with a `SESSION_SERVER` message, thereby completing the handshake process. Following this initial exchange, both endpoints are expected to send `SESSION_UPDATE` messages when appropriate, for instance, to signal substantial changes in session characteristics such as bitrate [Curley, 2025a].

Subscribers may optionally initiate an Announce Stream to discover tracks whose identifiers match a specified prefix. This mechanism is not mandatory; applications may alternatively determine track paths through out-of-band methods [Curley, 2025a].
The stream is started by the subscriber sending an `ANNOUNCE_PLEASE` message. In response, the publisher transmits `ANNOUNCE` messages for each matching track. Each `ANNOUNCE` message contains one of these three statuses:

- `active`: a matching track is currently available.
- `ended`: a previously `active` track has become unavailable.
- `live`: all currently active tracks have been sent.

Tracks initially appear with an `ended` status and may transition between `active` and `ended` over time. The `live` status applies to the entire stream rather than to individual tracks.

The publisher should send this once all `active` `ANNOUNCE` messages have been delivered. The subscriber should interpret the `live` status as a signal that all the current announcements have been received [Curley, 2025a].
Prefix matching is performed part by part. For instance, a prefix of `["meeting"]` would match a track path such as `["meeting", "1234"]`, but not `["meeting-1234"]`. It is the application's responsibility to encode prefixes unambiguously, such as avoiding that the same value can be encoded in multiple ways [Curley, 2025a].
Multiple Announce Streams may be used concurrently, potentially with overlapping prefixes. Each stream receives its own copy of each `ANNOUNCE` message in such cases [Curley, 2025a].

To request a specific Track, a subscriber opens a Subscribe Stream. This process begins with the subscriber opening an Info Stream and sending a `SUBSCRIBE` message, which may be followed by any number of `SUBSCRIBE_UPDATE` messages. The publisher must respond with an `INFO` message, followed by zero or more `SUBSCRIBE_GAP` messages. If the publisher is unable to serve the subscription at any point, it may reset the stream [Curley, 2025a].
The publisher must deliver a complete Group Stream or a corresponding `SUBSCRIBE_GAP` message for each Group within the specified subscription range. Consequently, the publisher must send a `SUBSCRIBE_GAP` message if a Group Stream is reset. The subscriber should close the subscription once all of the `GROUP` and `SUBSCRIBE_GAP` messages have been received. Conversely, the publisher may close the subscription after the subscriber has successfully acknowledged all the messages [Curley, 2025a].

Fetch Streams can be opened by the subscriber in order to receive a single Group starting at a specified offset. This mechanism is primarily intended for recovery scenarios, such as when a stream is abruptly terminated, resulting in the truncation of a Group [Curley, 2025a].
Subscribers start a Fetch Stream with a `FETCH` message, followed by zero or more `FETCH_UPDATE` messages. In response, the publisher must transmit the contents of the corresponding Group Stream, beginning at the specified offset after the `GROUP` message. This transmission includes any subsequent `FRAME` messages. The Fetch Stream remains active until it is explicitly closed by both endpoints or reset by either endpoint [Curley, 2025a].

Subscribers may open an Info Stream to request information about a specific Track explicitly. However, this is typically unnecessary, as such information is provided in response to a `SUBSCRIBE` request. Subscribers must send an `INFO_PLEASE` message at the start of the Info Stream. The publisher is then required to either respond with an `INFO` message or reset the stream. Both endpoints must subsequently close the stream [Curley, 2025a].

In response to a Subscribe Stream, the publisher generates Group Streams. Each Group Stream must begin with a `GROUP` message and may be followed by zero or more `FRAME` messages. A Group may contain no `FRAME` messages, which could indicate a gap in the associated track. Similarly, individual `FRAME` messages may carry empty payloads, potentially signaling a gap within the Group itself. Both the publisher and the subscriber may reset the Group Stream at any point. In the event that a Group Stream is reset, the publisher must send a corresponding `SUBSCRIBE_GAP` message on the related Subscribe Stream [Curley, 2025a].

MoQ Transfork transmits the following messages, defined in [Curley, 2025a], between endpoints:

- `STREAM_TYPE`: short header to indicate the type of the stream.
- `SESSION_CLIENT`: sent by the client to initiate the session, contains supported versions and extensions.
- `SESSION_SERVER`: response of the server to `SESSION_CLIENT`, contains selected version and extensions.
- `SESSION_UPDATE`: used to send the estimated bitrate of the QUIC connection.
- `ANNOUNCE`: sent by the publisher to advertise a Track.

- `ANNOUNCE_PLEASE`: sent by the subscriber to request `ANNOUNCE` messages matching the given Track prefix.
- `SUBSCRIBE`: sent by the subscriber to start a subscription to a Track.
- `SUBSCRIBE_UPDATE`: sent by the subscriber to modify a subscription.
- `SUBSCRIBE_GAP`: sent by the publisher to indicate it is not able to serve a Group for a given `SUBSCRIBE`.
- `INFO`: sent by the publisher in response to `SUBSCRIBE` or `INFO_PLEASE`, contains information about the given Track.
- `INFO_PLEASE`: sent by the subscriber to request an `INFO` message.
- `FETCH`: sent by the subscriber to request a (partial) Group from a given Track.
- `FETCH_UPDATE`: sent by the subscriber to modify a `FETCH` request.
- `GROUP`: sent by the publisher, includes information about the given Group, belonging to a certain subscription.
- `FRAME`: sent by the publisher, contains a payload at a specific point in time.

Generic relays and CDNs should only implement MoQ Transfork, not the media layer. The MoQ Transfork headers provide sufficient information to facilitate optimal caching and fan-out, even under network congestion. The payload remains opaque to the transport layer, allowing applications to encode arbitrary content, including encrypted media or non-media data [Luke Curley, 2024b].
A crucial concept in MoQ Transfork is the mapping of live media to MoQ Transfork. A critical consideration when mapping is how to handle congestion. In certain scenarios, such as conference calls, pausing and buffering are undesirable, which means some data needs to be dropped. Regarding video, the decision was made to skip frames until the next keyframe, in other words, until the start of the next GoP (which was briefly discussed in Section 2.2.1). For this reason, GoPs are mapped to Group Streams in MoQ Transfork. The underlying QUIC stream provides reliability, ordering, and prioritization. This enables fine-grained control over packet transmission under constrained network conditions, ensuring that the most important groups, such as recent audio, are delivered first. In contrast, less important media, like older video, will be starved or potentially cancelled [Luke Curley, 2024b].
This prioritization method is also compatible with varying latency targets. MoQ Transfork does not drop streams; instead, it starves lower-priority streams. Viewers who tolerate higher latency, such as those watching VODs, use larger playback buffers. This way, there is more time for network starvation recovery and, thus, recovery for the starved lower-priority groups. QUIC streams are only canceled if the user decides to skip forward in the playback [Luke Curley, 2024b].

Karp

Karp is the media layer of the protocol (Warp in the IETF version) and is modeled after the WebCodecs API [Luke Curley, 2024b]. The WebCodecs API is a browser API that allows developers to gain low-level access to individual video stream frames and audio chunks. This enables developers to have complete control over media processing [MDN web docs, 2024]. Karp contains only the necessary metadata to initialize a decoder and render a frame, and is optimized for low-overhead livestreaming. It consists of a catalog and a container [Luke Curley, 2024b].

The catalog is JSON data that describes the media tracks, an example of which is shown in Listing 2.2. The catalog file is similar to the playlist files in HLS. It is delivered over a MoQ Transport track, which is commonly named `catalog.json`. This JSON blob contains all the available broadcast tracks and their metadata, including audio, video, and alternative renditions of the media. The catalog itself is a live track, so that it can be updated with live media. Like

with other tracks, viewers subscribe to the catalog and receive updates to this file via groups or frames. As a result, tracks can be added or removed live. When joining the livestream, viewers first subscribe to the catalog, parse the catalog to determine the desired tracks, and subscribe to the chosen individual media tracks [Luke Curley, 2024b].

```
{
    "video": [{
        "track": {
            "name": "480p",
            "priority": 2
        },
        "codec": "avc1.64001f",
        "resolution": {
            "width": 1280,
            "height": 720
        },
        "bitrate": 3000000
        // etc
    }],
}
```

Listing 2.2: Example of a Karp catalog file; it shows information for MoQ Transport, the used codec, the video resolution, and the maximum bitrate in bits per second [Luke Curley, 2024b]

The container is a simple header around codec data. The header of the current version only contains a 1-8 byte presentation timestamp. This will be updated in the future to support more functionality, such as information surrounding encryption. However, future additions will be kept simple to minimize the overhead [Luke Curley, 2024b].

Application

The final layer is the application; this could be anything involving live media. MoQ is able to provide audio and video delivery; other media delivery requires custom implementation. This can be done with a custom track, such as a track for delivering chat messages between people. This approach gains all the advantages of MoQ Transport without having to implement it from scratch; the only remaining task is to delta-encode the desired media into groups and frames [Luke Curley, 2024b].

2.4.3 Addressing Known Problems

To conclude this chapter, this section will examine known problems with the protocols discussed in Section 2.3 and how MoQ addresses these.

Head-of-line blocking is a problem in all the discussed protocols that rely on TCP as their underlying transport protocol. However, the severity of its impact varies across protocols. QUIC avoids head-of-line blocking when using multiple streams, since QUIC streams are ordered, reliable, and independent. In the event of packet loss, only streams containing data in the lost packet are blocked and have to wait for a retransmission. Meanwhile, other streams can continue without interruption. However, if a single QUIC packet carries data from multiple streams, all those streams will be blocked when packet loss occurs. Therefore, implementations are advised to minimize the number of streams per packet while maintaining transmission efficiency (avoiding underfilled packets) [Iyengar and Thomson, 2021].
Since MoQ maps Group Streams (thus QUIC streams) to GoPs and uses stream prioritization to starve older media, head-of-line blocking and consequent buffering can be avoided. When packet loss occurs and data of the current GoP is lost, chances are that the next GoP is already

on its way before the current one is retransmitted. The stream continues, and the retransmitted GoP data gets starved and potentially cancelled [Luke Curley, 2024b].

HLS/DASH has long media segments (multiple seconds long), which have to be completed before they can be sent (adding the long segment duration to the latency) and have to be fully downloaded before being able to switch to a different bitrate (problematic during congestion) [Luke Curley, 2022].
Each MoQ segment corresponds to a single GoP in size. GoP size varies by application, but is typically 0.5 - 2 seconds long [Samis, 2020]. Since MoQ segments are mapped directly to Group Streams and MoQ streams use QUIC streams, multiple segments can be retrieved in parallel [Luke Curley, 2024b]. MoQ segments have the advantage of not having to be downloaded entirely before switching the bitrate. The protocol can start a new subscription to the Track with the desired bitrate and close the existing subscription.

HTTP, a request/response protocol, always needs the client to poll for data. Consequently, HLS/DASH and LL-HLS/LL-DASH do this too since they run on top of HTTP. The client must request every media segment before the server can send it.
QUIC allows both endpoints to create unidirectional or bidirectional streams on which an arbitrary amount of data can be sent. The data in unidirectional streams is carried from the stream's creator to the other endpoint, while bidirectional streams allow either endpoint to send data to the other [Iyengar and Thomson, 2021].
None of the MoQ Transfork streams requires data to be polled; data can be sent whenever it is available. Since there is no data polling in MoQ, the protocol does not have the issues that arise with polling, such as a high request rate, like in LL-HLS/LL-DASH, due to short segment sizes [Curley, 2025a].

WebRTC's hard-coded quality compromise to achieve lower latency is absent in MoQ since the protocol caters to multiple latency and quality targets [Curley, 2025a]. However, if specific applications require the same quality compromise to attain lower latency, this can be implemented in the application using MoQ.

Scaling WebRTC with SFUs presents several challenges, primarily because the transport layer, which uses RTP, is tightly coupled with the application layer, which also relies on RTP. To maintain compliance with WebRTC specifications, SFUs often resort to hacky workarounds. As a result, many services develop custom SFU implementations tailored to their specific application requirements [Luke Curley, 2024b].
As previously mentioned in Section 2.4.2, MoQ avoids this by decoupling the media and the transport layer. MoQ Transfork headers only contain the necessary information needed for optimal caching and fan-out. This allows any data to be encoded in the payload; it can be encrypted and does not necessarily have to be media [Luke Curley, 2024b].

Chapter 3

Logging

The first step to visualize network protocols, and in this case, MoQ specifically, is to log what happens at each endpoint that is involved. These logs can contain a tremendous amount of information, which in turn can be parsed to create visualizations that facilitate the debugging and analysis of the logged network protocols. This chapter examines the emerging qlog standard, its applicability to protocols such as QUIC and MoQ, the development of a logging library based on qlog, and the modifications made to existing QUIC and MoQ Transfork implementations to incorporate qlog-based logging functionality.

3.1 qlog

qlog is an extensible structured logging format designed for use with network protocols. It allows easy data sharing in order to facilitate debugging and analysis of the logged network protocols. All qlog logging schemas are independent of serialization formats, which allows all logs to be exported to various data formats, such as JSON and CSV. At the time of writing, qlog is still under active development [Marx et al., 2025a].

Endpoint logging can be particularly useful in understanding the behavior of applications using network protocols, especially when protocol data is encrypted. In order to do endpoint logging, many applications utilize their own custom, non-standard logging format. These custom logging formats hinder the development of universal analysis tools that parties with access to logs can use [Marx et al., 2025a].

The primary objective of qlog is to define a set of standardized features and default characteristics that logging files should include in order to facilitate the development of generic, reusable tools. These tools can then process and analyze logs generated by various protocols and use cases [Marx et al., 2025a].

3.1.1 Main Logging Schema

qlog is designed to be a streamable, flexible event-based logging format where event data and metadata are stored together. It is meant to reduce overhead for the log producer (protocol implementations), but at the cost of increasing the complexity for the log consumers (tools using logs). Furthermore, it is designed to be extensible, pragmatic, aggregation- and transformation-friendly. An example of this is that events can be tagged to specific contexts (using `group_ids`). qlog achieves this using a log file, traces, and events hierarchy. Log files can contain multiple traces, and each trace can contain multiple events [Marx et al., 2025a]. This concept is visualized in Figure 3.1.

The qlog draft defines two schemas: `QlogFile` for use with non-streamable file formats, and `QlogFileSeq` for use with streamable file formats. Since the rest of this thesis uses `QlogFileSeq`,

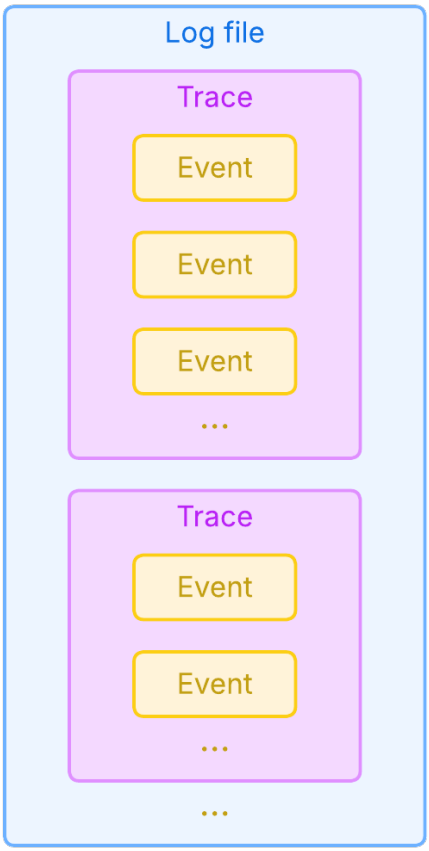


Figure 3.1: Visual representation of the hierarchical structure of qlog

this explanation will not further discuss `QlogFile`. The intention of a log file is to hold a collection of related events. Log files contain a file schema, serialization format, optionally a title and description, and a trace. The file schema is a URI that refers to the concrete log file schema. The qlog draft defines two values for the file schema: `urn:ietf:params:qlog:file:contained` and `urn:ietf:params:qlog:file:sequential`. The former is meant to be used with non-streamable file formats, while the latter is intended to be used with streamable file formats. The serialization format field contains the file format used for serialization, such as JSON or CSV [Marx et al., 2025a].

The qlog draft describes a trace as "a log of a single data flow collected at a single location or vantage point." For instance, when logging QUIC, a trace contains events associated with a single QUIC connection, recorded from the perspective of either the client or the server. Traces contain not only a list of events, but also the event schemas, and optionally a title, a description, common fields, and a vantage point. The event schemas are URIs identifying concrete event namespaces; this is needed to determine where each logged event of that trace is defined, since different sets of events can be defined for the same protocol. The common fields field allows data that is identical to all events in the trace to be logged in one place. That way, the identical fields do not have to be logged explicitly for each event. All fields in the common fields are optional, and they consist of a path, time format, reference time, group ID, and any number of custom fields. These fields will be discussed in the following paragraph. The vantage point describes the origin point of a trace. This can be the client, the server, or an observer

somewhere between the client and the server [Marx et al., 2025a].

Events are logged at a specific point in time and encapsulate detailed information relevant to the particular logging use case. The qlog draft does not define individual events since these are protocol-specific, but does define several fields that all events have in common, regardless of the protocol. These fields are the time, the name, the event data, optionally a path, time format, group ID, and some system information; on top of this, any number of custom fields can be added. The time field contains a timestamp that indicates when an event occurred; its value is relative to a chosen reference time and the time format. The name field of an event is the concatenation of the namespace identifier, a colon, and the event type identifier. This value must be globally unique to avoid name collisions with other events from other schemas. An example of such an event name would be `quic:packet_sent` where `quic` is the namespace identifier of the event schema `urn:ietf:params:qlog:events:quic`. The data field contains the data of a specific event; the writers of the events choose the exact data included in this field. The time format field specifies whether the time of an event is relative to an epoch value (part of the reference time) or to the time of the previously logged event (delta-encoded value). The path field makes it possible to associate an event with a network path; the exact content of this field is left to implementers, but a possible value is the 4-tuple of source and destination addresses. The group ID field allows events to be grouped together by assigning them the same group ID. Use of this field is optional, and its specific semantics are left to individual implementations. The system information field can be utilized to log system-specific details, which allows the process ID, processor ID, and thread ID of the system running the protocol to be logged [Marx et al., 2025a]. Listing 3.1 shows an example of a qlog event.

```
{
    "time": 1553986553572,

    "name": "quic:packet_sent",
    "data": { ... },

    "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",

    "time_format": "relative_to_epoch",

    "ODCID": "127ecc830d98f9d54a42c4f0842aa87e181a"
}
```

Listing 3.1: Example of a generic qlog event serialized as JSON [Marx et al., 2025a]

Since JSON is not a streamable file format, because new events cannot be appended at the end of the log file, JSON Text Sequences (JSON-SEQ) can be used for `QlogFileSeq` [Marx et al., 2025a]. JSON Text Sequences are very similar to regular JSON but are streamable. This format works by having any number of JSON texts (valid JSON blobs) prefixed with a Record Separator character (`0x1E`), and suffixed with a Line Feed character (`0x0A`). Appending to JSON Text Sequences works by simply appending the Record Separator, then appending the JSON data, and finally appending the Line Feed [Williams, 2015]. Instead of the trace containing a list of events, `QlogFileSeq` appends each new event to the end of the JSON-SEQ file, an example of this is shown in Listing 3.2. Unlike the non-streamable format, all the appended events belong to the same trace, which means that all log files using this format can only hold a single trace [Marx et al., 2025a].

```
// list of qlog events, serialized in accordance with RFC 7464,
// starting with a Record Separator character and ending with a
// newline.
// For display purposes, Record Separators are rendered as <RS>

<RS>{
    "file_schema": "urn:ietf:params:qlog:file:sequential",
    "serialization_format": "application/qlog+json-seq",
    "title": "Name of JSON Text Sequence qlog file (short)",
    "description": "Description for this trace file (long)",
    "trace": {
      "common_fields": {
        "group_id":"127ecc830d98f9d54a42c4f0842aa87e181a",
        "time_format": "relative_to_epoch",
        "reference_time": {
            "clock_type": "system",
            "epoch": "1970-01-01T00:00:00.000Z"
        },
      },
      "vantage_point": {
        "name":"backend-67",
        "type":"server"
      },
      "event_schemas": ["urn:ietf:params:qlog:events:quic",
                        "urn:ietf:params:qlog:events:http3"]
    }
}
<RS>{"time": 2, "name": "quic:parameters_set", "data": { ... } }
<RS>{"time": 7, "name": "quic:packet_sent", "data": { ... } }
...
```

Listing 3.2: Example of a qlog file using JSON Text Sequences as its serialization format [Marx et al., 2025a]

The qlog draft also defines a raw info datatype. This contains raw information and can be helpful during the tuning of packetization behavior or determining the overhead of encoding. This datatype should remain empty if not required, since it can take up considerable space and impact privacy and security. The raw info contains three optional fields: a length value, a payload length value, and the raw data displayed as a hexadecimal string. The length value refers to the complete length of the logged entity in bytes; this includes potential headers and trailers. The payload length value refers to the length of the logged entity's payload, which does not include headers or trailers. The data field is a hexadecimal string of the raw representation of the entire logged entity. If the data field is truncated for privacy or security reasons, length and payload length values should still represent the actual lengths, not the truncated ones. However, these are general definitions for this datatype; the exact usage of the fields depends on the use case [Marx et al., 2025a].

Finally, the draft defines a `QLOGFILE` environment variable that can be utilized to specify the path where the qlog file should be saved [Marx et al., 2025a].

3.1.2 QUIC qlog Definitions

Apart from working on the main logging schema of qlog, the IETF is also already working on its qlog event definitions for several protocols, including QUIC. The "QUIC event definitions for qlog" draft describes these qlog event definitions and their associated metadata for the core QUIC protocol and various QUIC extensions. The draft registers the `quic` namespace

with event schema URI `urn:ietf:params:qlog:events:quic` [Marx et al., 2025b]. Like many others cited in this thesis, this document is under active development and thus still a work in progress.

The draft recommends using QUIC's Original Destination Connection ID (ODCID), which is chosen by the client when starting a connection with the server and does not change during the connection, as the group ID to link events to the correct connection [Marx et al., 2025b].

Since the implementation only uses the `packet_sent` and `packet_received` events, the reasoning behind this decision will be explained in more detail in Section 3.2.2, the other events will not be discussed. The draft [Marx et al., 2025b] defines 34 qlog events for the QUIC protocol, subdivided into the following categories:

- Connectivity: events concerning the QUIC connection state, which include when a connection starts, closes, and updates the connection ID. This category contains eight events.
- Transport: events concerning data transport, which include when a packet is sent, received, dropped, and acknowledged. This category contains seventeen events.
- Security: events concerning updates to cryptographic keys. This category contains two events.
- Recovery: events concerning packet loss detection and congestion control, which include when congestion state is updated, when a packet is deemed as lost, and when data is marked for retransmission after packet loss has been detected. This category contains seven events.

Before discussing the `packet_sent` and `packet_received` events, it is important to explain the structure of QUIC packets. QUIC packets start with a header. The header has a long and a short form; the long header is used during connection establishment, while the short header is used after the connection is established. The long header data includes the packet type, the used QUIC version, the destination connection ID, the source connection ID, and a payload specific to the packet type, among other fields. The short header data includes the destination connection ID, the packet number, and a payload, among other fields. Furthermore, QUIC includes version negotiation packets, which are similar to long-header packets in structure but do not fully conform to the long header format. These are sent by the server when it does not support the QUIC version selected by the client. Version negotiation packets include the QUIC version (always set to 0), the destination connection ID, the source connection ID, and a number of supported QUIC versions, among other fields [Iyengar and Thomson, 2021]. The other fields of these headers and the version negotiation packets are not discussed here as they are irrelevant to this thesis.
According to [Iyengar and Thomson, 2021], the packet type of the long header refers to one of the following packet types:

- Initial: first packets sent by the client and server. The type-specific payload, which is still part of the header, of these packets contains a token, the packet number, and the payload.
- 0-RTT: allows clients to send application data before the server has responded to the client's initial packet in certain circumstances. The type-specific payload of these packets contains the packet number and the payload.
- Handshake: sent after the first server response. The type-specific payload of these packets contains the packet number and the payload.
- Retry: can be sent by the server during the handshake to validate the client's address. The type-specific payload of these packets contains a token.

The short header does not include a packet type since these headers only carry 1-RTT packets, which are sent after the handshake [Iyengar and Thomson, 2021].

The destination and source connection IDs are instances of connection IDs [Iyengar and Thomson, 2021]; these were discussed in more detail in Section 2.4.2.
Packet numbers in QUIC are not just integers; they are divided into three packet number spaces: the initial, handshake, and application data space. Each of these spaces starts with packet number 0. The initial space contains all initial packets, the handshake space all handshake packets, and the application data space all 0- and 1-RTT packets. Version negotiation and retry packets do not carry packet numbers [Iyengar and Thomson, 2021].
Tokens in initial and retry packets are opaque data blobs used for address validation [Iyengar and Thomson, 2021].
Finally, the payload of QUIC packets contains a sequence of frames. They begin with a frame type and are then followed by type-dependent fields. Some frames have multiple frame types; this is done so that frame-specific flags can be encoded in the frame type. The QUIC RFC [Iyengar and Thomson, 2021] defines the following frames:

- `PADDING`: used to increase the packet size of an initial packet to the minimum required size or to protect against traffic analysis. They carry no additional data.
- `PING`: used to check if peers are still alive or reachable. They carry no additional data.
- `ACK`: used to acknowledge the reception of packets. They contain the packet number of the largest acknowledged packet, the delay of the acknowledgment (in μs), ranges of packets that alternate between a range of packets that have not been acknowledged (gap) and a range of packets that have been acknowledged, and Explicit Congestion Notification (ECN) information.
- `RESET_STREAM`: used to terminate the sending side of a stream abruptly. They contain the ID of the terminated stream, an application protocol error code indicating the termination reason, and the final size of the stream in bytes.
- `STOP_SENDING`: used to inform the peer that incoming data on the stream will get discarded, thus requesting the peer to stop sending data on that stream. They contain the ID of the stream and an application protocol error code.
- `CRYPTO`: used to transmit cryptographic handshake messages. They contain a byte offset for the data in the stream, the length of the data in this frame, and the cryptographic message data.
- `NEW_TOKEN`: used to send a new token to the client that can be used in an initial packet for a future connection. They contain the new token.
- `STREAM`: used to carry stream data. The frame type of this frame has multiple values since it encodes an `OFF`, `LEN`, and `FIN` bit. The `OFF` bit indicates if an offset field is present, the `LEN` bit indicates whether a length field is present, and the `FIN` bit is set if the stream has ended. The absence of the offset field implicitly means an offset of 0, and the absence of the length field indicates that there are no further frames; the packet ends with the stream data. They contain the stream ID, the offset and length fields based on their corresponding bits, and the stream data.
- `MAX_DATA`: used to limit the amount of data that can be sent on the entire connection. They contain this data limit in bytes.
- `MAX_STREAM_DATA`: used to limit the amount of data that can be sent on the given stream. They contain the stream ID and the data limit for this stream.
- `MAX_STREAMS`: used to limit the total number of uni- or bidirectional streams that may be opened on the connection; every stream counts towards this limit, even closed ones. The frame type has multiple values since it encodes the stream type (uni- or bidirectional) in its value. These frames only contain this limit.
- `DATA_BLOCKED`: used to indicate to the receiver that the sender has extra data to be sent, but is blocked by the maximum amount of data on the entire connection. They contain

this data limit.

- `STREAM_DATA_BLOCKED`: used to indicate to the receiver that the sender has extra data to be sent on the given stream, but is blocked by the maximum amount of data on that stream. They contain the stream offset at which the blocking happened.
- `STREAMS_BLOCKED`: used to indicate to the receiver that the sender needs to open a new stream, but is blocked by the stream limit. The frame type has multiple values since it encodes the stream type (uni- or bidirectional) in its value. These frames only contain this stream limit.
- `NEW_CONNECTION_ID`: used to provide alternative connection IDs to the peer. They contain the sequence number of the new connection ID, a field indicating which connection IDs should be retired, the new connection ID, and a stateless reset token to use with the new connection ID. The stateless reset token is used to end the connection if an endpoint is unable to access the connection state.
- `RETIRE_CONNECTION_ID`: used to indicate that the sender of this frame will no longer use a connection ID that he had received earlier from its peer. They only contain the sequence number of the retired connection ID.
- `PATH_CHALLENGE`: used to check if the peer is reachable or to validate the path during connection migration. They only contain 8 bytes of arbitrary data.
- `PATH_RESPONSE`: used to respond to a `PATH_CHALLENGE` frame. They contain the 8 bytes of arbitrary data sent in the `PATH_CHALLENGE` frame.
- `CONNECTION_CLOSE`: used to inform the peer that the connection is being closed. The frame type has multiple values since it encodes the type of the error (QUIC error/no error, or application error) in its value. They contain an error code, the frame type that caused the error in case of a QUIC error, and optionally a reason phrase that provides extra information.
- `HANDSHAKE_DONE`: used to confirm the handshake to the client. They carry no additional data.

Additionally, the `DATAGRAM` frame defined in RFC 9221 as an extension to the QUIC protocol is included in the QUIC qlog definitions. This frame is used to transmit application data unreliably. The frame type has multiple values since it encodes a `LEN` bit with the same functionality as the `LEN` bit of the `STREAM` frame. These frames contain the length of the data based on the value of the `LEN` bit and the actual application data [Pauly et al., 2022].

The `packet_sent` and `packet_received` events are very similar; the only differences are the extra boolean value in the `packet_sent` event indicating whether the packet is an MTU probe packet, and the possible values for the trigger, indicating what triggered the packet being sent or received. The common data of these events includes the packet header, the embedded QUIC frames, a stateless reset token in case the packet is a stateless reset packet, a list of supported QUIC versions for a version negotiation packet, the raw info of the packet, and the datagram ID. Only the packet header is required; the rest is optional or only included in certain scenarios [Marx et al., 2025b].

The packet header contains the packet type, the packet type bytes if the packet type is unknown, the packet number if the packet has one, several flags including the spin bit, a token in case of an initial or retry packet, the packet length if type of the packet is initial, handshake, or 0-RTT, the used QUIC version, and the source and destination connection IDs. All of these values, except for the packet type, are optional or only included in some cases [Marx et al., 2025b].
The packet type can have the value ‘initial’, ‘handshake’, ‘0-RTT’, ‘1-RTT’, ‘retry’, ‘version negotiation’, ‘stateless reset’, or ‘unknown’ [Marx et al., 2025b].
Tokens include a token type, token details, and the token’s raw info. All of these token fields are optional. The token type can be ‘retry’ from a retry packet or ‘resumption’, which is used

to resume a connection. The token details field can contain any key-value pairs, since any value can be encoded in the token, these encoded values can be optionally reflected in this field [Marx et al., 2025b].
The packet length contains the sum of the length of the packet number and the payload [Marx et al., 2025b].
Connection IDs in qlog are displayed as hexadecimal strings [Marx et al., 2025b].

The QUIC qlog draft has qlog definitions for all QUIC frames defined in RFC 9000, the `DATAGRAM` frame defined in RFC 9221, and an 'unknown' type for unrecognized frame types. Each frame has a required frame type field with the same value as the frame's name (but in lowercase) and an optional raw info field. The following list provides an overview of each frame and the additional data it carries [Marx et al., 2025b]:

- `PADDING`: carries no additional data.
- `PING`: carries no additional data.
- `ACK`: carries the acknowledgment delay (in ms), the acknowledged ranges, and ECN-related fields. The acknowledged ranges field is a list of ranges, where each range is one or two numbers. A range of one number contains the only acknowledged packet number of that range. In a range of two numbers, the first is the lowest acknowledged packet number, while the second is the highest acknowledged packet number in that range. All of these fields are optional.
- `RESET_STREAM`: carries the stream ID, an application error code, the error code bytes if the error code has the value 'unknown', and the final size of the stream (in bytes). All fields, except for the error code bytes, are required.
- `STOP_SENDING`: carries the stream ID, an application error code, and the error code bytes if the error code has the value 'unknown'. All fields, except for the error code bytes, are required.
- `CRYPTO`: carries the offset and the length of the cryptographic message data. Both of these fields are required.
- `NEW_TOKEN`: carries only the required token field. The value of this field contains a token; the contents of a token were described in the previous paragraph.
- `STREAM`: carries the stream ID, the offset and length of the stream data, and a fin boolean that is assumed to be false when absent. Only the fin value is optional, but always present since it has a default value.
- `MAX_DATA`: only carries the required maximum value.
- `MAX_STREAM_DATA`: carries the stream ID and the maximum value. Both of these fields are required.
- `MAX_STREAMS`: carries the stream type and the maximum value for that stream type. Both of these fields are required. The stream type is either 'unidirectional' or 'bidirectional'.
- `DATA_BLOCKED`: only carries the limit at which the blocking occurred. This field is required.
- `STREAM_DATA_BLOCKED`: carries the stream ID and the limit. Both of these fields are required.
- `STREAMS_BLOCKED`: carries the stream type and the limit. Both of these fields are required.
- `NEW_CONNECTION_ID`: carries the sequence number, a field indicating which connection IDs should be retired, the connection ID length, the connection ID, and a stateless reset token. The stateless reset token is displayed as a hexadecimal string. All fields except for the connection ID length and the stateless reset token are optional. The length of the connection ID is optional since this field is primarily used when the entire connection ID cannot be logged due to privacy reasons, for instance.

- `RETIRE_CONNECTION_ID`: only carries the sequence number of the retired connection ID. This field is required.
- `PATH_CHALLENGE`: only optionally carries the 8 bytes of arbitrary data as a hexadecimal string.
- `PATH_RESPONSE`: only optionally carries the 8 bytes of arbitrary data from the `PATH_CHALLENGE` frame.
- `CONNECTION_CLOSE`: carries the error space, the error code, the error code bytes if the error code has the value 'unknown', the reason text, the reason bytes when the reason text is not UTF-8 or the endpoint decides not to decode the reason text, and the frame that triggered the error if it is a transport error. The error space is either 'transport' or 'application'. The error code can be a transport error, a cryptographic error, or an application error. All of these fields are optional.
- `HANDSHAKE_DONE`: carries no additional data.
- `DATAGRAM`: only carries an optional length field.
- `UNKNOWN`: only carries the required frame type bytes.

Some QUIC packets can be coalesced into a single UDP datagram, which potentially reduces the number of datagrams needed to finish the handshake and start the sending of data [Iyengar and Thomson, 2021]. The datagram ID field can be used to track which packet was included in which datagram, thus seeing which packets were coalesced into the same datagram [Marx et al., 2025b].

As can be seen from the definitions given here, these qlog events can contain deep hierarchies of detailed data.

3.1.3 Media over QUIC qlog Definitions

This thesis defines its own set of MoQ Transfork qlog definitions since they were not defined elsewhere at the time of writing. MoQ Transport does have qlog definitions [Pardue and Engelbart, 2025], but these were first introduced after the custom event definitions were developed for this thesis; at the time of implementation, these were not yet published. Even at the time of writing, only two drafts have been released, so the development of these logging events is still very early. The custom MoQ Transfork event definitions have thus not been based on the MoQ Transport ones, but have some similarities. The events defined in this thesis were inspired by the QUIC and HTTP/3 events [Marx et al., 2025b, Marx et al., 2024].

This thesis only defines events based on the messages sent between MoQ endpoints. Since the current version of MoQ Transfork does not define many different message types, and to keep it simple, every message type has two events: a 'created' and a 'parsed' event, based on some HTTP/3 events. The 'created' events are emitted when the MoQ endpoint creates the message to be sent to its peer, and the 'parsed' events are emitted after receiving and parsing a message from a peer. The following events are defined in this thesis; all of these have a `_created` and a `_parsed` suffix, which have been omitted here since the contained data is equal for both versions:

- `stream`: emitted when creating/parsing a new MoQ stream. They only contain the stream type, which can be 'session', 'announced', 'subscribe', 'fetch', 'info', or 'group'.
- `session_started_client`: emitted when creating/parsing a `SESSION_CLIENT` message. They contain a list of supported versions, a list of IDs of possible extensions, and a tracing ID since WebTransport does not expose QUIC's ODCID.
- `session_started_server`: emitted when creating/parsing a `SESSION_SERVER` message. They contain the selected version and the IDs of the chosen extensions.

- `session_update`: emitted when creating/parsing a `SESSION_UPDATE` message. They only contain the estimated bitrate of the underlying QUIC connection.
- `announce_please`: emitted when creating/parsing an `ANNOUNCE_PLEASE` message. They only contain the prefix of Tracks of which `ANNOUNCE` messages are requested.
- `announce`: emitted when creating/parsing an `ANNOUNCE` message. They contain an announce status and Track suffixes to indicate the status of specific Tracks matching the prefix of the `ANNOUNCE_PLEASE` message. The announce status can have the value 'ended', 'active', or 'live'.
- `subscription_started`: emitted when creating/parsing a `SUBSCRIBE` message. They contain the subscribe ID, the Track path, the Track priority, the Group order, the Group minimum, and the Group maximum. The subscribe ID is a unique identifier for the subscription. The Track path refers to the Track on which the subscription is requested. The priority value is the subscription's priority; subscriptions with higher priorities are transmitted first during congestion. The Group order can be ascending, descending, or the default set by the publisher and refers to the order in which Groups should be transmitted. With ascending Group order, Groups will be played in the order they are sent, while with descending Group order, the latest Groups will be played to stay live. The Group minimum refers to the minimum Group sequence number to retrieve; a value of 0 means the latest Group. The Group maximum refers to the maximum Group sequence number to retrieve; a value of 0 means to continue the subscription indefinitely.
- `subscription_update`: emitted when creating/parsing a `SUBSCRIBE_UPDATE` message. They contain updated values for an existing subscription's Track priority, Group order, Group minimum, and Group maximum.
- `subscription_gap`: emitted when creating/parsing a `SUBSCRIBE_GAP` message. They contain the starting Group, the Group count, and an error code. The starting Group is the sequence number of the first Group that the publisher cannot serve. The Group count is the number of additional Groups that cannot be served after the starting Group.
- `info`: emitted when creating/parsing an `INFO` message. They contain the Track priority, latest Group, and Group order of the Track specified in the `INFO_PLEASE` message. The latest Group is the sequence number of the latest available Group of that Track. The Group order refers to the default Group order set by the publisher.
- `info_please`: emitted when creating/parsing an `INFO_PLEASE` message. They only contain the Track path.
- `fetch`: emitted when creating/parsing a `FETCH` message. They contain the Track path, Track priority, Group sequence, and Frame sequence. The Group sequence is the sequence number of the Group to be fetched. The Frame sequence is the sequence number of the first Frame to be fetched; all Frames after this Frame are also fetched.
- `fetch_update`: emitted when creating/parsing a `FETCH_UPDATE` message. They only contain the Track priority.
- `group`: emitted when creating/parsing a `GROUP` message. They contain the subscribe ID and the Group sequence number.
- `frame`: emitted when creating/parsing a `FRAME` message. They only contain the Frame payload, represented by the raw info datatype defined in the main qlog draft [Marx et al., 2025a].

The meaning of all the explained fields is defined by [Curley, 2025a], and all of these fields are required.

The MoQ Transport event definitions draft does not define separate 'created' and 'parsed' events for each message. However, it employs a style similar to HTTP/3's event definitions,

where more generic 'created' and 'parsed' events are defined with message details in the event data. Most of the differences between the MoQ Transport events and the events defined in this thesis are because of the structural difference of the events and the differences between the two protocols [Pardue and Engelbart, 2025]. The remainder of this discussion will highlight the similarities between these sets of events, but will not discuss the differences further since these are mainly protocol differences.
The MoQ Transport events include the `control_message_created` and `control_message_parsed` events. Both contain the same data, including a stream ID, the length, the message, and the raw info. Only the stream ID and the message are required. The message is an instance of one of the possible control messages defined in the draft. These include `ClientSetupMessage`, `ServerSetupMessage`, `Announce`, `Subscribe`, `SubscribeUpdate`, and `Fetch`, among others. The `ClientSetupMessage` datatype contains data similar to the `session_started_client` events defined in this thesis. The same applies to the `ServerSetupMessage` datatype and `session_started_server` events, the `Announce` datatype and `announce` events, the `Subscribe` datatype and `subscription_started` events, the `SubscribeUpdate` datatype and `subscription_update` events, and the `Fetch` datatype and `fetch` events [Pardue and Engelbart, 2025].

3.2 Implementation

This section begins by presenting the implementation of a custom logging library that implements the main logging schema of qlog [Marx et al., 2025a], incorporates the QUIC qlog draft events [Marx et al., 2025b], and extends support to include the MoQ Transfork events defined as part of this thesis. Subsequently, the modifications made to the Quinn QUIC implementation [Ochtman and Saunders, 2025], which is used within the MoQ Transfork implementation, called moq-rs [Curley, 2025b], to integrate qlog support via the custom logging library, are described. Finally, analogous changes made to the MoQ Transfork implementation are discussed.

3.2.1 Logging Library

The logging library is written in the Rust programming language since Quinn and moq-rs, the codebases to which logging must be added, are written in Rust. There is an existing qlog library in Rust, simply called qlog [Cloudflare, 2025a], that was written as part of Cloudflare's QUIC implementation, named quiche [Cloudflare, 2025b]. The decision was made not to use this library since the initial focus was on logging MoQ Transfork events, and this functionality had to be added regardless. Furthermore, the main logging schema is not very extensive and is well described by the qlog draft, which made it relatively easy to translate to Rust. This was further aided by not implementing everything in the draft, but only the necessary parts for this thesis. Only the sequential file schema is used since the only supported serialization format is JSON Text Sequences. For the purposes of this thesis, it was not useful to support multiple serialization formats, and JSON Text Sequences was a better choice than JSON since it's a streamable format and still easy to serialize and parse using existing JSON libraries. Therefore, the `file_schema` and `serialization_format` variables are hard-coded to the only supported values. Other time formats than the relative to epoch one, with the epoch being '1970-01-01T00:00:00.000Z', are also unsupported since other time formats with variable epochs would not add much value to this thesis. Finally, incorporating QUIC qlog events into the library, similar to the integration of qlog's main logging schema, is relatively straightforward, owing to the clear and detailed specification provided in the draft. The same implementation approach used for the MoQ events can be applied to support QUIC events as well.

The goal of the library was to keep logging simple so as not to have to modify a lot of code in the existing QUIC and MoQ Transfork libraries. For this reason, the library uses a static qlog writer that acts like a singleton, which means static functions can be called to log events instead of having to create an object that needs to be passed to every function in which logging needs to happen. This way, codebases using the library do not have to be extensively modified

to add qlog support. Furthermore, the library provides static functions for constructing events of each type, requiring only the specific data relevant to the corresponding event. Rust allows features to be defined. Features are a conditional compilation mechanism that allows code to only be compiled when its corresponding feature is enabled [Rust, 2025]. The logging library is structured in a way that every set of events, the QUIC and MoQ Transfork events in this case, is bound to a feature. By enabling only the QUIC feature, QUIC libraries can include support for QUIC-specific logging without incorporating code related to other protocols, for instance. Furthermore, event schemas are added to the list of event schemas based on whether their corresponding feature is enabled.

The library uses the `QLOGFILE` environment variable to specify where to save the qlogfile, but also to enable logging. By not specifying the `QLOGFILE` variable, logging is disabled without having to alter the protocol implementation.

The library provides two necessary static functions to start adding qlog support: `log_file_details()` and `log_event()`. The former is called once in the application using the protocol to be logged. This function requires users to specify the necessary qlog file details. It allows the user to specify a title and description for the file and the contained trace, the vantage point, and any custom fields. Listing 3.3 shows an example of the usage of this function. The `log_event()` function is called in the protocol implementation whenever an event needs to be logged. This function expects an event, which can be created using the static functions mentioned earlier. These functions log the file and event data to the specified qlog file by first writing the Record Separator character to the file buffer, then using the serde library to serialize the data to JSON and appending it to the file buffer, and finally writing the Line Feed character to support the JSON Text Sequences format. One of the MoQ demos included in the moq-rs repository used for testing the logging and visualization implementations is a clock application, which is a simple example application where a publisher sends the current time to any subscribers each second. Since this command-line application runs indefinitely, it needs to be exited with Ctrl + C. The disadvantage of this is that Rust's built-in file buffer does not write its contents to the file when exiting the program this way. To solve this issue, the file buffer gets flushed after every event. To lessen the impact of writing each log directly to the file while the protocol is running, this operation gets delegated to a different thread. However, this creates the issue that events that are logged almost simultaneously, in the same millisecond, thus having the same timestamp, can potentially be logged in the wrong order. Events logged in the wrong order, but at a different timestamp, can be sorted back in the right order. An example of the `log_event()` function is shown by Listing 3.4.

```
QlogWriter::log_file_details(
    Some("MoQ Clock Logs".to_string()),
    None,
    Some("Publisher".to_string()),
    Some("Logs from the publisher's perspective".to_string()),
    Some(VantagePoint::new(
        Some("clock-pub".to_string()),
        VantagePointType::Client,
        None
    )),
    None
);
```

Listing 3.3: Example from the MoQ clock demo application of how the qlog file details can be logged

```
let event = Event::moq_subscription_started_created(
    request.id,
    request.path.to_vec(),
    request.priority.try_into().unwrap(),
    request.group_order as u64,
    request.group_min,
    request.group_max,
    tracing_id
);

QlogWriter::log_event(event);
```

Listing 3.4: Example from the MoQ Transfork protocol of how events can be logged

The approach of creating events using a static function worked for the MoQ Transfork implementation, but did not work for QUIC since Quinn's logic concerning the sending and receiving of packets is split into multiple functions, and even multiple files. In order to support logging these events, caching methods were added to the logging library. This allows events to be created when the first data of the event is available, events can then later be updated when extra data is available, and can finally be logged when completed. Section 3.2.2 will go into more detail about this.

3.2.2 QUIC Logging

While implementing qlog in QUIC, the decision was made only to log the `packet_sent` and `packet_received` events. The reasoning is that these two events contain all the data sent over the connection, which helps see how MoQ and QUIC interoperate. Some of the other events, such as `connection_started` and `version_information`, can be derived from the data in the packets. The remaining events are helpful, but not as valuable to log as the `packet_sent` and `packet_received` events. This way, the focus lies on logging these two events.

Adapting the Quinn QUIC library to add qlog logging was initially challenging due to the size of the codebase and never having looked at this source code before. The first step here was understanding the structure of the code and getting to know what roughly happens in each file. During this process, it became clear that simply creating events and calling the `log_event()` function would not work for the `packet_sent` and `packet_received` events. The `packet_sent` event cannot be logged immediately before invoking the operating system kernel's UDP send function, as that stage has already encrypted the packet(s) encapsulated within the UDP datagram. Similarly, for the `packet_received` event, upon receipt of a UDP datagram, the enclosed packet(s) remain encrypted and therefore cannot yet be logged. In both cases, logging must occur at a later point, after decryption for incoming packets, or prior to encryption for outgoing packets has taken place. Logging elsewhere was not feasible due to Quinn's packetization functionality. Quinn's logic concerning the building of packets is split up into multiple functions and even multiple files, and the headers and frames are directly encoded into a byte buffer when they are added to the packet. There is no point where all the packet data is available in one place. The ability to cache QUIC events was added to the logging library to fix this issue.

The caching functionality works by keeping a hash table of QUIC packets. For this to work, a unique key is needed for each packet, which is why it combines the ODCID and the packet number. The packet number includes the packet number space (initial, handshake, or application data) since packets in different spaces can have the same number. When the header for a new packet gets constructed, an incomplete `packet_sent` event is created and cached in this hash table. This can still be done by calling a static function and not having to keep any additional state in the protocol implementation. Afterward, when other functions add frames to the packet, these frames are added to the event by searching the hash table for the cached packet

and updating it (this functionality is implemented in the logging library). Finally, when sending the packet via a UDP datagram, a call to the logging library is made to remove the packet from the cache and actually log it. Logging of the `packet_received` event works similarly. This implementation approach required only the construction of the logging data, invoking the appropriate caching and logging functions, and modifying a few functions to accept the ODCID and packet number as additional arguments to enable correct logging behavior in the Quinn protocol. However, two additional issues arose. First, a sent packet's header does not initially contain the correct length since frames have not been added yet. This is fixed by updating the length value right before encryption happens. The second one is that the timestamp of the received event needs to be cached along with the packet instead of being generated when logging, so this value is also cached with each received packet.

The focus here was on logging each packet and the frames contained in its payload, which is why some extra data is not logged at the time. This data includes, among other fields, the datagram ID, the sent/received trigger of a packet, and the header flags. An additional small change that was made involved the logging of `PADDING` frames. Instead of logging these separately, the length field of its raw info variable is set to the number of `PADDING` frames, so as not to clutter the logs with these frames since some packets can contain hundreds of them.

3.2.3 Media over QUIC Logging

As with Quinn, adding qlog support to the MoQ Transfork implementation was initially challenging. However, it was less complicated since the protocol is still in early stages and the codebase is less extensive than the Quinn one. After understanding the structure and knowing where messages are sent, it was reasonably straightforward to add qlog support.

The only problem was that the WebTransport API does not expose the ODCID, which meant finding another way to link events to the same connection on the MoQ layer. A stable ID is accessible on each endpoint, but this turned out to be the memory address of the connection, which is a different value on each endpoint. During a conversation with Luke Curley, the developer of the moq-rs implementation, about this issue, he suggested adding a tracing ID to the `SESSION_CLIENT` message since a value shared with both endpoints needs to be negotiated. This way, the client can choose a random ID both sides use. The tracing ID is implemented as a 64-bit unsigned integer encoded to a variable integer. Variable integers use 2 bits to denote if the integer is 1, 2, 4, or 8 bytes. This means that, for a 64-bit integer, 62 bits are used for the actual integer value, which is why the tracing ID is generated as a random integer between 0 and $2^{62} - 1$. Adding the tracing ID to the `SESSION_CLIENT` message has one minor issue. A Session stream is opened before the `SESSION_CLIENT` and `SESSION_SERVER` messages are exchanged. When logging the creation and parsing of this stream, the tracing ID is not yet available. This is solved by caching these logs until the logs for creating or parsing the `SESSION_CLIENT` message are created, which happens directly after opening the Session stream. The group ID of the `stream_created` or `stream_parsed` logs is then updated with the correct tracing ID, after which they are actually logged, followed by the logs of the `SESSION_CLIENT` message.

In order to aid the visualizations, a `main_role` was added to the custom fields of the trace's common fields. It is intended to clarify what each MoQ endpoint mainly does. Its possible values are 'publisher', 'subscriber', 'pubsub', and 'relay'. For instance, a streamer on a platform like Twitch is mainly a publisher since they publish videos on the website, but will also subscribe to their viewers' chat. The viewers are mainly subscribers since they mostly consume the content published by the streamer, but can publish their own data by sending chat messages. Participants in an online meeting, using platforms such as Google Meet or Zoom, mainly both publish and subscribe ('pubsub') since they publish their own webcam and microphone output, but also subscribe to the video and audio of other participants. Relays are essentially subscribers and publishers, but with a specific functionality of forwarding the subscribed data, which is why they have their own category. The functionality of this additional `main_role` field will be discussed in Section 4.2.2.

Eventually, not a lot of the existing moq-rs code needed to be adapted due to the structure of the logging library. Apart from adding the tracing ID and calling the logging functions, only a handful of functions required the tracing ID as an additional parameter. Additionally, a few dependencies needed to be adjusted. The WebTransport library used by the MoQ Transfork implementation employs Quinn for QUIC support. This dependency needed to be changed to the Quinn fork with qlog support. After this, the WebTransport dependency in moq-rs needed to be adjusted to the version with qlog support in QUIC. To test whether logging works, logging was added to the clock and relay applications in the moq-rs repository. This was done by simply calling the `log_file_details()` function and setting the `QLOGFILE` environment variable, as described in Section 3.2.1. The final result of adding qlog support to MoQ and QUIC is that logs of both protocols are written to a single file for each endpoint. To illustrate this, an excerpt from one of the generated qlog files is shown in Listing 3.5. It also shows the tracing ID and how the group ID is set to this value to link logs to the same MoQ session.

```
// The Record Separator byte is represented here as <RS>

<RS>{
    "time": 1749752980574,
    "name": "quic-10:packet_sent",
    "data": {
        "header": {
            "quic_bit": true,
            "packet_type": "1RTT",
            "packet_number": 6,
            "dcid": "1370a84baa3f095a"
        },
        "frames": [
            {
                "frame_type": "stream",
                "stream_id": 4,
                "offset": 0,
                "length": 3,
                "fin": false
            }
        ],
        "is_mtu_probe_packet": false
    },
    "group_id": "046b03a70b43866ccaea2e04a12bc52a8d037426"
}
<RS>{
    "time": 1749752980574,
    "name": "moq-transfork-03:stream_created",
    "data": {
        "stream_type": "session"
    },
    "group_id": "3562649542218034601"
}
<RS>{
    "time": 1749752980574,
    "name": "moq-transfork-03:session_started_created",
    "data": {
        "supported_versions": [
            4278955268
        ],
        "extension_ids": [],
        "tracing_id": 3562649542218034601
    },
    "group_id": "3562649542218034601"
}
```

Listing 3.5: Excerpt from a generated qlog file with both MoQ and QUIC logs

Chapter 4

Visualization

The ability to generate qlog files containing events from both QUIC and MoQ enables the creation of various insightful visualizations. Such visualizations can reveal MoQ's scalability, low-latency support, and relationship with QUIC. The qlog data serves as the foundation for generating the desired visualizations. This chapter examines qvis, a toolsuite designed to visualize QUIC and HTTP/3. Afterward, the development of an application able to construct visualizations based on the generated MoQ and QUIC logs, named moq-vis, is discussed.

4.1 qvis

qvis is a QUIC and HTTP/3 visualization toolsuite [Marx, 2024] capable of importing qlog files, among other formats, and served as the inspiration for moq-vis. Its home page allows users to load files, including examples provided by the tool itself. The second page presents a sequence diagram depicting the traffic between a QUIC client and server. Each endpoint is represented by a vertical axis along which events are positioned according to their timestamp. `packet_sent` and `packet_received` events are visualized as arrows connecting the two endpoints, annotated with details such as the packet number space, packet number, and the frames contained in the packet's payload. Other events appear as labels alongside the corresponding axis. Clicking on an arrow annotation or event label opens a modal window containing the full event details [Marx, 2024]. An example of a qvis sequence diagram is shown in Figure 4.1.

The third page displays congestion-related information. This includes a chart showing the sent data over time, when that data was acknowledged, which data was lost, as well as the connection- and stream-level flow control limits, the congestion window, and the bytes in flight. Another chart on this page shows the Round-Trip Time (RTT) over time [Marx, 2024], as illustrated in Figure 4.2.

The fourth page visualizes multiplexing behavior [Marx, 2024], which refers to how data from multiple parallel QUIC streams is transmitted over time. Data may be transmitted sequentially or in chunks scheduled via Round-Robin (RR) [Marx et al., 2020]. An example is shown in Figure 4.3.

The fifth page focuses on packetization, illustrating which frames and which streams' data are packed into each QUIC packet, as in Figure 4.4. Finally, the sixth page contains a statistics table summarizing metrics such as the number of events and the counts of each QUIC frame type. Many of these visualizations are interactive, revealing additional information on hover [Marx, 2024].

Developers can form hypotheses about potential issues using general-purpose tools such as the sequence diagram and statistics table. More specialized views, such as the congestion, multiplexing, and packetization charts, allow for deeper investigation. The adoption of qlog

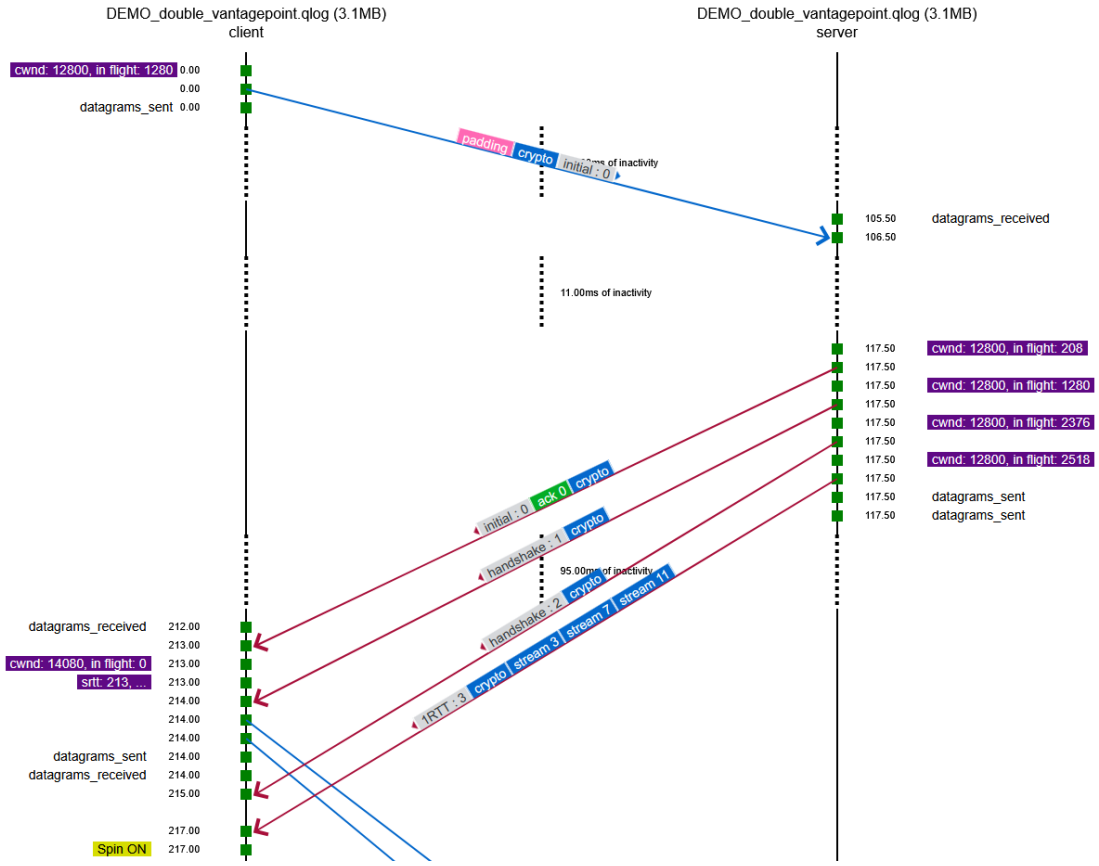


Figure 4.1: Example of the beginning of a sequence diagram in qvis [Marx, 2024]

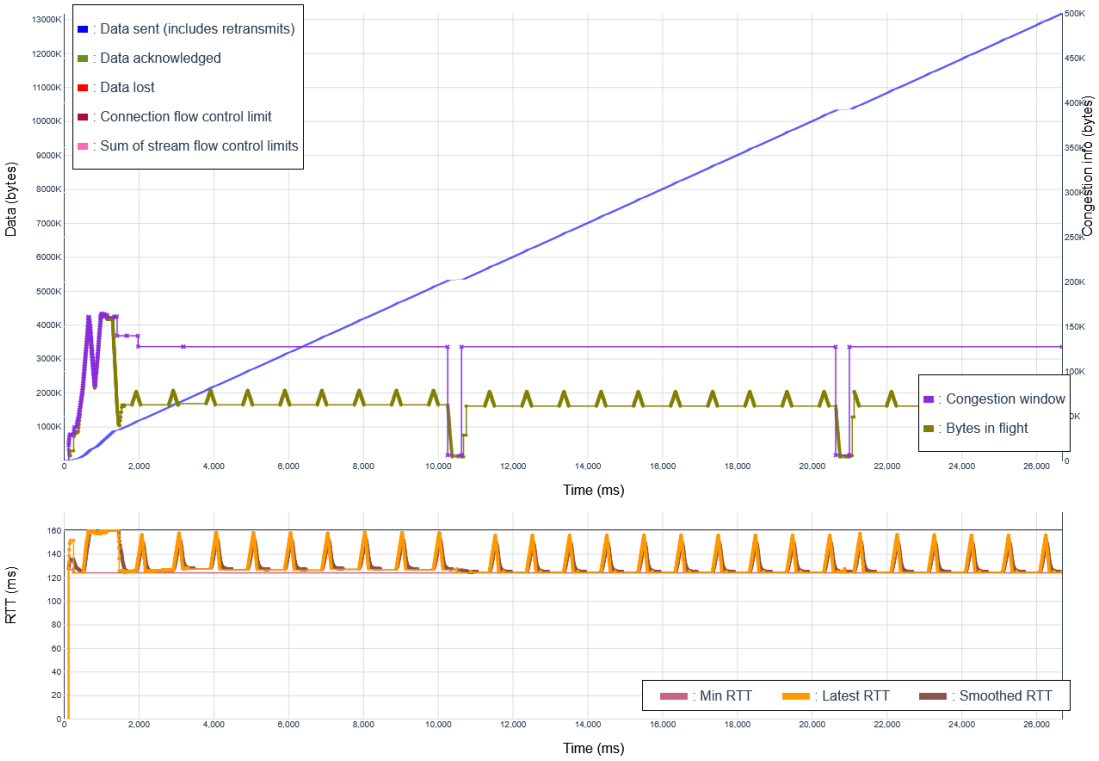


Figure 4.2: Example of the congestion control charts in qvis [Marx, 2024]

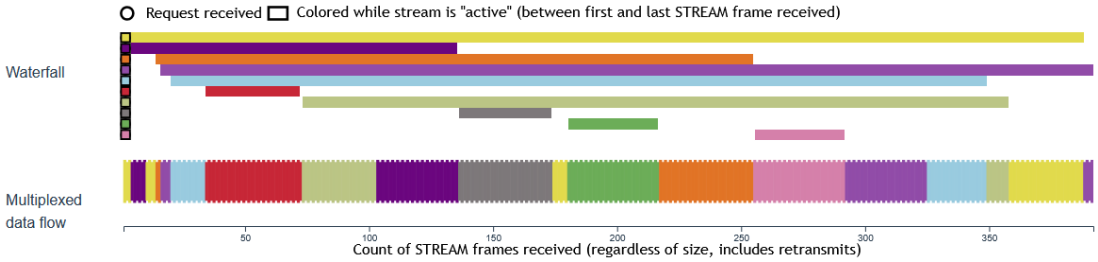


Figure 4.3: Example of the multiplexing charts in qvis [Marx, 2024]

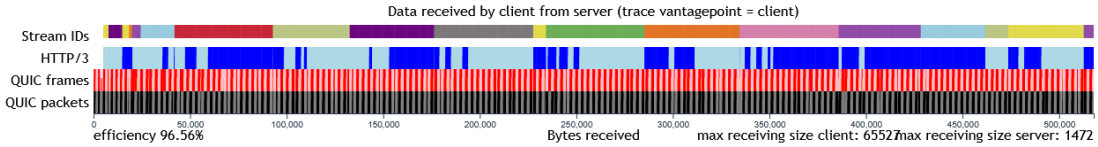


Figure 4.4: Example of the packetization chart in qvis [Marx, 2024]

and qvis within the QUIC community demonstrates the technology's value; 12 of the 18 active QUIC stacks support qlog. Multiple problems in different QUIC implementations have already been detected and resolved using qlog and qvis, demonstrating their effectiveness in improving debugging capabilities [Marx et al., 2020].

4.2 Implementation

This section presents the implementation of moq-vis, a visualization application designed to load and analyze qlog files from multiple endpoints, generating visual representations based on the information contained in the traces. MoQ is designed to support scalable and low-latency livestreaming; these characteristics are reflected in the visualizations. Scalability is represented through a network graph (Section 4.2.2), while low-latency aspects are captured in the latency charts (Section 4.2.4). In addition, a sequence diagram is included to visualize protocol-level interactions (Section 4.2.3). All visualizations are designed to be interactive, so as not to overwhelm users with too much information on the screen. Additional details about traces, specific events, or other data are available on demand. A minor drawback of qvis is that it contains interactive elements without any visual indication of their interactivity. In contrast, moq-vis makes interactivity apparent by highlighting elements on hover and changing the mouse cursor.

Following the example set by qvis, the decision was made to implement moq-vis as a web-based application, ensuring accessibility without requiring users to install additional software. Due to the complexity and custom nature of the visualizations, the D3 library was selected for its expressiveness and flexibility in building interactive data-driven graphics. In order to keep the visualization code maintainable and modular, the application uses the React framework, which supports the development of reusable components [React, 2024]. React integrates effectively with D3 [D3, 2025]. The application is structured using Next.js [Next.js, 2025], a React-based framework that facilitates efficient web application development. Styling is done using Tailwind CSS [Tailwind CSS, 2025], further enhanced with UI components, such as toggle buttons and file inputs, from Flowbite [Flowbite, 2025], enabling a polished user interface without spending too much time on styling. All code is written in TypeScript to improve type safety and maintainability. The latest version of moq-vis is currently hosted on GitHub Pages [Grispen, 2025].

4.2.1 File Import

Before the qlog data can be visualized, it must first be imported into the application. The homepage of moq-vis provides an input field that allows users to upload qlog files directly. Additionally, a set of buttons is provided for loading predefined demo files, enabling users to explore the visualizations without the need to generate or supply their own qlog data. Below these elements, a list of the currently imported files is displayed. Each file must have a unique name; duplicate file names are ignored to ensure distinguishable traces, as each file corresponds to a single trace in the sequential file schema. An example of the homepage with several imported files is shown in Figure 4.5.

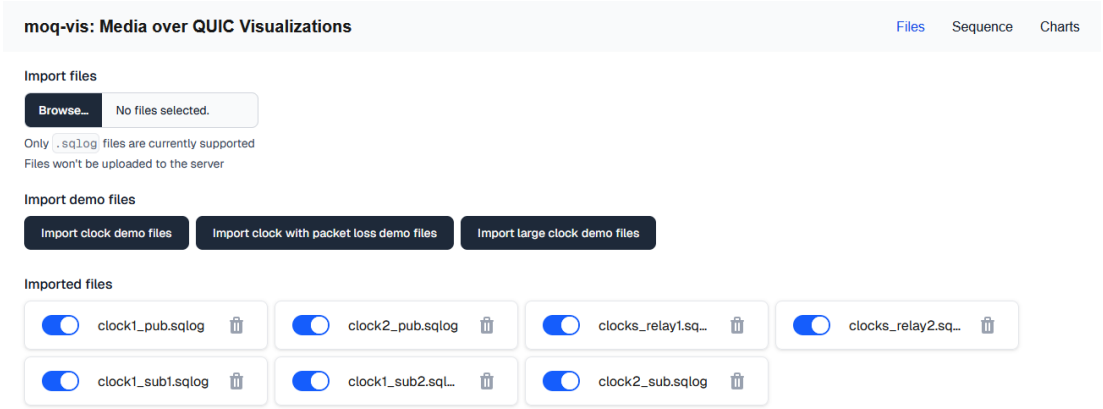


Figure 4.5: moq-vis homepage with several imported files

File import is immediately followed by a parsing step. The raw contents of each qlog file are split using the Record Separator byte, and each resulting segment is parsed as an individual JSON object using JavaScript's built-in JSON parser. The application defines TypeScript representations of the relevant data structures as specified in the qlog main logging schema [Marx et al., 2025a], the QUIC qlog events [Marx et al., 2025b], and the MoQ Transfork events defined in this thesis. This type mapping enables strong typing and direct access to event fields during visualization, and it facilitates early detection of malformed events, such as those missing required fields.

Each imported file is represented as an interactive block in the file list. This block contains the file's name, a toggle button, and a delete button. The toggle button allows users to enable or disable the file's inclusion in the visualizations. Toggling a file triggers a re-render of all visualizations on the page to reflect the updated file set. The delete button removes the file from the list and, by extension, from all visualizations. If the file is needed again after deletion, it must be reimported. Clicking anywhere else on the block opens a modal window displaying the file details for the selected file, including its title and the common fields of the trace.

4.2.2 Network Graph

The network graph visualization aims to provide a high-level overview of all endpoints and the connections between them, as inferred from the qlog traces. This representation offers insight into the overall topology of the network and helps identify the connectivity of each endpoint, including whether an endpoint is connected at all. Since MoQ Transfork is designed to scale using relays in CDNs, this visualization is particularly valuable for understanding how endpoints are interconnected. Figure 4.6 shows an example of the network graph in moq-vis.

The underlying data structure for the network graph is constructed using the set of enabled, imported qlog files. Group IDs from each trace are compared to identify which traces share the same group identifier, indicating a connection between the associated endpoints. Connections

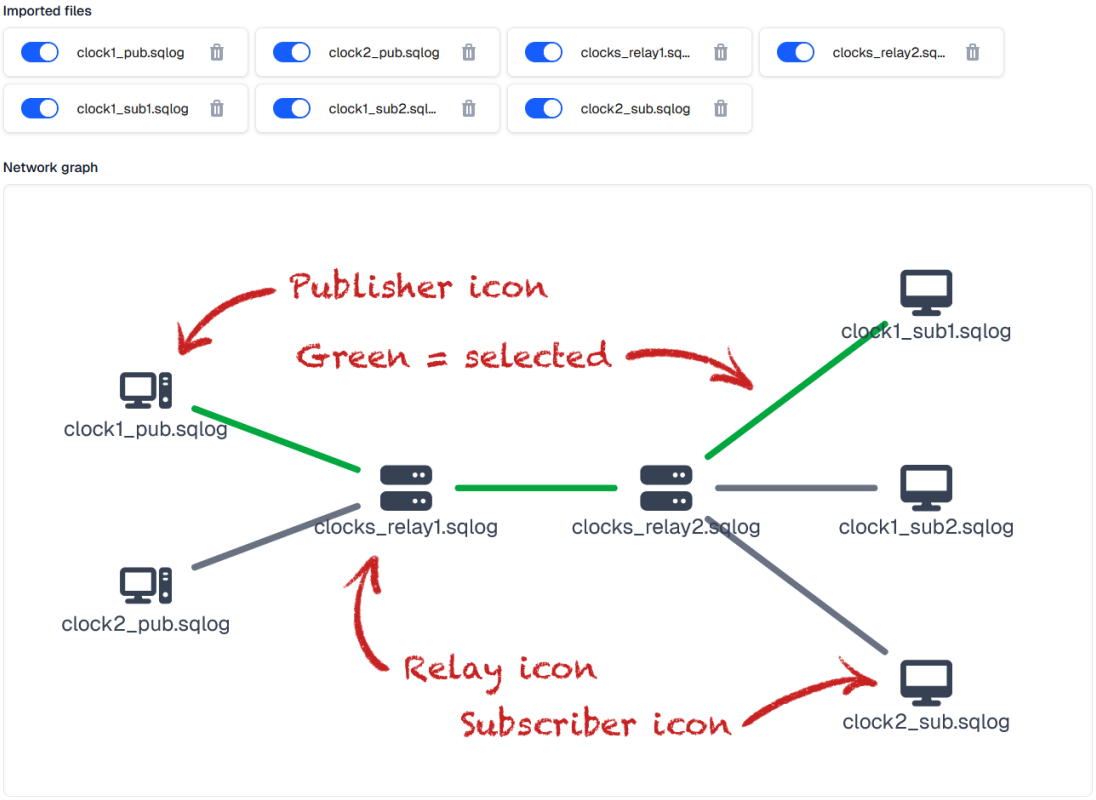


Figure 4.6: Example of a network graph with some selected connections, annotations in red were added afterward

are formed by linking the two corresponding nodes that contain only the qlog events relevant to their respective sides of the connection. Subsequently, an initial layout is computed to assign default positions to each element in the graph. This layout is designed to offer a readable and organized view immediately upon rendering, minimizing the need for manual adjustments. To enhance usability for both small and large-scale networks, the graph supports zooming and panning, allowing users to adapt the view to fit the screen comfortably.

Each node in the graph is represented by an icon and a textual identifier. Since each endpoint produces a single trace per qlog file, the filename is used as the node's identifier. Icons are assigned based on the custom `main_role` field to distinguish endpoint roles visually. Publishers are represented with a desktop computer icon, subscribers with a monitor icon, pubsub endpoints with a laptop icon, and relays with a server icon. Nodes were initially color-coded by role to provide additional differentiation. However, this approach was abandoned because it made the graph visually overwhelming. Nodes remain draggable, allowing users to rearrange the layout if the default positioning does not provide the best overview.

Edges in the graph represent connections between two endpoints as identified in the qlog traces. Their start and end positions are dynamically calculated based on the current positions of the two nodes they connect. Furthermore, edges are interactive: selecting an edge highlights it in green and adds the corresponding connection to either the sequence diagram or the latency charts, depending on the current page. Deselecting an edge reverts its color to gray and removes it from the visualizations. Figure 4.6 shows a number of selected connections. The ability to select which connections to visualize supports focused analysis. This is especially important in the context of MoQ's scalability. In large networks with many endpoints and connections, visualizing all connections simultaneously in the sequence diagram or the latency charts would be overwhelming, and it would make the visualizations more challenging to analyze. Therefore,

allowing users to choose which connections to analyze ensures that the visualizations remain interpretable.

4.2.3 Sequence Diagram

The sequence diagram is designed to provide a detailed overview of all the events occurring within the selected connections. Events are displayed chronologically, with the earliest at the top and the most recent at the bottom. Where applicable, related events across endpoints, such as a `packet_sent` and a corresponding `packet_received` event, are connected visually to illustrate the flow of the connection. The top of a sequence diagram can be seen in Figure 4.7. Another reason for not visualizing all connections in the network by default is that some endpoints, such as relays, could have more than two connections. This decision avoids visual clutter, as each endpoint can only clearly display a maximum of one connection to its left and one connection to its right in the diagram layout without introducing overlap. As such, users must explicitly select which connections to visualize via the network graph.

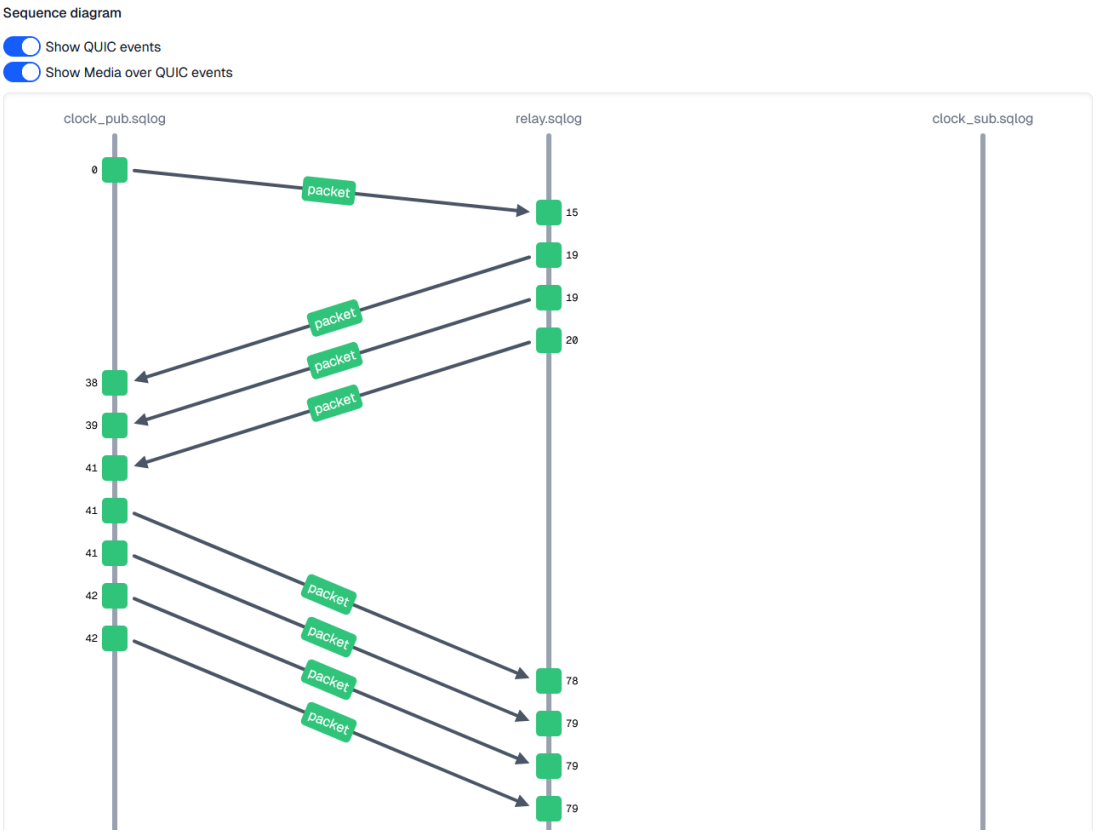


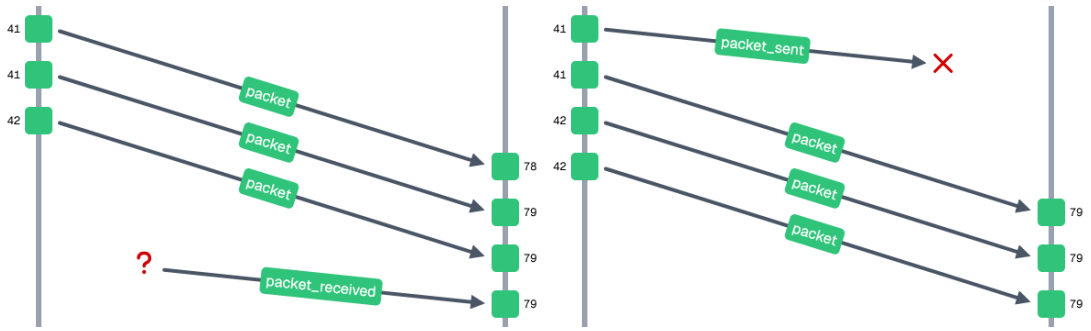
Figure 4.7: Example of the top of a sequence diagram showing two connections between three endpoints, the packets shown contain the QUIC handshake data (the right connection has not started yet)

The internal data structure of the sequence diagram is built based on the user's current selection of connections. Each event is assigned an event number, which determines its vertical placement along the corresponding endpoint's axis. These numbers are calculated such that events with identical timestamps are rendered on the same horizontal level, where possible. An exception to this arises when two events with the same timestamp are emitted by the same endpoint, resulting in overlapping events. This layout optimization makes it visually more apparent when events happen simultaneously and reduces the amount of vertical space needed. When the set of selected connections is modified, either by adding or removing a connection via the network graph, the event layout must be recalculated. Adding a new connection may insert events that

belong in the middle of other events in the diagram, while removing a connection leaves unused vertical space. Recomputing event numbers upon each change ensures visual consistency and compactness but introduces additional computational overhead.

The sequence diagram consists of one vertical axis per endpoint, labeled at the top with the filename corresponding to the endpoint's qlog file. Events belonging to each endpoint are rendered along these axes based on their event numbers and are categorized into three types: regular events, message events, and half message events. Each type is visualized differently:

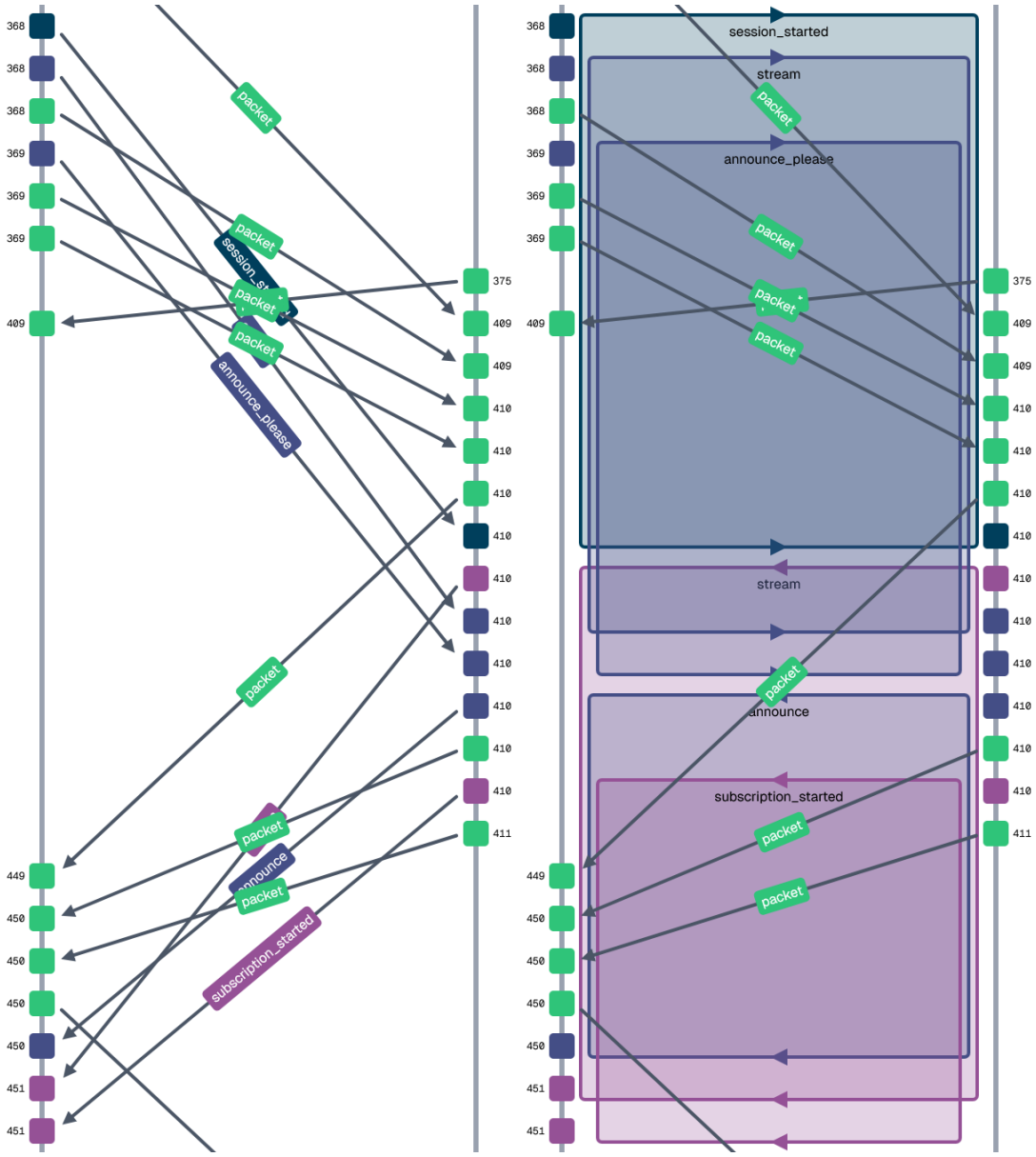
- Regular events are standalone and rendered as blocks on the axis with the event name and timestamp displayed alongside. These are currently not used since all logged events are message events.
- Message events are associated with a counterpart on the opposite endpoint, such as 'created' and 'parsed' event pairs in MoQ. These are visualized with an event block and a timestamp on both axes belonging to the endpoints involved in the message. The event blocks are similar to the regular events, but they omit the name since the name is placed on an arrow connecting the event blocks. The arrow shows the direction of the message flow (for instance, by going from the 'created' to the 'parsed' event). These events can be seen in Figure 4.7.
- Half message events occur when a counterpart of a message event is missing. If the 'created' event is absent, a question mark is shown at the arrow's origin, signaling potentially missing log data. An example of this is shown in Figure 4.8a. If the 'parsed' event is missing, a cross is displayed at the arrowhead to indicate potential data loss, such as from packet loss. This variant is displayed in Figure 4.8b.



(a) Half message event with the 'created' event missing **(b)** Half message event with the 'parsed' event missing

Figure 4.8: The two kinds of half message events

A special visualization form is used to visualize MoQ message events due to an overlap between MoQ and QUIC message events. This overlap commonly arises when a MoQ message is created, after which a QUIC packet is sent, the QUIC packet is then received, and finally, the MoQ message is parsed. If visualized with traditional arrows, this pattern leads to excessive overlap, making it difficult to see what exactly is happening. This overlap is shown in Figure 4.9a. To address this, MoQ message events are alternatively displayed as semi-transparent blocks spanning from the 'created' to the 'parsed' events between the axes of both endpoints of the connection. The background is made semi-transparent to create a visual distinction when multiple MoQ events overlap. Embedded arrows on the solid top and bottom borders of the blocks indicate message flow. Block widths vary slightly to distinguish overlapping MoQ messages further. QUIC events are rendered on top of the blocks to make them still accessible. This block format also aids in understanding which QUIC events occurred during the transmission of a given MoQ message, as any QUIC events falling within the block's bounds are considered to have happened during that time span. However, it is worth noting that some QUIC packets may coincidentally be logged during that interval, such as packets containing only `ACK` frames. Figure 4.9b shows this alternative display of MoQ messages.



(a) Multiple overlapping MoQ en QUIC events **(b)** Alternative representation of MoQ events

Figure 4.9: The same sequence of events visualized in two different ways

All of the events within the sequence diagram are interactive. Clicking regular events, the small event blocks of message events and half message events, and the arrows of half message events opens a modal window displaying the raw qlog data for that event. Clicking the arrows of message events or the transparent blocks of MoQ message events opens a modal window summarizing both related events and providing additional metadata such as latency. An example of this summary can be seen in Figure 4.10.

The colors bound to each event are dependent on the event type. QUIC events are rendered in green, while MoQ events are colored based on the stream type in which they occur. Session stream messages are dark blue, announce stream messages are dark purple, subscribe stream messages are light purple, info stream messages are orange, fetch stream messages are pink, and group stream messages are yellow. The exact color codes used for the MoQ streams are from a data visualization color palette generator [Learn UI Design, 2018].

Finally, two toggle buttons located above the diagram allow the user to enable or disable the rendering of QUIC and MoQ events, respectively. Each toggle causes the diagram to be re-rendered to either introduce or remove the corresponding events. When QUIC events are disabled, MoQ events default to the simpler arrow representation instead of blocks, as the added context provided by the block format becomes unnecessary, and the use of arrows is clearer.

4.2.4 Latency Charts

The latency charts are designed to provide an overview of the latency characteristics of established QUIC connections and MoQ sessions. This visualization is particularly relevant in the context of MoQ, which aims to support low-latency livestreaming scenarios. By visualizing the latency, users are able to assess the performance of connections and identify potential issues. Two example charts are shown in Figure 4.11.

The charts are generated based on the connections selected in the network graph. A connection pair is formed for each selected connection, consisting of the QUIC connection data and the corresponding MoQ session data. Subsequently, a separate latency chart is then created for each connection pair. Each chart is labeled with a title indicating the endpoints involved in the respective connection. The left and bottom axes are labeled clearly to indicate the displayed information and their respective units. The x-axis shows the elapsed time since the start of the connection, while the y-axis represents the latency in milliseconds. Tick marks with numerical values are included along both axes to aid interpretation.

Latency values are derived by calculating the time difference between paired ‘created’ and ‘parsed’ events. The x-coordinate (time) for each data point is determined by the normalized timestamp of the ‘created’ event, which is defined as the time elapsed since the first event of that connection. The y-coordinate (latency) corresponds to the time difference between the ‘created’ and ‘parsed’ events. Each latency chart presents this data as a sequence of circular data points. A line is drawn between consecutive points to illustrate the progression of latency over time.

The latency charts are also interactive. Users can zoom and pan along the x-axis, enabling both high-level overviews and detailed inspection of specific time intervals. As the user zooms or pans, the tick marks and their corresponding values on the x-axis dynamically update to reflect the current viewport. When the cursor hovers over a data point, the point is slightly enlarged for emphasis, and a tooltip is displayed. This tooltip provides precise information about the corresponding latency value and timestamp. This interactivity is shown in the second chart of Figure 4.11. To enhance interpretability, two toggle buttons are provided to independently control the visibility of QUIC and MoQ latency data. These toggles are color-coded to match the data points they correspond to and thus also serve as a visual legend. When toggling visibility, the axis ranges remain fixed to prevent visual shifts in the data layout.

Figure 4.10: Example of a message summary showing the sent time, received time, latency, names of both events, and the common data

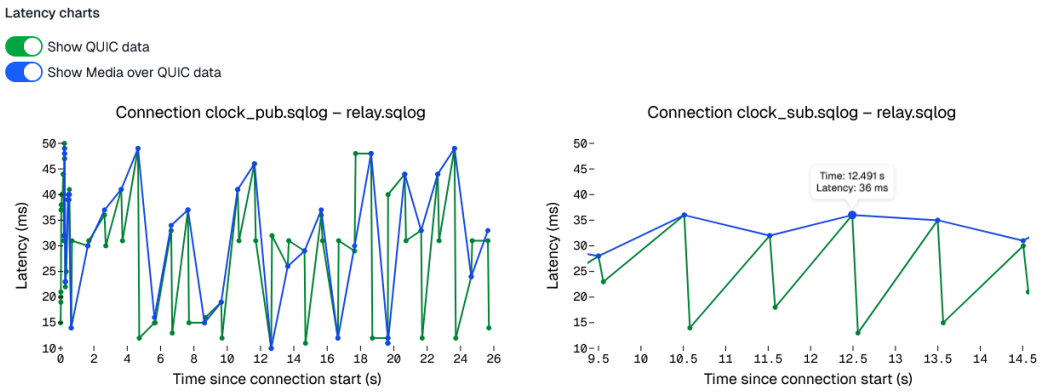


Figure 4.11: Example latency charts of two connections, the second one showing the interactivity

Chapter 5

Evaluation

The visualization application, moq-vis, is evaluated using three demonstration scenarios. The first demo features a simple topology consisting of one publisher, one relay, and one subscriber. The second demo replicates this setup but introduces packet loss. The third demo models a more complex scenario, comprising two publishers and three subscribers, two of which are subscribed to the first publisher and the third to the second publisher, as well as two relays. In this configuration, the first relay connects to both publishers and the second relay, while the second relay forwards data to all three subscribers. The last demo is the one shown in Figure 4.6.

All demos utilize the clock application from moq-rs. In this application, a publisher initially sends the current date in the format "YYYY-MM-DD hh:mm:" (without the seconds) at the start of a subscription or at the beginning of a new minute. Thereafter, the current seconds value is transmitted every second. Video data is not used in these demos, as the subscribers operate over moq-web, which employs WebAssembly to run MoQ Transfork's Rust implementation within the browser. Due to browser security restrictions, it is not possible to directly create or write to files without user interaction. Since logging occurs exclusively in the MoQ Transfork and QUIC layers, which have no knowledge of the data contained in Frames, and the use of video would primarily impact the volume of transmitted data, the decision was made to use the clock application for evaluation. Furthermore, video streams tend to generate a large amount of qlog data at high speed, and moq-vis is currently a proof of concept that is not optimized for handling large-scale qlog datasets. All demos were generated with the help of clumsy [Tao, 2022], a network manipulation tool that enables the introduction of artificial latency and packet loss. In these scenarios, clumsy was used to simulate latency in all demos and to introduce packet loss in the second demo.

5.1 Inefficient Packetization

One of the observations from the first demo is that MoQ frequently sends multiple QUIC packets in rapid succession rather than aggregating the data into a single packet. This behavior is clearly visible in the `GROUP` and `FRAME` messages of the left connection in Figure 5.1.

In this example, the first QUIC packet contains only the 3-byte payload of the `INFO` message. The second packet initiates a new QUIC stream via WebTransport and contains the 3-byte WebTransport unidirectional stream header solely. The third packet carries just a single byte of application data, representing the stream type of the new Group Stream. The fourth packet contains the Group header, which is 2 bytes in this case. The fifth packet holds the length and data of the first frame, which is the current date string without the seconds. This string is 17 characters long, encoded in 17 bytes, plus one byte for the length field, yielding 18 bytes of data in total. The sixth packet contains the length and data of the second frame, two second

characters, resulting in 3 bytes including the length byte. The remaining two packets contain only acknowledgments.

As shown by the timestamps in the figure, all of these packets are sent within two milliseconds. In contrast, the right-hand connection in Figure 5.1 transmits all the same data, except for the `INFO` message payload, totaling 27 bytes, in a single QUIC packet (the first packet on the right contains only acknowledgments and is sent earlier). Similar behavior is observed when the second Group Stream is initiated: in this case, the left connection sends only two QUIC packets. One with all the metadata of the new Group Stream and the closure of the previous one, and another with the complete date string split into two frames. Meanwhile, the connection on the right splits this data across multiple packets in the same way as the first scenario.

The sequence diagram further indicates that this is not explicitly defined behavior: in some cases, multiple MoQ messages are sent within one or two packets. Similar patterns are seen for other message types, such as `SESSION_CLIENT` and `SUBSCRIPTION_STARTED`, though it is most apparent in the example above.

The drawback of this behavior lies in the significant protocol overhead it introduces. In many of these cases, the MoQ messages contain only a few bytes of application data. Each 1-RTT QUIC packet uses a short header, and, since short-header packets lack a length field, they cannot be coalesced into a single UDP datagram [Iyengar and Thomson, 2021]. Consequently, each packet incurs the additional overhead of a UDP header and the headers of lower-layer protocols.

Whether this originates from the Quinn implementation or from how MoQ Transfork interacts with Quinn remains unclear and requires further investigation. Nonetheless, this finding demonstrates that the sequence diagram is an effective tool for identifying unwanted behavior, an important first step toward resolving problems.

5.2 Packet Loss

When analyzing the sequence diagram of the demo with packet loss, it becomes apparent how QUIC handles lost packets while MoQ continues operating normally. This process is illustrated in Figure 5.2.

In this example, MoQ Frame data is sent but lost in transit (first packet). After approximately 160 milliseconds without receiving an acknowledgment, the sender transmits two packets, each containing only three `PADDING` frames. The receiver then acknowledges the receipt of three packets sent prior to the lost packet and the two packets sent after it. Upon receiving these acknowledgments, the sender detects the packet loss and retransmits the lost packet. Since QUIC does not reuse packet numbers within a given packet number space, the retransmission is assigned a new packet number [Iyengar and Thomson, 2021]. The first retransmission is also lost, triggering the same process again. The second retransmission is successfully delivered, allowing the Frame data to be parsed and the corresponding block in the visualization to be closed.

This retransmission behavior is defined in [Iyengar and Swett, 2021]. When an acknowledgment for ack-eliciting packets is not received within a specified time interval, a Probe Timeout (PTO) occurs, prompting the sender to transmit one or two probe datagrams. These probe datagrams must contain ack-eliciting packets, defined in [Iyengar and Thomson, 2021] as QUIC packets containing frames other than `ACK`, `PADDING`, and `CONNECTION_CLOSE`, which trigger the recipient to send an acknowledgment.

In the observed example, however, the probe packets contain only `PADDING` frames and thus should not be ack-eliciting. Nevertheless, the packets are acknowledged immediately, as seen from the timestamps. The most plausible explanation is that these packets also include an `IMMEDIATE_ACK` frame, which is an ack-eliciting frame type defined in [Iyengar et al., 2025], an RFC draft extending QUIC. Quinn already implements `IMMEDIATE_ACK`, but these frames are not currently logged in qlog, as they are not (yet) part of the QUIC qlog specification.

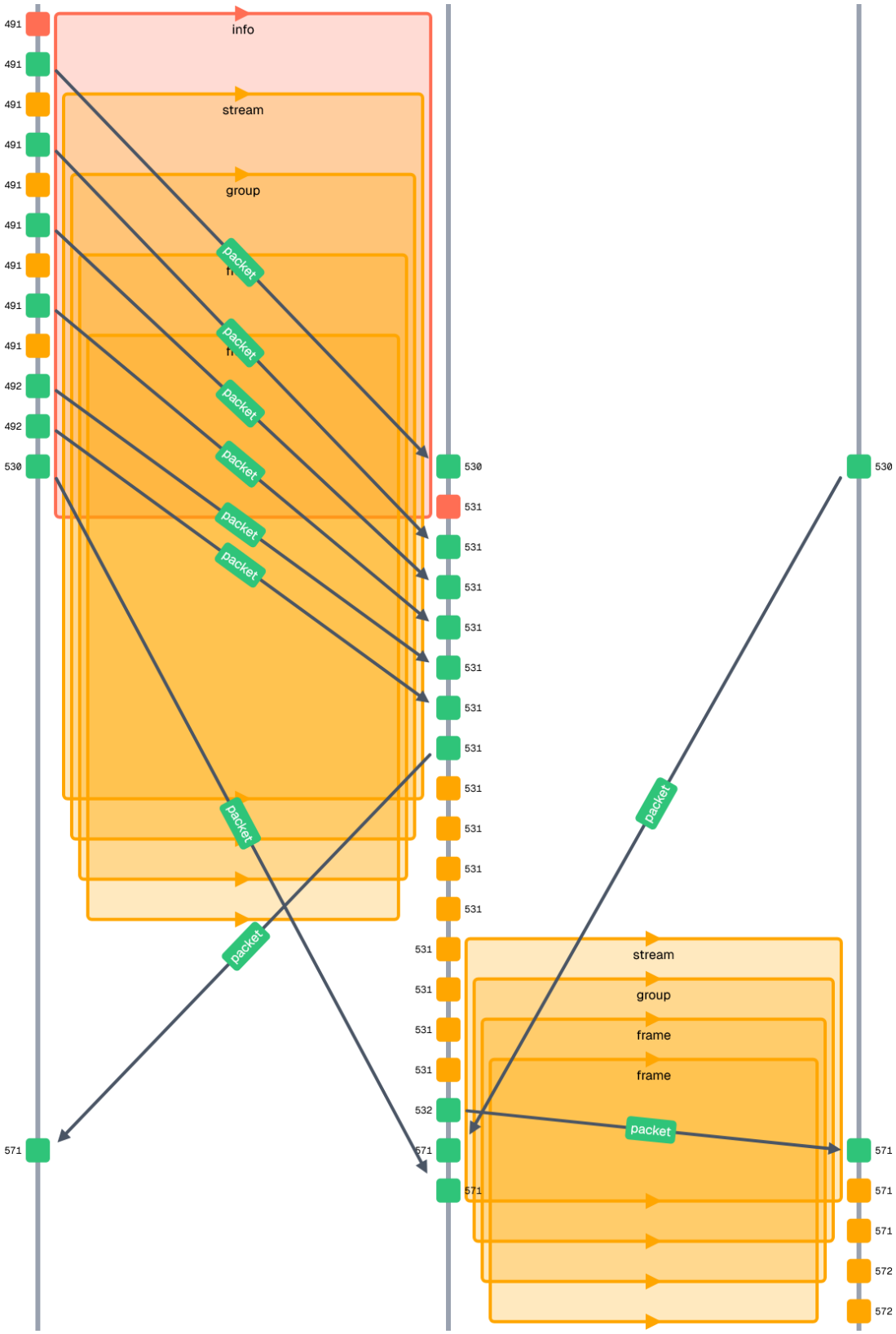


Figure 5.1: The left connection sends several small QUIC packets, the right connection groups them together in one QUIC packet

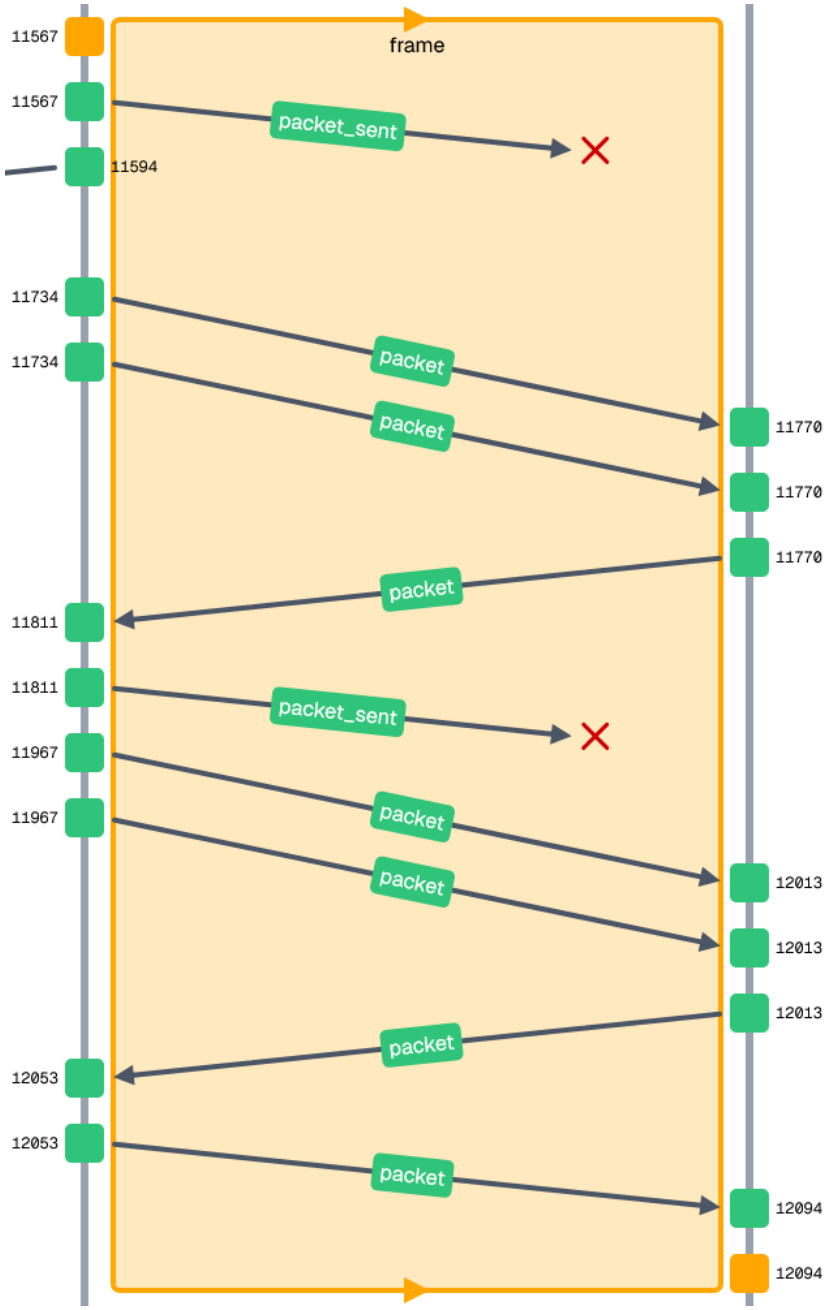


Figure 5.2: Packet loss occurring twice for the same data

Packet loss during MoQ message transmission significantly increases the latency of these messages. While QUIC reliably retransmits lost data, MoQ must wait until the missing data arrives. Consequently, QUIC's measured latency remains relatively stable, whereas MoQ's latency depends on how quickly QUIC recovers from the loss. This effect is visible in Figure 5.2, where all QUIC packets have latencies of around 40 milliseconds, while the corresponding Frame block spans approximately 500 milliseconds.

The latency charts make this effect even clearer. As shown in Figure 5.3, packet loss during MoQ message transmission is visible as distinct latency spikes. These spikes indicate the added time MoQ must wait for QUIC's retransmission process to complete.

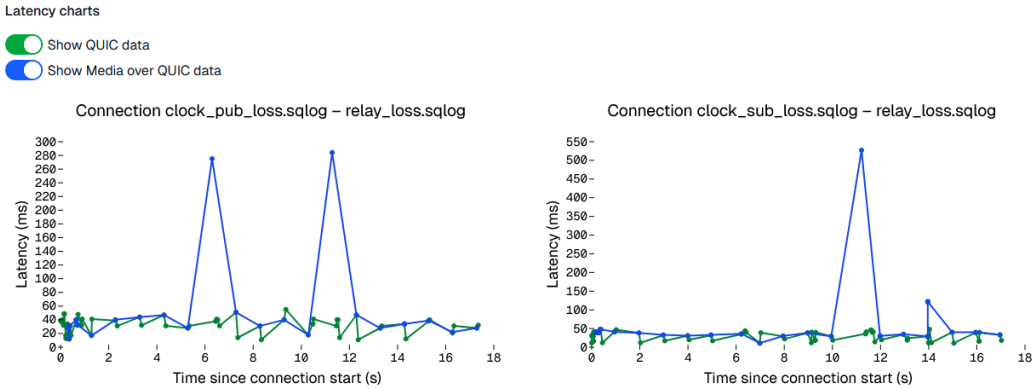


Figure 5.3: Increased MoQ latency during packet loss

Similar behavior is observed in the other two demos when connections are terminated abruptly at the end (by pressing Ctrl + C in the console). In the first demo, the final Frame sent by the relay to the subscriber is never received. The QUIC packet carrying this Frame is marked as lost because the subscriber closed the connection before the packet arrived. Approximately 140 ms later, the relay transmits two QUIC packets containing three `PADDING` frames and, presumably, an `IMMEDIATE_ACK` frame. When no acknowledgment is received within roughly 260 ms, the relay retransmits two new QUIC packets with the same frames. This process repeats a third and final time after about 510 ms without acknowledgment. The third instance of this process is the final one in this case, since the relay was closed manually as well, not necessarily because Quinn limits it to three attempts. Note that the time interval between each instance of this process approximately doubles with each repetition.

5.3 Network Graph Connections

The network graph provides a quick visual overview of the network's structure. One practical use case is comparing the actual network layout to the expected topology. For example, Figure 5.4 shows a network consisting of a small subnet with two nodes, a larger subnet with seven nodes, and a single node not connected to any other nodes. If the unconnected node were supposed to be linked to the relay of the small subnet, the network graph would make this immediately apparent, allowing more in-depth debugging to follow.

Currently, it is not possible to display the events of an individual node in the sequence diagram; only connections can be visualized. Adding node-level visualizations to the sequence diagram could enhance debugging by revealing, for instance, that connection attempts were made but the packets were never received.

A current limitation is that the network graph does not distinguish between multiple connections between the same two nodes. An edge is drawn if there is at least one connection between the nodes, and all such connections are represented as a single edge. As a result, the sequence diagram will display all connections between those nodes. On the other hand, the latency

charts will only display one of them. This is an issue that can be improved by adapting the graph to show multiple connections.

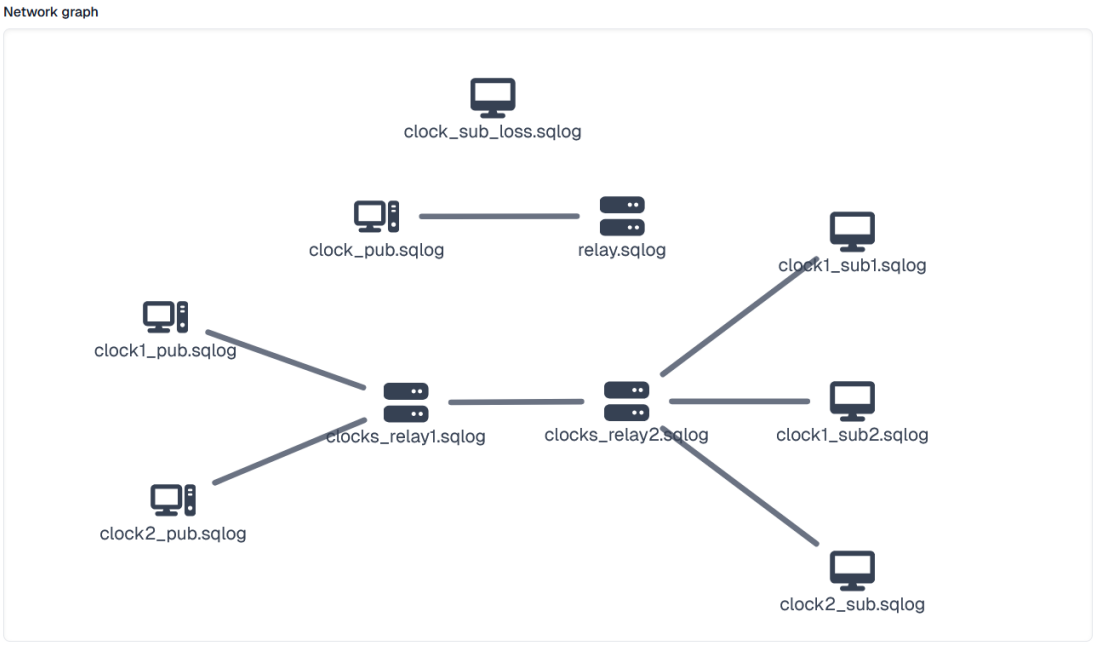


Figure 5.4: Network graph showing two separate subnets and an unconnected node (nodes are manually rearranged; not the default layout)

5.4 Multiple MoQ Sessions

When inspecting the sequence diagram between the two relays in the third demo, there seemed to be a bug with the visualization of the MoQ message event blocks. A closer examination revealed that the issue was caused by events from three distinct MoQ sessions, corresponding to three separate QUIC connections—overlapping in time. The current implementation does not account for multiple simultaneous sessions when calculating block widths or determining event ordering.

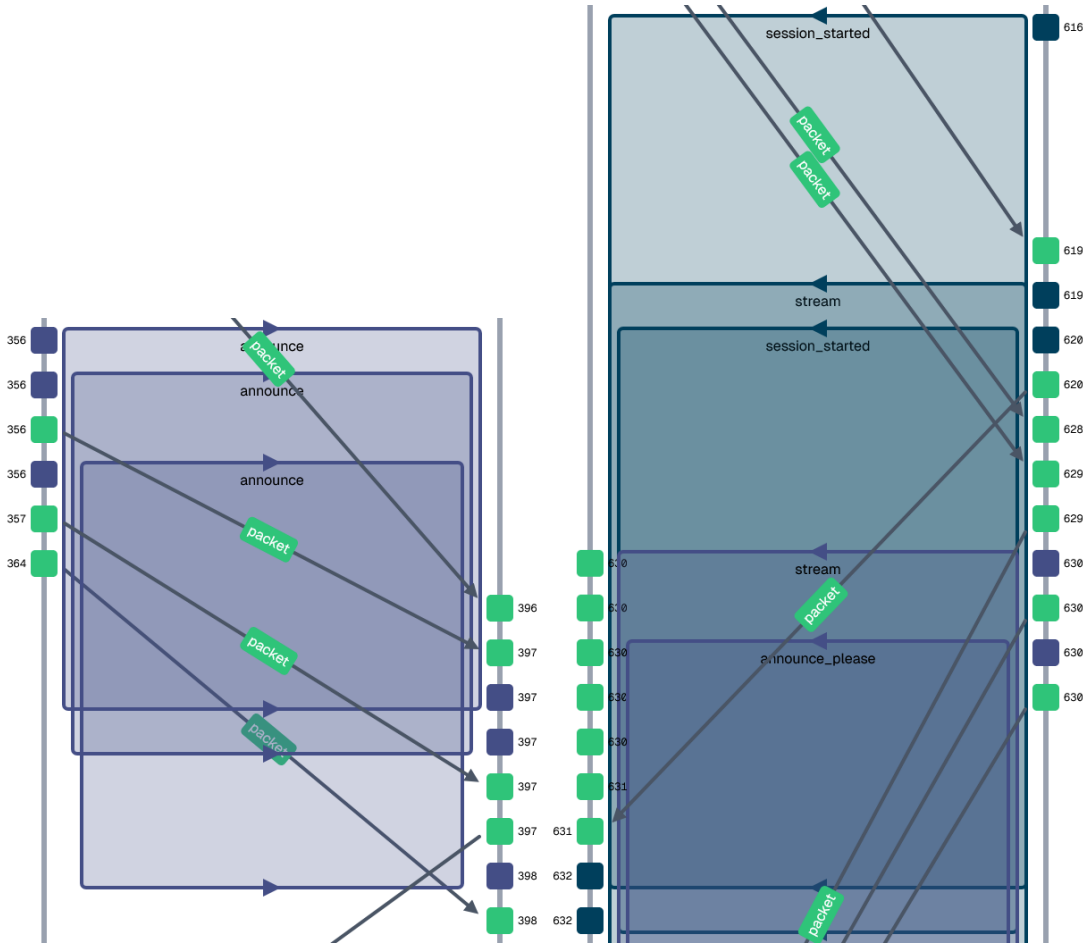
As a result, some overlapping blocks are rendered with identical widths, while certain QUIC events end up positioned beneath MoQ event blocks, making them difficult to interact with. Both of these issues are illustrated in Figure 5.5.

Addressing this problem will require improvements both in the network graph, as discussed in Section 5.3, and in the sequence diagram's rendering logic to properly handle multiple sessions when the user chooses to display more than one.

5.5 Relay Functionality

The functionality of the relays is most clearly demonstrated in the third demo. In this scenario, there are two publishers, with subscribers to both, requiring the relays to forward application data from both publishers.

The relays' behavior becomes clear when viewing the end-to-end events from the first publisher to the first subscriber, as shown on the network graph in Figure 4.6. First, the relays establish a connection with each other. Next, the publisher initiates a MoQ session with the first relay and announces the available Tracks. The first relay forwards this announcement to the second relay, which in turn sends the same announcement back to the first relay, even though the first



(a) QUIC events obscured beneath MoQ events **(b)** Overlapping MoQ blocks with identical widths

Figure 5.5: Current visualization issues when displaying multiple concurrent MoQ sessions

relay already originated this information. Whether this is intentional or an oversight requires further investigation.

Because the sequence diagram displays all events between two endpoints, announcements from the second publisher are also visible between the two relays, even when the second publisher's connection is not explicitly displayed here. While this behavior may be unwanted when focusing strictly on an end-to-end subscription, it can help analyze relay interactions.

After the announcement phase, the subscriber connects to the second relay and starts a subscription to the first publisher's Track. This subscription request is forwarded over both relays to the publisher. The publisher then sends data in `FRAME` messages, which are forwarded hop-by-hop to the subscriber. Between the relays, the sequence diagram also reveals subscriptions and data from the second publisher. For example, when the first publisher sends a `FRAME`, the second relay forwards two: one from the first publisher and one from the second publisher. This is illustrated in Figure 5.6.

Finally, note that the subscription from the other subscriber to the first publisher is not transmitted between the relays in this case, since the second relay already receives that data and can forward it directly to both subscribers.

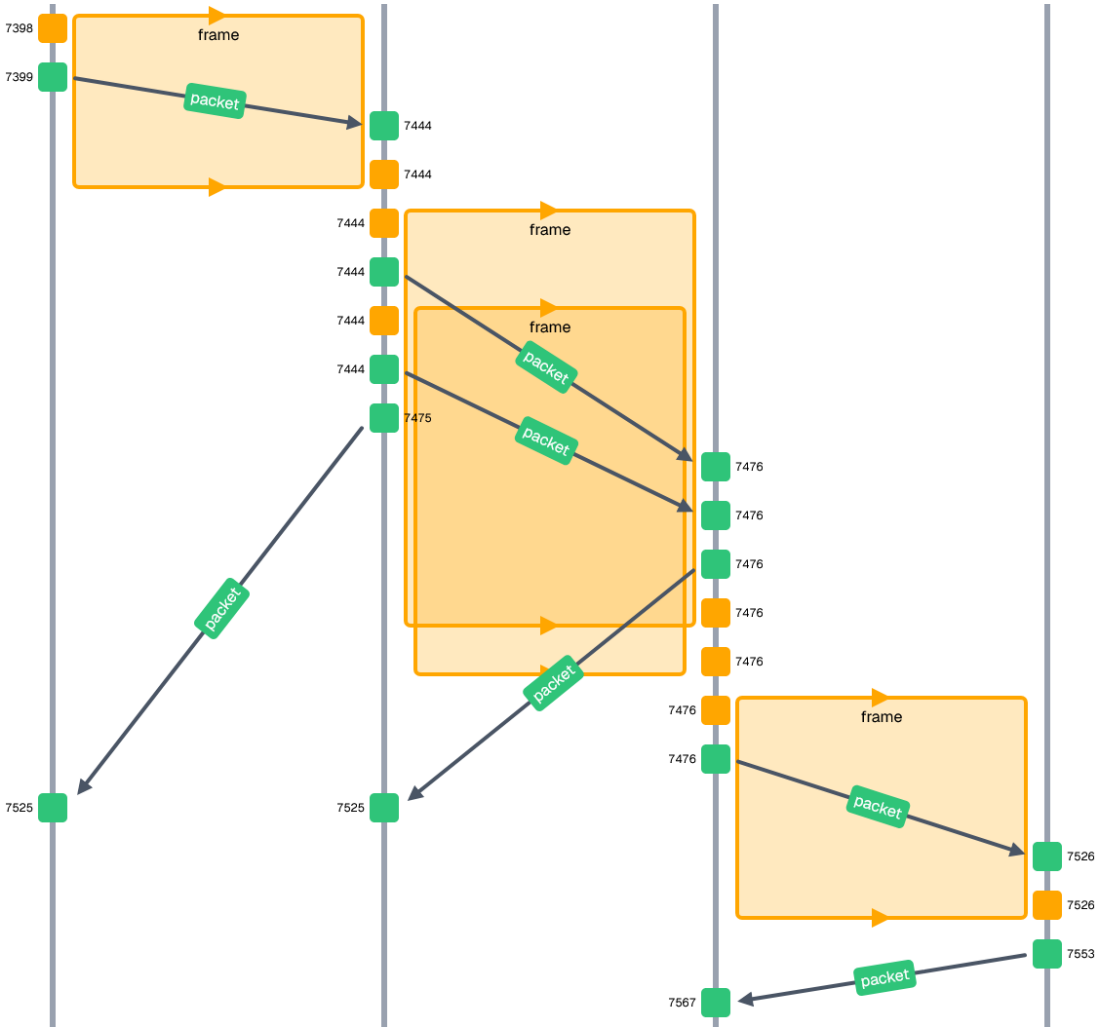


Figure 5.6: End-to-end data flow starting from the publisher, over the relays, towards the subscriber

Chapter 6

Conclusion

This thesis investigated using qlog-based logging and visualization techniques to analyze and debug MoQ traffic. The focus was on visualizing MoQ's scalability, latency behavior, and its use of QUIC to achieve these goals. The developed visualizations were evaluated through multiple demonstration scenarios. The results of these evaluations provide the answers to the research questions.

For clarity, the research questions are restated below:

1. How can qlog-based logging and visualization techniques support the analysis and debugging of MoQ?
   (a) How can MoQ's scalability be effectively represented in visualizations?
   (b) How can MoQ's low-latency support be captured and illustrated?
   (c) How can MoQ's use of QUIC be visualized to highlight protocol behavior?
2. In what ways can visualizations reveal issues in MoQ implementations and deployments?
   (a) How can inefficient packetization be identified and analyzed through visualizations?
   (b) How can packet loss be seen in the visualizations?
   (c) How can topology issues be identified?

To address the first question:
The evaluation demonstrates that qlog-based visualizations can provide both a broad and detailed perspective on MoQ's operation. The network graph gives an immediate overview of the network's topology, scaling effectively from simple two-node setups to more complex deployments containing multiple relays, publishers, and subscribers. Latency charts make it possible to spot patterns over time, capturing not only MoQ's low-latency operation under normal conditions but also deviations caused by retransmissions. Users can zoom and pan to focus on specific time windows for a more granular examination. The sequence diagram offers an event-level view, showing the message flow of a connection, MoQ's interaction with QUIC, and event metadata. Importantly, these tools are interconnected: the network graph can be used to select specific connections for deeper inspection in the latency charts and sequence diagram, allowing a gradual zoom from high-level topology down to individual packets and frames. Together, these tools offer high- and low-level perspectives on MoQ's inner workings, enabling in-depth analysis and debugging.

For the second question:
Inefficient packetization is visible in the sequence diagram when multiple QUIC packets, each carrying only a few bytes, are sent in rapid succession, as described in Section 5.1. Packet loss is revealed when `packet_sent` events occur without being acknowledged, marked by red

crosses in the diagram; when these packets carry MoQ data, corresponding latency spikes also appear in the latency charts, as indicated in Section 5.2. Topology issues are detected via the network graph, which makes missing or unexpected connections quickly noticeable, as shown in Section 5.3.

While the current visualizations already enable valuable insights, they are not without limitations. Improvements could include better handling of multiple MoQ sessions in the sequence diagram (Section 5.4) and better connection differentiation in the network graph (Section 5.3). Nonetheless, the work presented here forms a solid foundation, demonstrating that qlog-based approaches can be a powerful aid in analyzing, debugging, and improving MoQ implementations and deployments.

6.1 Future Work

Future work primarily involves enhancing existing visualizations, logging additional qlog events, and introducing new types of visualizations.

The network graph could be improved by displaying the number of MoQ sessions between any two endpoints and allowing users to select each individual connection. This would give finer control over what is displayed in the sequence diagram and latency charts. Another valuable addition would be the ability to show the end-to-end flow of subscriptions on the graph directly. Such a feature would immediately make it clear which subscribers are connected to which publishers and what subscriptions each relay handles. Providing an option to select specific subscriptions for visualization would also enable users to isolate and examine particular end-to-end flows in greater detail.

The current set of MoQ-specific qlog events is limited to messages exchanged between MoQ endpoints. Expanding this to include events related to adaptive bitrate (ABR) decisions, congestion control state changes, and other protocol-level information would give a more complete view of MoQ's behavior. This additional data could also enable the creation of new visualization types, such as charts correlating bitrate changes with network conditions or visual representations of congestion window dynamics, similar to qvis.

6.2 Self-Reflection

Overall, I am satisfied with the outcome of this project and the insights gained. The visualizations developed meet the core objectives of the thesis, and the evaluation demonstrates their value for analyzing and debugging MoQ. Working closely with the implementation significantly deepened my understanding of various protocols, especially QUIC and MoQ, which proved invaluable when interpreting the logged data. This project was the largest I have worked on so far, and managing its scope was challenging but ultimately rewarding.

At the start, the writing process felt difficult, as I have never considered it one of my strong suits. The hardest part was often translating my thoughts into clear words. However, as the thesis progressed, writing became easier. Initially, I read many papers and RFCs, marking important paragraphs for later reference. In hindsight, it would have been better to immediately integrate these insights into the thesis, as I sometimes forgot the exact reason I marked a passage or the specific point I wanted to emphasize.

Looking back, I would also approach the workflow differently. Too much emphasis was placed on developing the implementation first, followed by writing the thesis. This created a separation between the technical work and the documentation process, meaning that some insights had to be rediscovered or rephrased later. Starting the writing process earlier and working in parallel with the implementation would have allowed knowledge to be captured immediately, making both the thesis writing smoother and the implementation more focused. This parallel

approach might also have revealed missing features or requirements earlier, improving the final result.

Despite these challenges, I am happy with both the technical achievements and the personal growth this project brought. It strengthened my technical expertise, improved my writing, and gave me valuable experience in managing a large and complex research project from start to finish.

Bibliography

[Aboba, 2023] Aboba, B. (2023). WebRTC extended use cases. Technical report, W3C. https://www.w3.org/TR/2023/DNOTE-webrtc-nv-use-cases-20231214/.

[Alvestrand, 2021] Alvestrand, H. T. (2021). Transports for WebRTC. RFC 8835.

[Apple, 2018] Apple (2018). HTTP Live Streaming (HLS) authoring specification for Apple devices. Available online: https://developer.apple.com/documentation/http-live-streaming/hls-authoring-specification-for-apple-devices. Last checked June 17, 2025.

[AppLogic Networks, 2024] AppLogic Networks (2024). The Global Internet Phenomena Report 2024. Available online: https://www.applogicnetworks.com/hubfs/Sandvine_Redesign_2019/Downloads/2024/GIPR/GIPR%202024.pdf. Last checked August 17, 2025.

[AppLogic Networks, 2025] AppLogic Networks (2025). The Global Internet Phenomena Report 2025. Available online: https://www.sandvine.com/hubfs/AppLogic_Networks/Collateral/Global%20Internet%20Phenomena%20Reports/GIPR%202025.pdf. Last checked August 17, 2025.

[Bentaleb et al., 2025] Bentaleb, A., Lim, M., Akcay, M. N., Begen, A. C., Hammoudi, S., and Zimmermann, R. (2025). Toward one-second latency: Evolution of live media streaming. *IEEE Communications Surveys & Tutorials*, pages 1–1.

[Cloudflare, 2025a] Cloudflare (2025a). qlog: qlog data model for QUIC and HTTP/3. Available online: https://github.com/cloudflare/quiche/tree/master/qlog. Last checked August 2, 2025.

[Cloudflare, 2025b] Cloudflare (2025b). quiche: Savoury implementation of the QUIC transport protocol and HTTP/3. Available online: https://github.com/cloudflare/quiche. Last checked August 2, 2025.

[Conviva, 2020] Conviva (2020). Streaming in the Time of Coronavirus. Available online: https://pages.conviva.com/rs/138-XJA-134/images/RPT_Streaming_in_the_Time_of_Coronavirus.pdf. Last checked August 17, 2025.

[Curley, 2025a] Curley, L. (2025a). Media over QUIC - Transfork. Internet-Draft draft-lcurley-moq-transfork-03, Internet Engineering Task Force. Work in Progress.

[Curley, 2025b] Curley, L. (2025b). moq-rs: Rust library for Media over QUIC. Available online: https://github.com/kixelated/moq. Last checked February 4, 2025.

[D3, 2025] D3 (2025). Getting started. Available online: https://d3js.org/getting-started. Last checked August 4, 2025.

[Diaz et al., 2024] Diaz, D., Stolarz, D., and Aguerre, J. (2024). Toward real-time video streaming over webrtc data channels to support supplementary video codecs and formats in the

browser. In *2024 IEEE International Conference and Expo on Real Time Communications at IIT (RTC)*, pages 39–45. IEEE.

[Digital Samba, 2024] Digital Samba (2024). Understanding the Differences Between HLS and Low-Latency HLS. Available online: https://www.digitalsamba.com/blog/hls-vs-ll-hls. Last checked August 11, 2025.

[Duke et al., 2025] Duke, M., Banks, N., and Huitema, C. (2025). QUIC-LB: Generating Routable QUIC Connection IDs. Internet-Draft draft-ietf-quic-load-balancers-20, Internet Engineering Task Force. Work in Progress.

[Durak et al., 2020] Durak, K., Akcay, M. N., Erinc, Y. K., Pekel, B., and Begen, A. C. (2020). Evaluating the performance of apple's low-latency hls. In *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6. IEEE.

[Flowbite, 2025] Flowbite (2025). Flowbite - Tailwind CSS component library. Available online: https://flowbite.com/docs/getting-started/introduction/. Last checked August 4, 2025.

[Grigorik, 2013] Grigorik, I. (2013). *High Performance Browser Networking: What every web developer should know about networking and web performance.* " O'Reilly Media, Inc.".

[Grispen, 2025] Grispen, D. (2025). moq-vis: Media over QUIC Visualizations. Available online: https://dannyg-1952723.github.io/moq-vis/. Last checked August 8, 2025.

[Internet Engineering Task Force, 2022] Internet Engineering Task Force (2022). Media over QUIC Working Group Charter. Retrieved from https://datatracker.ietf.org/group/moq/about/.

[Iyengar and Swett, 2021] Iyengar, J. and Swett, I. (2021). QUIC Loss Detection and Congestion Control. RFC 9002.

[Iyengar et al., 2025] Iyengar, J., Swett, I., and Kühlewind, M. (2025). QUIC Acknowledgment Frequency. Internet-Draft draft-ietf-quic-ack-frequency-11, Internet Engineering Task Force. Work in Progress.

[Iyengar and Thomson, 2021] Iyengar, J. and Thomson, M. (2021). QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000.

[Jennings et al., 2024] Jennings, C., Boström, H., Castelli, F., and Bruaroey, J.-I. (2024). WebRTC: Real-time communication in browsers. W3C recommendation, W3C. Retrieved from https://www.w3.org/TR/2024/REC-webrtc-20241008/.

[Jesup et al., 2021] Jesup, R., Loreto, S., and Tüxen, M. (2021). WebRTC Data Channels. RFC 8831.

[Karthikeyan, 2022] Karthikeyan, S. (2022). Introduction to Low Latency Streaming with HLS. Available online: https://www.100ms.live/blog/hls-low-latency-streaming. Last checked May 13, 2025.

[Kurose and Ross, 2016] Kurose, J. and Ross, K. (2016). *Computer Networking: A Top-Down Approach.* Pearson, 7 edition.

[Learn UI Design, 2018] Learn UI Design (2018). Palette Generator. Available online: https://www.learnui.design/tools/data-color-picker.html. Last checked August 6, 2025.

[Luke Curley, 2022] Luke Curley (2022). Distribution @ Twitch. Available online: https://quic.video/blog/distribution-at-twitch. Last checked April 30, 2025.

[Luke Curley, 2023a] Luke Curley (2023a). QUIC's (hidden) Super Powers. Available online: https://quic.video/blog/quic-powers. Last checked June 29, 2025.

[Luke Curley, 2023b] Luke Curley (2023b). Replacing HLS/DASH. Available online: https://quic.video/blog/replacing-hls-dash. Last checked April 30, 2025.

[Luke Curley, 2023c] Luke Curley (2023c). Replacing WebRTC. Available online: https://quic.video/blog/replacing-webrtc/. Last checked January 20, 2025.

[Luke Curley, 2024a] Luke Curley (2024a). Fork. Available online: https://quic.video/blog/transfork. Last checked July 7, 2025.

[Luke Curley, 2024b] Luke Curley (2024b). The MoQ Onion. Available online: https://quic.video/blog/moq-onion. Last checked June 29, 2025.

[Marx, 2024] Marx, R. (2024). qvis: tools and visualizations for QUIC and HTTP/3. Available online: https://qvis.quictools.info/#/files. Last checked August 13, 2025.

[Marx et al., 2024] Marx, R., Niccolini, L., Seemann, M., and Pardue, L. (2024). HTTP/3 qlog event definitions. Internet-Draft draft-ietf-quic-qlog-h3-events-09, Internet Engineering Task Force. Work in Progress.

[Marx et al., 2025a] Marx, R., Niccolini, L., Seemann, M., and Pardue, L. (2025a). qlog: Structured Logging for Network Protocols. Internet-Draft draft-ietf-quic-qlog-main-schema-11, Internet Engineering Task Force. Work in Progress.

[Marx et al., 2025b] Marx, R., Niccolini, L., Seemann, M., and Pardue, L. (2025b). QUIC event definitions for qlog. Internet-Draft draft-ietf-quic-qlog-quic-events-10, Internet Engineering Task Force. Work in Progress.

[Marx et al., 2020] Marx, R., Piraux, M., Quax, P., and Lamotte, W. (2020). Debugging quic and http/3 with qlog and qvis. In *Proceedings of the 2020 Applied Networking Research Workshop*, pages 58–66.

[MDN web docs, 2024] MDN web docs (2024). WebCodecs API. Available online: https://developer.mozilla.org/en-US/docs/Web/API/WebCodecs_API. Last checked July 3, 2025.

[MDN web docs, 2025a] MDN web docs (2025a). RTCDataChannel. Available online: https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel. Last checked April 10, 2025.

[MDN web docs, 2025b] MDN web docs (2025b). The WebSocket API (WebSockets). Available online: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. Last checked July 3, 2025.

[MDN web docs, 2025c] MDN web docs (2025c). WebRTC connectivity. Available online: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity. Last checked April 2, 2025.

[MDN web docs, 2025d] MDN web docs (2025d). WebTransport API. Available online: https://developer.mozilla.org/en-US/docs/Web/API/WebTransport_API. Last checked July 3, 2025.

[Nandakumar et al., 2025] Nandakumar, S., Vasiliev, V., Swett, I., and Frindell, A. (2025). Media over QUIC Transport. Internet-Draft draft-ietf-moq-transport-13, Internet Engineering Task Force. Work in Progress.

[Next.js, 2025] Next.js (2025). Next.js Docs. Available online: https://nextjs.org/docs. Last checked August 4, 2025.

[Ochtman and Saunders, 2025] Ochtman, D. and Saunders, B. (2025). Quinn: Async-friendly QUIC implementation in Rust. Available online: https://github.com/quinn-rs/quinn. Last checked April 30, 2025.

[OptiView, 2018] OptiView, D. (2018). The importance of low latency in video streaming. Available online: https://optiview.dolby.com/resources/blog/streaming/the-importance-of-low-latency-in-video-streaming/. Last checked June 16, 2025.

[Pantos and May, 2017] Pantos, R. and May, W. (2017). HTTP Live Streaming. RFC 8216.

[Pardue and Engelbart, 2025] Pardue, L. and Engelbart, M. (2025). MoQ qlog event definitions. Internet-Draft draft-pardue-moq-qlog-moq-events-01, Internet Engineering Task Force. Work in Progress.

[Pauly et al., 2022] Pauly, T., Kinnear, E., and Schinazi, D. (2022). An Unreliable Datagram Extension to QUIC. RFC 9221.

[Perkins et al., 2021] Perkins, C., Westerlund, M., and Ott, J. (2021). Media Transport and Use of RTP in WebRTC. RFC 8834.

[Petrangeli et al., 2018] Petrangeli, S., Pauwels, D., van der Hooft, J., Wauters, T., De Turck, F., and Slowack, J. (2018). Improving quality and scalability of webrtc video collaboration applications. In *Proceedings of the 9th ACM Multimedia Systems Conference*, pages 533–536.

[Petrangeli et al., 2019] Petrangeli, S., Pauwels, D., Van Der Hooft, J., Žiak, M., Slowack, J., Wauters, T., and De Turck, F. (2019). A scalable webrtc-based framework for remote video collaboration applications. *Multimedia Tools and Applications*, 78:7419–7452.

[React, 2024] React (2024). Quick Start. Available online: https://react.dev/learn. Last checked August 4, 2025.

[Rust, 2025] Rust (2025). Features. Available online: https://doc.rust-lang.org/cargo/reference/features.html. Last checked August 2, 2025.

[Samis, 2020] Samis, B. (2020). Back to basics: GOPs explained. Available online: https://aws.amazon.com/blogs/media/part-1-back-to-basics-gops-explained/. Last checked July 17, 2025.

[Stewart et al., 2022] Stewart, R. R., Tüxen, M., and karen Nielsen (2022). Stream Control Transmission Protocol. RFC 9260.

[Synchronous, 2021] Synchronous (2021). What Is HLS (HTTP Live Streaming)? How HLS Works? Available online: https://www.synopi.com/hls-http-live-streaming. Last checked May 1, 2025.

[Tailwind CSS, 2025] Tailwind CSS (2025). Get started with Tailwind CSS. Available online: https://tailwindcss.com/docs/installation/using-vite. Last checked August 4, 2025.

[Tao, 2022] Tao, C. (2022). clumsy: clumsy makes your network condition on Windows significantly worse, but in a managed and interactive manner. Available online: https://github.com/jagt/clumsy. Last checked August 7, 2025.

[Voximplant, 2020] Voximplant (2020). An Introduction to Selective Forwarding Units. Available online: https://voximplant.com/blog/an-introduction-to-selective-forwarding-units. Last checked April 8, 2025.

[West, 2024] West, J. (2024). Jake Paul vs. Mike Tyson draws 60 million households on Netflix amid streaming issues. Available online: https://www.nytimes.com/athletic/5926460/2024/11/16/jake-paul-mike-tyson-ratings-viewers-netflix/. Last checked August 18, 2025.

[Williams, 2015] Williams, N. (2015). JavaScript Object Notation (JSON) Text Sequences. RFC 7464.

[Yoss, 2022] Yoss, J. (2022). WebRTC vs. RTMP: Which Protocol Is Best for Your Video Deployment? Available online: https://www.wowza.com/blog/webrtc-vs-rtmp-which-protocol-is-best-for-your-video-deployment. Last checked May 31, 2025.

[Yuzzit, 2025] Yuzzit (2025). RTMP: Understanding this protocol, still essential in 2025. Available online: https://www.yuzzit.video/en/resources/rtmp-streaming. Last checked May 31, 2025.

[Zenoni, 2025] Zenoni, I. (2025). Complete Guide to RTMP: The Real-Time Messaging Protocol. Available online: https://www.wowza.com/blog/rtmp. Last checked May 31, 2025.

[Zhang et al., 2021] Zhang, B., Teixeira, T., and Reznik, Y. (2021). Performance of low-latency http-based streaming players. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pages 356–362.

[Zhao, 2022] Zhao, D. (2022). How we built a globally distributed mesh network to scale WebRTC. Available online: https://blog.livekit.io/scaling-webrtc-with-distributed-mesh/. Last checked April 8, 2025.