UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Enhancing MoQ Visibility Using a Modular Logging Infrastructure

Yannick Goedhuys

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

Enhancing MoQ Visibility Using a Modular Logging Infrastructure

Yannick Goedhuys
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

Acknowledgments

I would like to begin by expressing my sincere gratitude to Mr. Vandersanden, my supervisor, for his invaluable guidance and unwavering support throughout this work. His willingness to meet almost weekly to discuss and refine ideas has been instrumental in shaping the outcome of this thesis.

I am equally grateful to my promoter, Prof. Dr. Quax, and my co-promoter, Prof. Dr. Lamotte, for granting me the opportunity to explore the Media over QUIC protocol and for their constructive feedback during the development of the system.

My heartfelt thanks also go to my family for their continuous encouragement and to my friends for their much-needed moral support throughout this journey.

Finally, I wish to acknowledge the MoQ working group for their dedication to the development of the protocol, their helpful responses to my questions, and the insights they provided. A special thanks to Luke Curley for initiating the working group and laying the foundation for this field of study.

Abstract

As the landscape of media streaming evolves, emerging protocols such as Media over QUIC (MoQ) present new opportunities to address challenges inherent to existing solutions like WebRTC and HTTP Adaptive Streaming. While these traditional methods either deliver low latency at the cost of scalability or offer scalability with increased latency, MoQ aims to provide a balanced approach, leveraging QUIC's multiplexed and reliable transport features combined with publish-subscribe models and prioritisation.

However, the adoption of MoQ faces hurdles, notably the complexity in debugging and monitoring due to its dynamic structure and asynchronous control messaging. To overcome these challenges, this thesis introduces a scalable, web-based logging and visualisation framework explicitly designed for MoQ. Built upon the structured qlog format, initially developed for QUIC and HTTP/3, this system is extended to support MoQ-specific semantics, significantly enhancing observability and operational insight into protocol behaviours.

This thesis presents a comprehensive approach to enhancing observability for MoQ by designing and implementing a structured logging and analysis system. A key contribution is the definition of a qlog extension schema tailored to capture the semantics of MoQ. This extension enriches standard qlog data by introducing structured representations of protocol-specific events, including parsing and creation metadata, which enables consistent and meaningful analysis of MoQ sessions across client and relay contexts. Building on this foundation, the thesis proposes a modular, scalable system architecture comprising a client-side instrumentation interface, a high-throughput backend for log ingestion and storage, and an interactive visualisation frontend. This system supports real-time feedback and extensibility, making it suitable for both development and experimental analysis. Functional and architectural evaluations demonstrate that the proposed system significantly improves debugging capabilities, facilitates protocol validation, and enhances the overall comprehension of MoQ dynamics. By enabling visibility into previously opaque behaviours, the system lays the groundwork for standardised observability in next-generation media transport protocols.

Samenvatting

Dit werk onderzoekt het ontwerp en de implementatie van een observability systeem, specifiek ontwikkeld voor het Media over QUIC (MoQ) protocol. MoQ is een relatie nieuw streamingprotocol dat voordelen combineert van traditionele HTTP Adaptive Streaming (HAS) en WebRTC. Hoewel HAS bekend staat om schaalbaarheid via content delivery networks (CDN's), is de latency te hoog voor interactieve toepassingen. WebRTC biedt lage latency, maar is dan weer moeilijk schaalbaar voor grote aantallen gebruikers. MoQ tracht een balans te vinden tussen lage latency en hoge schaalbaarheid door gebruik te maken van het QUIC-transportprotocol, aangevuld met een flexibele publish-subscribe architectuur.

Het observability systeem, ontworpen in dit werk, is bedoeld voor ontwikkelaars en onderzoekers meer inzicht te geven in het gedrag van MoQ-sessies. Hiervoor is gebruik gemaakt van qlog, een gestructureerd en uitbreidbaar logformaat dat oorspronkelijk ontwikkeld is voor QUIC en HTTP/3, maar in deze thesis uitgebreid wordt om MoQ-specifieke gebeurtenissen vast te leggen.

Het Media over QUIC protocol

MoQ is een transport protocol dat ontworpen in om mediagegevens op een schaalbare en efficiënte manier te verspreiden via QUIC en gebruik te maken van de voordelen die QUIC te bieden heeft. Het grootste voordeel van QUIC, wat ook bijdraagt tot het verkrijgen van deze lage latency, is het stream multiplexen. Dankzij deze eigenschap kunnen meerdere onafhankelijke datastromen gelijktijdig over één enkele verbinding verlopen zonder dat vertragingen in de ene stroom impact hebben op de andere, iets wat bij TCP wel het geval is.

Op dit onderliggende QUIC-transport bouwt MoQ een eigen hiërarchisch gegevensmodel op. De fundamentele bouwsteen is het object: een afgebakend blok mediadata zoals een videoframe of een stukje audio. Objecten worden gegroepeerd in subgroepen, die op hun beurt worden samengebracht in grotere groepen. Deze structuur maakt het mogelijk om mediagegevens logisch te organiseren en op efficiënte wijze te verzenden. Een reeks gerelateerde objecten vormt een track, bijvoorbeeld een audiotrack of videotrack, dat uniek geïdentificeerd wordt binnen een namespace.

MoQ gebruikt een publish-subscribe-architectuur waarin producenten tracks kunnen aankondigen en publiceren, terwijl abonnees kunnen aangeven welke tracks zij willen ontvangen. Dit mechanisme wordt geregeld via de MOQT-laag (Media Over QUIC Transport), die verantwoordelijk is voor sessieopbouw, protocolonderhandelingen, en het versturen van controleberichten zoals `CLIENT_SETUP`, `SUBSCRIBE`, `FETCH`, en `ANNOUNCE`. Hiermee kunnen zowel live als on-demand mediastreams ondersteund worden, wat het protocol flexibel inzetbaar maakt.

Een typische MoQ setup wordt uitgebeeld in Figuur 1. Dit proces begint met het vastleggen van een QUIC connectie. Hierna zal de client een setup bericht uitsturen met alle nodige protocolinformatie, zoals bijvoorbeeld het versienummer dat kan gebruikt worden. De pusblisher zal hier op antwoorden en de MoQ connectie hiermee vast leggen. Vervolgens abboneert de client op de announce berichten van de publisher. Deze bevatten de streams die de publisher

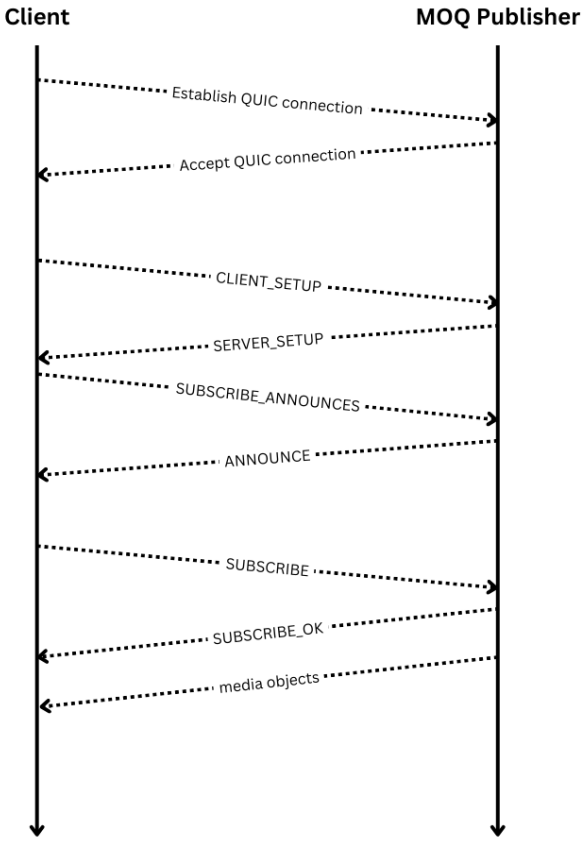


Figure 1: Typische MoQ sessie startup. Dit begint met de QUIC connectie vast te leggen, gevolgd door enkele setup berichten. Hierna zal de client subscriben op de announce track die informatie bevat over de beschikbare streams van de publisher. Dit wordt opgevolgd door het subscriben op een bepaalde stream waarna de publisher begint met het sturen van media objecten naar de client.

aanbiedt waarmee de client dan aan de slag kan gaan om te kiezen welke hij wil ontvangen. Wanneer deze gekozen heeft stuurt hij een subscribe bericht naar de publisher met de nodige informatie waarna hij een bevestiging terug krijgt, kort gevolgd door het begin van de media berichten.

Om mediadata consistent te structureren en interoperabiliteit te verbeteren maakt MoQ gebruik van het WARP-formaat. Dit formaat definieert hoe objecten verpakt worden, inclusief metadata zoals tijdsaanduidingen, codec-informatie en afhanekelijkheden tussen tracks. Via deze informatie kan een subscriber client dan beslissen welke video kwaliteit deze nodig heeft en op die manier aan Adaptive Bitrate Streaming (ABR) doen.

Tot slot voorziet MoQ relays wat tussenliggende knooppunten zijn die fungeren als distributiepunten tussen publishers en subscribers. Deze relays verbeteren de schaalbaarheid en veerkracht van het protocol, en kunnen extra funcionaliteit bieden zoals caching, filtering of prioriteitsafhandeling.

Het qlog logformaat

qlog is een gestandaardiseerd, gestructureerd logformaat dat ontworpen is voor het loggen van nerwerkprotocollen zoals QUIC en HTTP/3. Het biedt een uniforme manier om protocolgebeurtenissen vast te leggen met als doel debugging analyse en visualisatie eenvoudiger en con-

sistenter te maken. Een belangrijk voordeel van qlog is dat het toelaat om inzicht te verkrijgen in de interne werking van protocollen waarvan de zichtbaarheid in het netwerk door encryptie grotendeels onzichtbaar is geworden.

Om structuur te realiseren definieert qlog een gemeenschappelijk, uitbreidbaar schema voor events. Deze bevatten metadata zoals tijdstempels, betrokken streams, pakketnummers, errors en contextuele informatie over de sessie. Hierdoor kunnen logs eenvoudig gedeeld en eanalyseerd worden met generieke tooling.

Een kernconcept in qlog is de opdeling in traces die één specifieke netwerkverbinding representeren. Elke trace bestaat uit een reeks eventen waarbij elke event een welgedefinieerde gebeurtenis beschrijft volgens een bepaald schema. Als standaard serialisatieformaat maakt qlog gebruik van JSON, wat het gemakkelijk leesbaar en verwerkbaar maakt voor zowel mensen als machines. Dit bevordert interoperabiliteit en maakt het mogelijk om visuele tooling zoals qvis of zelfgemaakte dashboard te gebruiken voor real-time analyse.

Het qlog formaat laat expliciet toe uitgebreidt te worden via het toevoegen van namespaces. Voor MoQ breiden we dit formaat uit met de "moq" en "moq-custom" namespace. Respectievelijk staan deze in voor de huidige MoQ event types te beschrijven en eventuele nieuwe events types. Dit laatste is van groot belang aangezien MoQ een snel vorderend protocol is en we op deze manier visuele tools voorzien van de mogelijkheid om steeds voorwaards compatibel te blijven wanneer er nieuwe event types zouden bijkomen.

Het observability systeem

Deze thesis stelt voor het probleem omtrent het waarnemen van MoQ events een schaalbaar gedistribueerd logging systeem voor. Het systeem is opgebouwd uit meerdere componenten zoals zichtbaar in Figuur 2: een instrumentation laag die de developers toelaat in hun eigen code log aanroepingen te maken, een broker die als tussencomponent dient om het systeem schaalbaar te maken, een backend die alle logs gaat ontvangen, verwerken en opslaan en tot slot een visualisatie tool die toelaat de gemaakte logs live te tonen alsook te exporteren en importeren.

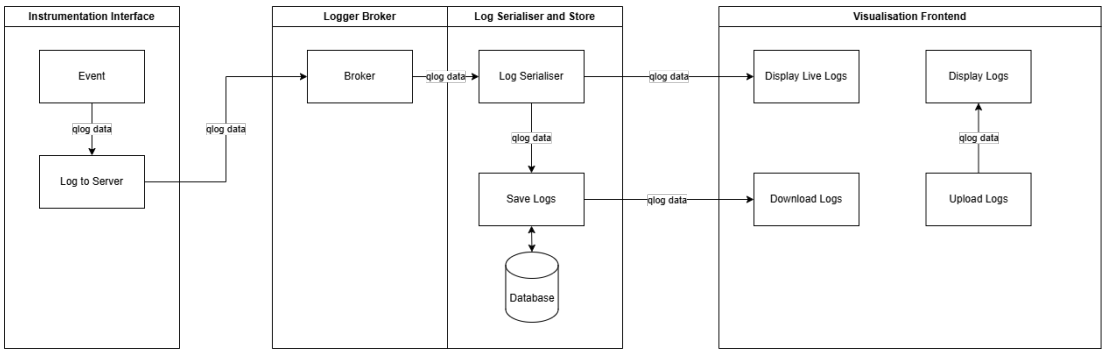


Figure 2: Deze figuur toont de voorgestelde systeemarchitectuur, die bestaat uit vier onderdelen die gecombineerd worden met de reeds bestaande MoQ-implementatie. Deze onderdelen zijn, in volgorde vanaf waar de logging start: de instrumentation-interface, de logger-broker, de log-serialisatie en opslag, en de visualisatie-frontend.

Het systeem is vervolgens ook uitgebouwd in de vorm van een proof-of-concept om te bekijken of het waarde oplevert. Uit deze implementatie is gebleken dat een dergelijk observability systeem toelaat om onderandere inzichten te krijgen in subscribe initialisaties, connectie fouten en latency waardes, waarbij elk van deze situaties zouden aanzienlijk moeilijker zijn moest het systeem er niet zijn.

Contents

Chapter 1

Introduction

As media streaming continues to evolve toward real-time, low-latency, and high-scalability scenarios, new transport paradigms are emerging to address the shortcomings of existing protocols [BLA+23]. Media over QUIC (MoQ), currently under development within the IETF, proposes a flexible and modern approach to media delivery that draws on the strengths of QUIC's multiplexed, reliable transport while introducing concepts such as publish-subscribe models, prioritisation, and relays for scalable distribution.

Structured observability becomes increasingly important in supporting and validating these ideas. Debugging and analysing real-time media flows, especially in the presence of congestion, prioritisation logic, and asynchronous control messages, demands a robust, extensible logging and visualisation framework.

However, as a young and evolving protocol, MoQ still faces significant challenges. Its dynamic publish-subscribe model, prioritisation logic, and reliance on asynchronous control messages complicate implementation and debugging. The lack of standardised tooling makes it difficult to observe protocol behaviour, especially under realistic network conditions, posing a risk of inefficiencies or missed specification issues during standardisation.

To address these challenges, this thesis presents the design and implementation of a scalable, web-based logging and analytics system tailored specifically to MoQ. The system aims to give developers and researchers deeper insight into the behaviour of MoQ sessions, with a strong emphasis on modularity, extensibility, and real-time feedback.

The system is composed of several core components. These include a lightweight client-side logger that can be integrated with MoQ implementations, a backend pipeline capable of high-throughput ingestion, parsing, and storage of structured logs, and a frontend visualisation interface that allows users to explore session behaviour over time. Particular attention is given to managing bursty traffic, high log volume, real-time streaming, and the integration of protocol-specific logging schemas.

In support of this infrastructure, the system builds upon the qlog format, which provides a structured and extensible approach to logging originally developed for QUIC and HTTP/3. A central contribution of this thesis is the development of an extension to qlog that captures MoQ-specific semantics. This extension focuses on control messages, which play a crucial role in session setup, track announcements, subscriptions, and fetching. These messages are essential for understanding the protocol's operation across time and varying network conditions. By providing a structured means to observe such interactions, the extension enables effective debugging, supports experimentation, and facilitates performance analysis. It also establishes a foundation for standardised tooling across MoQ implementations.

The primary focus of this work is the development of a scalable infrastructure that facilitates the practical deployment and evaluation of MoQ. It introduces a system that functions both as a proof-of-concept for analysing MoQ sessions and as a step towards the standardisation of observability tooling in next-generation media protocols. To place this contribution within a broader context, the protocol landscape is also examined. Existing approaches, such as WebRTC and HTTP Adaptive Streaming, are discussed to contextualise the need for MoQ.

Ultimately, this thesis demonstrates the value of dedicated observability infrastructure for emerging media protocols. By providing structured insight into protocol behaviour, developers and researchers are empowered to better understand, evaluate, and refine systems built atop MoQ.

The core research question explored throughout this work is how a scalable, web-based logging and visualisation system, built on enriched qlog data, can enhance observability and operational insight for Media over QUIC sessions. This central inquiry is further examined through the following subquestions:

- What are the technical and architectural requirements necessary to implement a scalable, distributed logging and visualisation system for MoQ streams?
- Is the designed system architecture able to support scalable, live collection and visualisation of MoQ stream events?
- Does visualising MoQ protocol events using enriched qlog data significantly improve the user's understanding of MoQ's behaviour and dynamics?
- Can the proposed system provide actionable insights for protocol developers and operators that would otherwise remain hidden when using traditional logging methods?

To answer the research questions, a solid understanding of multiple domains is required, including transport protocols, structured logging and real-time media streaming. These concepts are brought together in a proof-of-concept observability system, which is subsequently evaluated through a series of functional and architectural tests. The remainder of this thesis is structured as follows:

- **Chapter 2** explores existing media streaming protocols and outlines the motivation for MoQ as a next-generation alternative.
- **Chapter 3** introduces the MoQ protocol in detail, including its key mechanisms, such as object delivery, track prioritisation and session control.
- **Chapter 4** presents the qlog structured logging format and defines how it can be extended to support MoQ-specific events.
- **Chapter 5** examines how the logging system can be scaled to handle high-throughput scenarios, including design trade-offs and technical considerations.
- **Chapter 6** describes the system architecture, focusing on log ingestion, serialisation and visualisation components.
- **Chapter 7** describes a proof-of-concept implementation of the described system and its practical constraints.
- **Chapter 8** evaluates the system in various scenarios to assess its effectiveness and the gaps it fills in the limited MoQ observability landscape.
- **Chapter 9** concludes the thesis with a summary of the findings and a critical reflection on the process, highlighting lessons learned and directions for future work.

Chapter 2

Popular Video Streaming Protocols

Video streaming protocols focus on delivering video and audio over the Internet. As global internet traffic consists primarily of video data, with video streaming alone accounting for 73.74% of internet traffic in America in 2023 according to Sandvine's 2023 Global Internet Phenomena Report [San23], efficient delivery of this content becomes increasingly important. To achieve high efficiency, compromises have to be made. For example, Video On Demand (VOD) can have a high video quality but cannot reach a low latency. Low latency is more critical for live video streaming, where the viewer watches the video at the same time it is recorded. In this chapter, we will look at two popular alternatives for a video streaming protocol: WebRTC and HTTP Live Streaming (HLS). We will look at their architecture, transport protocol and how they handle scalability. We will also look at the problems that arise with these protocols and how they try to solve them. This will give a good overview of the current video streaming landscape and provide a solid foundation for understanding the choices made for MoQ.

2.1 WebRTC

WebRTC [Alv21] is an open-source project that provides real-time communication between browsers and mobile applications through a straightforward API. It was created to be used in web conferencing applications. While often referred to as a protocol, it's more accurately described as a collection of features and protocols working together [SSP15]. Initially designed with components from Global IP Solutions (GIPS), WebRTC gained momentum after Google acquired GIPS in 2010 and subsequently open-sourced the technology. This move fostered collaboration with industry standards bodies like IETF and W3C, ensuring widespread consensus and adoption.

At its core, WebRTC facilitates the transport of video, audio, and other data over peer-to-peer connections. However, its functionality extends beyond simple data transfer. WebRTC adeptly handles various aspects of real-time communication, including:

- **NAT Traversal:** Overcoming challenges posed by Network Address Translation (NAT) to establish direct connections between clients.
- **Congestion Control:** Efficiently managing network congestion to maintain smooth and uninterrupted streams.
- **Security:** Implementing robust security measures to protect the integrity and confidentiality of transmitted data.

These key aspects, along with its open-source nature and extensive support, make WebRTC a cornerstone in the realm of real-time communication, powering various applications like video conferencing, live streaming, and more.

2.1.1 Transport

WebRTC, as mentioned before, actually leverages a combination of protocols to achieve real-time communication. At its core, WebRTC utilises the Real-time Transport Protocol (RTP) [SCFJ03] over UDP for the efficient delivery of video and audio data. RTP's use of UDP allows for low-latency transmission (around 150-500ms of latency [GLFGG16]), crucial for real-time applications such as conferencing calls. However, UDP lacks certain features like quality-of-service guarantees and in-order delivery. To address these shortcomings, the Real-time Transport Control Protocol (RTCP) [SCFJ03] works in conjunction with RTP. RTCP provides feedback on network conditions, such as round-trip times and packet loss, enabling applications to dynamically adjust encoding parameters and sending rates for optimal transport. This allows for a degree of adaptability and monitoring not inherently present in UDP.

The real challenge arises when trying to establish peer-to-peer connections that cross routers that perform Network Address Translation (NAT). Many clients reside behind these types of routers, which mask their internal IP addresses and make direct connections difficult. To overcome this, WebRTC employs the Interactive Connectivity Establishment (ICE) protocol [Ros10]. ICE handles NAT traversal by identifying all possible communication paths between two peers. This involves gathering candidate pairs, which combine local and remote transport addresses. Two key components of ICE are STUN (Session Traversal Utilities for NAT) [MRWM08] and TURN (Traversal Using Relays around NAT) [MRM10]. STUN servers help clients discover their public IP address and port mapping performed by the NAT. However, some NATs, like symmetric NATs, can't be traversed with STUN alone because a different public IP and port are assigned for each unique destination. In these cases, TURN servers act as relays, forwarding traffic between the peers. A client first attempts to connect directly, then uses STUN, and if that fails, it falls back to using a TURN server. These three options are displayed in Figure 2.1.

Beyond video and audio, WebRTC also supports the transmission of arbitrary data using the Stream Control Transmission Protocol (SCTP) [IBM24]. SCTP provides features like partial ordering (full ordering within the same stream) and reliable message delivery, making WebRTC more versatile than just a media streaming protocol. This allows for applications to send metadata, control signals, or other data alongside the media stream.

2.1.2 Transport-wide Congestion Control

Congestion control is crucial for efficient media transmission over networks. Given that WebRTC utilises UDP, a protocol without inherent congestion management capabilities, it implements a transport-wide congestion control (TWCC) algorithm [Goo25]. This algorithm functions by having the receiver measure packet arrival times and transmit these measurements to the sender. The sender subsequently uses this data to adapt its transmission rate, mitigating network congestion. However, while this approach helps maintain low latency, it comes with a trade-off: reducing buffering time increases the risk of playback interruptions if network conditions fluctuate. Unlike HTTP-based streaming protocols that rely on pre-buffering to ensure smooth playback, WebRTC prioritises real-time transmission, which can result in visible stutters or quality drops in unstable network environments.

This way of handling congestion control works well for real-time video streaming because it operates at the transport layer, giving it visibility across all streams (audio, video, and data) in a session. It can allocate bandwidth more efficiently between streams, ensuring critical streams like audio maintain quality while scaling video streams dynamically based on available bandwidth. TWCC leverages precise feedback mechanisms using transport-wide sequence numbers

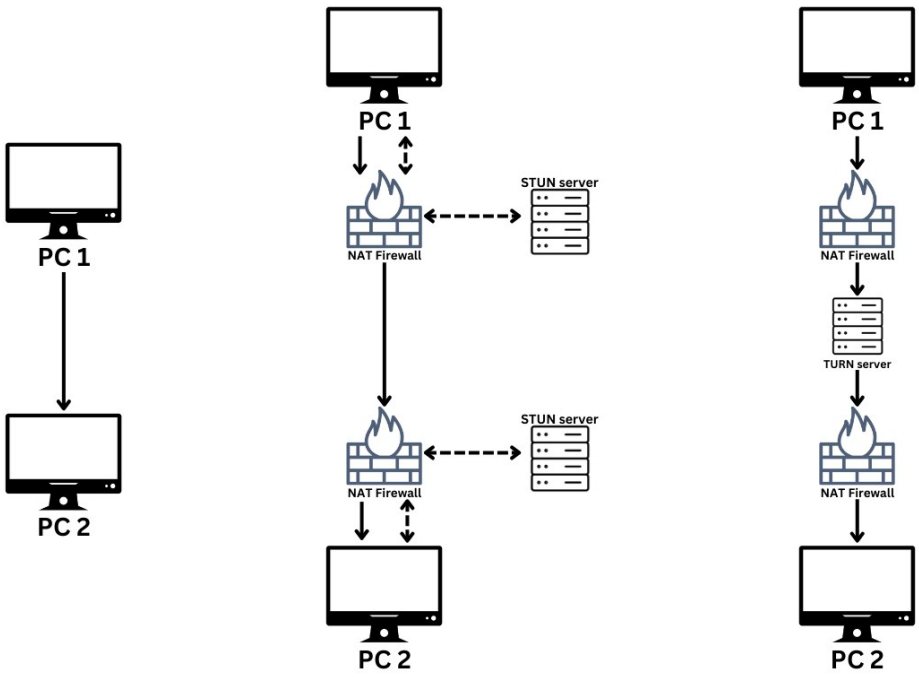


Figure 2.1: This figure illustrates how STUN and TURN servers help establish a connection between PC 1 and PC 2, where both are behind a NAT. A direct connection, the ideal scenario, is shown in the infographic on the left. The middle infographic represents the use of a STUN server to identify clients behind NATs. Here, the clients first send a request to a STUN server (depicted by the striped arrows) to gather their own public IP and port. Finally, the right infographic shows how a TURN server relays traffic between clients when a direct connection is not possible.

and acknowledgements, providing detailed information on packet delays, loss, and arrival times. This allows for more responsive and accurate congestion adjustments.

2.1.3 Problems with WebRTC

While WebRTC has revolutionised real-time communication on the web, it faces several significant challenges that impact its implementation and adoption. These challenges range from technical complexities to deployment issues that developers and organisations must carefully consider.

NAT Traversal Complexity

As mentioned before in Section 2.1.1, the establishment of reliable peer-to-peer connections through NATs represents one of the most significant hurdles in WebRTC implementations [Syl25]. This process demands sophisticated infrastructure, including STUN/TURN servers and complex ICE protocols. Organisations must also account for various firewall configurations and bear additional infrastructure costs for TURN relay servers. These requirements significantly increase deployment complexity and create potential points of failure, particularly within corporate environments that maintain strict security policies.

Browser Implementation Inconsistencies

Despite ongoing standardisation efforts, different browsers implement varying subsets of the WebRTC specification [Tea25]. This variation extends to codec support across platforms and devices, subtle differences in API implementations, and disparate approaches to screen sharing and device access permissions. Media handling capabilities also differ between platforms, forcing developers to implement complex fallback mechanisms and browser-specific optimisations. This necessity for multiple implementation paths significantly increases both development time and maintenance complexity.

Scalability Limitations

Scalability represents a fundamental challenge in WebRTC implementations, particularly when dealing with multi-party communications. The inherent peer-to-peer architecture of WebRTC creates significant technical burdens as the number of participants grows. In a basic mesh topology, each participant must maintain separate peer connections with every other participant, resulting in a $\frac{N(N-1)}{2}$ connection model. This exponential growth in connections creates substantial demands on both client resources and network infrastructure.

The resource requirements become particularly demanding with increased participation. Each peer must encode their media stream separately for every other participant, leading to intensive CPU utilisation and memory overhead. Bandwidth consumption presents another critical challenge, as upstream bandwidth requirements multiply with each additional participant. For instance, a 1 Mbps video stream in a ten-person conference would require 9 Mbps of upload bandwidth from each participant.

To address these scalability limitations, many WebRTC implementations employ Selective Forwarding Units (SFUs) [Gro19]. These intermediate routing servers receive media streams from participants and selectively forward them to other participants, significantly reducing the resource requirements on client devices. SFUs can implement sophisticated stream management strategies, such as selective forwarding based on active speakers and dynamic quality adjustment for individual receivers. However, this centralised architecture introduces additional latency and infrastructure costs that organisations must carefully consider.

2.2 HTTP Adaptive Streaming

HTTP Adaptive Streaming (HAS), leveraging Adaptive Bitrate (ABR) Streaming [Clo], has become the dominant technology for delivering video content over the internet. ABR Streaming is the core mechanism that adapts the video bitrate to the user's network conditions, ensuring a smooth and uninterrupted viewing experience. This is achieved by encoding video into multiple streams of varying bitrates and resolutions. The HAS system then dynamically switches between these streams based on real-time network measurements, selecting the highest possible quality that the current network conditions can support. By utilising standard HTTP infrastructure, HAS is highly scalable and compatible with existing web servers and Content Delivery Networks (CDNs), making it a versatile solution for delivering video to a wide range of devices and network environments.

2.2.1 HTTP Live Streaming

HTTP Live Streaming (HLS) [PM17], initially developed by Apple, is a widely adopted HAS protocol used for both live and on-demand video streaming. Its popularity stems from its use of HTTP, its adaptive bitrate capabilities, and Apple's strong support. HLS encodes video using H.264 [Int24a] or HEVC/H.265 [Int24b] codecs, creating multiple streams with different quality levels. Each stream is divided into fixed-duration segments, typically 2-10 seconds, aligned by keyframes for seamless switching. A primary manifest file lists available streams, their metadata, and links to individual media playlist files. Media playlist files contain URLs to

segment files and segment metadata. The client requests the primary manifest, selects a bitrate based on network conditions, and requests segments from the corresponding media playlist. Client-driven adaptation allows the client to periodically measure network speed (e.g. the time to download the last segment) and switch bitrates if needed. HTTP range requests [Wie23] can be used to download portions of a segment. To address latency issues, especially in live streaming, the community developed Low-Latency HLS (LHLS), which uses chunked transfer encoding to deliver segments in smaller parts as they are encoded. This reduces the time a client waits for a full segment. Apple also introduced their own Low-Latency HLS (LL-HLS), initially using HTTP/2 server push but later adopting a similar chunked encoding approach as LHLS to achieve lower end-to-end latency.

2.2.2 Dynamic Adaptive Streaming over HTTP

Dynamic Adaptive Streaming over HTTP (MPEG-DASH) is an international standard for HAS, offering a flexible and codec-agnostic approach. While sharing many similarities with HLS, DASH distinguishes itself through its standardisation and broader codec support. Similar to HLS, DASH segments video into chunks and uses manifest files to describe available streams. DASH allows the use of any encoding standard, providing greater flexibility than HLS. Like HLS, DASH uses client-side adaptation. Key differences from HLS include its status as an international standard, its support for a broader range of codecs, and its distinct manifest file format.

2.2.3 Transport

Both HLS and DASH rely on HTTP as their application layer protocol, utilising TCP for transport. Video content is prepared for transportation by encoding it with codecs such as H.264, HEVC/H.265, or others allowed by DASH. This encoded content is divided into segments, which are standalone media files containing compressed audio and video data. These segments are aligned across all bitrates to facilitate seamless bitrate switching. A primary manifest file is requested by the player, listing available streaming versions and metadata. Each bitrate has its own media playlist file containing all segments in order, with URLs pointing to the segment files. These files also contain segment metadata. Segments and playlists are hosted on web servers or CDNs. Video retrieval begins with requesting the primary manifest, followed by selecting an optimal bitrate and requesting the corresponding media playlist. Segments are fetched sequentially, and playback starts once enough data is buffered. Client-driven adaptation involves periodic network speed measurements, allowing for bitrate switching based on bandwidth changes.

2.2.4 Problems with HTTP Adaptive Streaming

A significant challenge with HAS, mainly from using TCP, is head-of-line (HOL) blocking [Ste19]. This issue is particularly problematic for real-time applications, where maintaining the most recent video content is crucial for user experience. In such contexts, the priority is to deliver the newest parts of the video stream, ensuring that users receive the most up-to-date data despite potential connection issues. However, TCP's in-order delivery mechanism exacerbates HOL blocking, as the newest packet added to the send buffer must wait for earlier packets to be delivered, regardless of their relevance.

HLS further amplifies these challenges due to its inherent latency, which can exceed 10 seconds in standard implementations. The protocol relies on segment-based delivery, with typical segment durations ranging from two to ten seconds, introducing delays between encoding, uploading, and final playback. Low-latency HLS (LL-HLS) has mitigated some of these issues by using Chunked Transfer Encoding (CTE) and Common Media Application Format (CMAF), reducing latency to a few seconds. However, even LL-HLS cannot match WebRTC's sub-second responsiveness, making it unsuitable for real-time interaction.

Additionally, bitrate adaptation in HLS is constrained by segment boundaries (e.g., every two seconds). If a user is streaming a 1080p video and their bandwidth decreases, they must still download several seconds of high-bitrate content before the stream switches to a lower resolution. This delayed adaptation further worsens latency and impacts the viewing experience.

While MPEG-DASH shares many similarities with HLS, it provides more flexibility in segment handling and allows clients to make smarter adaptation decisions. Unlike HLS, DASH does not inherently require preloading three full segments before playback, enabling faster startup times. Furthermore, DASH implementations can support faster bitrate adaptation, as segment lengths are not fixed by the protocol, allowing for finer control over quality switching. However, like HLS, DASH still relies on TCP and is subject to HOL blocking, though optimisations such as low-latency DASH (LL-DASH) and HTTP/3 adoption are improving its responsiveness.

2.3 Comparison of Alternatives

2.3.1 Latency

Latency is a crucial factor in media streaming, particularly for applications requiring real-time interaction. Different streaming protocols handle latency in distinct ways, influencing their suitability for various use cases. This section examines the importance of latency, typical latency values for WebRTC and HTTP Adaptive Streaming (HAS), and the trade-offs between low latency and buffering. The talking points in this section are summarised at the end in Table 2.1 to give a side-by-side view of the different approaches these protocols take on latency.

Importance of Latency in Media Streaming

Latency refers to the time delay between capturing an event and displaying it to viewers. In real-time applications such as video conferencing, online gaming, and interactive live streaming, low latency is essential to ensure smooth and natural interactions. High latency can lead to disjointed communication, delayed feedback, and reduced user engagement. In contrast, for non-interactive streaming scenarios such as video-on-demand or large-scale broadcasting, latency is less critical, and a moderate delay is often acceptable in exchange for improved playback stability.

Typical Latency Values for WebRTC

WebRTC is designed specifically for real-time communication, prioritising low latency over other considerations. Under typical network conditions, WebRTC achieves latency values between 150 and 500 milliseconds. This ultra-low latency is ideal for interactive applications, enabling natural conversations, instant feedback, and synchronised actions in multiplayer environments. However, maintaining such low latency requires continuous adaptation to network fluctuations, often at the cost of higher processing demands.

Typical Latency Values for HAS Protocols

Unlike WebRTC, HAS protocols, such as MPEG-DASH and HLS, are optimised for scalability and stability rather than real-time responsiveness. Standard implementations of HAS introduce latency in the range of several seconds to tens of seconds due to segment-based video delivery. Traditional HAS workflows require multiple segments to be buffered before playback begins, contributing to this delay.

To address this, low-latency enhancements have been introduced [BLA+25], leveraging technologies such as Chunked Transfer Encoding (CTE) and Common Media Application Format (CMAF). These improvements reduce latency to just a few seconds, making HAS more suitable for near-real-time applications, though it still falls short of the real-time responsiveness required for interactive communication.

Trade-offs Between Low Latency and Buffering

Balancing latency and buffering is a key challenge in media streaming. Low latency minimises the delay between content production and playback, making it crucial for real-time applications. However, it also reduces the time available for buffering, increasing the risk of playback interruptions if network conditions fluctuate. On the other hand, larger buffers improve playback stability by ensuring that content is continuously available, even during network hiccups. This stability comes at the cost of added delay, making it unsuitable for interactive scenarios. The choice of latency versus buffering depends on the application's goals: while real-time interactions prioritise low latency, non-interactive streaming, like on-demand video, can tolerate higher latency for smoother playback.

Finding the right balance ensures a better user experience tailored to the specific needs of the application and its audience.

Aspect	WebRTC	HAS
Importance of Latency	Critical for real-time interaction (video conferencing, gaming). Low latency ensures smooth, engaging experiences.	Less critical for one-way broadcasts. Higher latency is acceptable for stability and reach.
Typical Latency Values	150-500 milliseconds, ideal for interactive applications.	Native HLS: >10 seconds. Low-latency HLS: a few seconds, improved with CTE and CMAF.
Trade-offs	Low latency minimises delays but reduces buffering time, risking interruptions.	Larger buffers ensure stability but introduce latency, which comes at the cost of real-time interaction.

Table 2.1: Summarisation of the differences regarding latency for WebRTC and HAS protocols.

2.3.2 Scalability

Scalability is a crucial consideration when evaluating video streaming protocols, as different architectures handle increasing audience sizes in distinct ways. WebRTC and HTTP Adaptive Streaming follow fundamentally different approaches, each with its own strengths and limitations. The following sections explore how these protocols manage scalability, the challenges they face, and their respective trade-offs in cost and complexity. The talking points in this section are summarised at the end in Table 2.2 to give a side-by-side view of the different approaches these protocols take on scalability.

How WebRTC Handles Scalability: Peer-to-Peer vs. SFU Models

WebRTC supports two primary scalability models: Peer-to-Peer (P2P) and Selective Forwarding Unit (SFU). In the P2P model, media streams are exchanged directly between participants, which works efficiently for small groups but quickly becomes unsustainable for larger ones due to exponential bandwidth usage. The SFU model addresses this by introducing a centralised server that receives and selectively forwards streams, significantly reducing client bandwidth requirements. This enables WebRTC to scale for group calls while maintaining low latency, making it ideal for interactive applications.

Scalability Challenges in WebRTC for Large Audiences

Despite the improvements provided by SFUs, WebRTC faces significant challenges when scaling to large audiences. Supporting hundreds or thousands of viewers requires extensive server

infrastructure to handle real-time media processing. Unlike traditional streaming methods, WebRTC does not inherently support caching or distribution via Content Delivery Networks (CDNs), leading to higher operational costs and complexity. Additionally, ensuring consistent quality across geographically distributed users can be difficult due to network variability.

HTTP Adaptive Streaming's CDN-Friendly Architecture for Massive Scalability

In contrast, HAS protocols are designed for large-scale content distribution through their CDN-friendly architecture. By segmenting video into small chunks, HAS allows content to be cached and delivered efficiently from CDN servers, significantly reducing the load on the origin server. This makes HAS an excellent choice for one-to-many broadcasts, such as live events or webinars, where stability and reach are prioritised over real-time interaction. However, the segmentation and distribution process introduces higher latency, making HAS less suitable for applications requiring immediate responsiveness.

Comparison of Cost and Complexity for Scaling Each Approach

The cost and complexity of scaling WebRTC and HAS differ significantly. WebRTC requires dedicated server infrastructure, particularly when using SFUs, leading to higher operational costs and increased deployment complexity. However, its low-latency performance makes it the preferred option for interactive communication.

HAS, on the other hand, leverages existing CDN infrastructure, making it a more cost-effective solution for large-scale streaming. Its deployment is straightforward, as video content is cached and distributed efficiently. However, the trade-off is increased latency, which limits its suitability for real-time interactions.

Aspect	WebRTC	HAS
Scalability Models	Peer-to-peer for small groups, Selective Forwarding Unit for larger groups. SFU reduces client bandwidth.	CDN-friendly architecture. Segments video into chunks, cached and distributed across CDN servers.
Scalability Challenges	Resource-intensive for large audiences, requires dedicated server infrastructure. Quality of service across distributed users is complex.	Higher latency due to segmentation and distribution. Less suitable for real-time interaction.
Scalability Advantages	SFUs improve scalability for group calls. Low latency is ideal for interactive applications.	Highly scalable for massive audiences, cost-effective due to CDN infrastructure.
Cost and Complexity	Higher operational costs due to dedicated server infrastructure. Complex deployment and maintenance.	Lower cost for large-scale streaming due to CDN usage. Simpler deployment for one-to-many broadcasts.

Table 2.2: Summarisation of the differences regarding scalability for WebRTC and HAS protocols.

In summary, WebRTC offers low-latency communication but faces significant scalability challenges, especially for large audiences. HAS, in contrast, provides cost-effective scalability but with higher latency, making it ideal for passive viewing experiences. Understanding these trade-offs is essential when selecting the appropriate streaming technology for a given use case.

2.3.3 Conclusion

Both WebRTC and HTTP-based live streaming present robust solutions for video streaming, each excelling in distinct areas based on their design and intended use cases. WebRTC stands out with its low latency and real-time communication capabilities, making it ideal for interactive applications such as video conferencing and live collaboration. Its advanced congestion control mechanisms and support for peer-to-peer connections facilitate seamless and responsive user interactions. However, this comes at the cost of increased complexity and limited scalability when catering to large audiences, necessitating additional infrastructure like Selective Forwarding Units (SFUs) to manage extensive participant numbers effectively.

On the other hand, HTTP-based live streaming, including protocols like HLS and DASH, provides unmatched scalability through its compatibility with existing CDNs. By leveraging standard HTTP infrastructure, these streaming methods enable efficient content distribution to massive audiences while ensuring adaptive bitrate streaming, which optimises video quality based on network conditions. This makes them highly effective for delivering high-quality video at scale, particularly for one-to-many broadcast scenarios. However, traditional HTTP-based streaming suffers from higher latency due to segment-based delivery, making it less suitable for real-time applications. While optimisations such as Low-Latency HLS and CMAF have helped reduce delay, they still do not match the immediacy required for interactive use cases.

Media over QUIC (MoQ) aims to bridge the gap between these two approaches by leveraging QUIC's transport-layer advantages, such as multiplexed streams, improved congestion control, and reduced connection establishment overhead. By incorporating WebRTC's low-latency capabilities and HTTP-based streaming's efficient content distribution model, MoQ offers a flexible and scalable solution for modern media delivery. Its ability to support both real-time and near-real-time streaming makes it well-suited for applications ranging from interactive communication to large-scale live broadcasting. Additionally, MoQ's use of QUIC's native congestion control can dynamically adapt to network conditions, optimising both latency and quality without the need for complex server-side infrastructure. As a result, MoQ presents a promising evolution in video streaming, combining the best aspects of WebRTC and HTTP-based live streaming while addressing their respective limitations.

Chapter 3

The Media over QUIC Protocol

The evolution of media transport protocols has led to the adoption of QUIC [IT21] as a foundation for modern streaming technologies. QUIC, designed as a transport layer protocol, offers key advantages over traditional TCP-based streaming by reducing latency, improving congestion control, and enabling better multiplexing of data streams. This makes it particularly attractive for media applications where low latency and reliability are critical.

Media over QUIC (MoQ) leverages the capabilities of the QUIC transport protocol to provide a more efficient and flexible foundation for modern media delivery. It is designed to bridge the gap between traditional HTTP Adaptive Streaming (HAS) protocols and real-time communication solutions like WebRTC, combining the scalability of the former with the low-latency characteristics of the latter.

While HAS protocols such as HLS and DASH support large-scale distribution via segment-based delivery over HTTP/TCP, they inherently introduce playback latency. WebRTC, on the other hand, excels at minimising delay for interactive applications but faces challenges when scaling to broad audiences. MoQ addresses these limitations by introducing a session-based, publish/subscribe model capable of supporting both near-real-time streaming and efficient content distribution.

Operating atop QUIC, MoQ inherits benefits such as connection multiplexing, improved congestion control, encryption, and support for connection migration. These features enhance media transport performance but also pose practical integration challenges. Many existing infrastructures, particularly CDNs, are tightly coupled to HTTP-based delivery models and must be adapted or re-architected to support MoQ traffic, including the deployment of MoQ-aware relays.

MoQ's architectural shift, from client-pull to session-oriented publish/subscribe, also necessitates a rethinking of content delivery strategies and session management. Furthermore, while QUIC provides some inherent NAT traversal support, MoQ lacks a fully standardised peer connectivity solution comparable to ICE, which may complicate deployments in NAT-restricted environments, particularly for peer-to-peer implementations.

Although MoQ is still undergoing standardisation and ecosystem development, its potential to unify responsiveness and scalability makes it a promising candidate for the next generation of media transport. However, realising this potential in practice depends on its ability to integrate with current infrastructures, evolve robust tooling, and gain widespread industry support.

Media over QUIC is built out of several components, as can be seen in Figure 3.1. It can

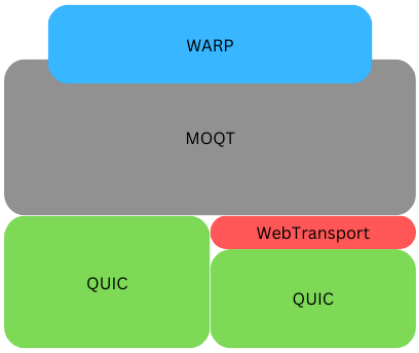


Figure 3.1: This figure demonstrates the building blocks of the Media over QUIC protocol and how every building block relates to the others.

be implemented using either WebTransport or directly with QUIC. WebTransport provides a browser-friendly API for using QUIC streams and datagrams securely within web applications, while direct QUIC implementations allow more control for native applications. Media over QUIC Transport (MOQT) is a media transport protocol built on top of these transport options, defining how media is published, subscribed to, and transmitted efficiently. WARP, then, is the streaming format that MoQ uses in MOQT to define how the streams are structured, packaged, and time-aligned.

This chapter presents the design of the Media over QUIC protocol and explains how it addresses the limitations of current media streaming solutions. The MoQ protocol is still under active development within the IETF, and the discussion in this chapter is based on version 08 of the draft specification [CPN+25], the most recent at the time of writing. The content reflects the protocol's latest mechanisms and architectural choices as defined in that draft. The chapter begins by describing the core concepts of MoQ, including sessions, publish-subscribe semantics, and object-based transport. It then explores key components such as stream management, prioritisation, and relaying. Next, it introduces WARP, a media streaming format designed for MoQ that defines how media is structured, described, and transmitted using catalogues. Finally, the chapter concludes with a comparison between MoQ and established protocols like HAS and WebRTC, illustrating the design improvements that MoQ introduces in terms of latency, scalability, and protocol convergence.

3.1 MOQT Session

MOQT is the transport component of MoQ and builds on QUIC by introducing a structured method for media distribution. It follows a publish-subscribe model, where publishers announce available media tracks, and subscribers request access to them. This model enables efficient data routing, caching, and relay-based distribution. Unlike traditional streaming protocols, MOQT does not rely on HTTP-based segment fetching but instead treats media as discrete objects that can be dynamically prioritised and delivered over QUIC. The structured nature of MOQT allows it to achieve both low latency and scalability by leveraging QUIC's unique transport characteristics. A high-level view of this session can be seen in Figure 3.2, which contains a message sequence chart of the communication between two clients. All aspects of this conversation are explained further in this section.

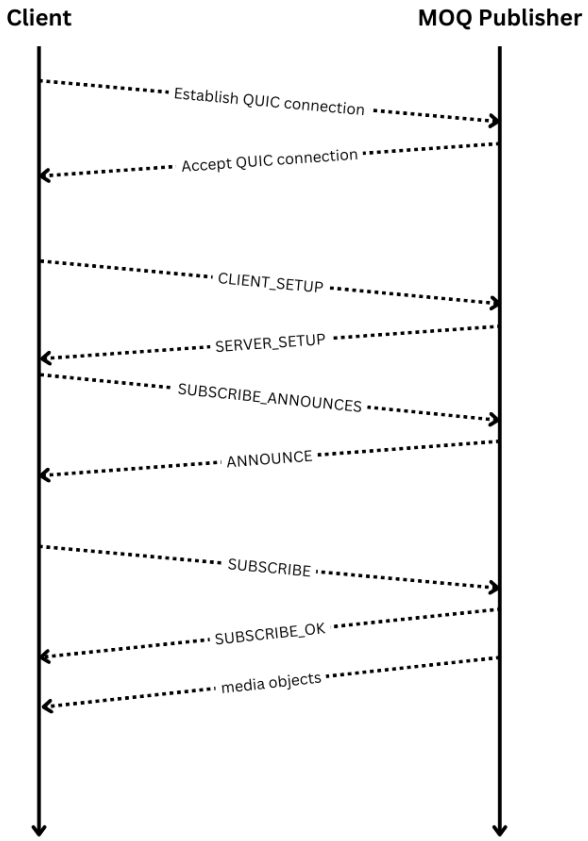


Figure 3.2: This diagram illustrates the process of establishing a MOQT session between a client and a MoQ publisher. It begins with the establishment of a QUIC connection, followed by the exchange of `CLIENT_SETUP` and `SERVER_SETUP` messages to configure the session. The client then sends a `SUBSCRIBE_ANNOUNCES` message to request available tracks, to which the publisher responds with an `ANNOUNCE` message. The client subsequently subscribes to a track using a `SUBSCRIBE` message, and upon receiving a `SUBSCRIBE_OK` from the publisher, media objects are transferred to the client. This sequence highlights the structured and efficient communication protocol employed by MOQT for media distribution.

3.1.1 Session Setup and Track Announcement

The first step in connecting two MoQ entities is to create a QUIC connection. This connection can be made using WebTransport in the case of a web application or just by using QUIC itself. During this process, a bidirectional control channel is opened. This channel allows the two entities to communicate control messages, which are shown in the control message table provided in Table 1 in the Appendix. When a subscriber connects to a MOQT-enabled server, a session setup process occurs. The subscriber and server exchange `CLIENT_SETUP` and `SERVER_SETUP` messages, over the control channel, to negotiate transport options, authentication, and supported features. These messages define the parameters that will govern the session, including QUIC-specific settings and optional security policies.

Once the connection is enabled, the client prompts the publisher for its available tracks using a `SUBSCRIBE_ANNOUNCES` message. This message contains a namespace and lets the publisher know that it has to send a notice of every available track in the requested namespace to the requesting client. This information gets sent over in an `ANNOUNCE` message.

3.1.2 Subscription and Fetching Mechanism

MOQT supports two ways of accessing content:

- **SUBSCRIBE:** The subscriber requests a track to receive continuous updates, making this mode ideal for live streaming. When subscribing, the client may specify certain parameters, such as priority levels or specific subtracks (e.g., only requesting audio instead of full video and audio). Additional flags, such as `subscribe_start` and `subscribe_end`, enable fine-grained control over the start and end group sequence numbers, allowing clients to control the temporal scope of the subscription.
- **FETCH:** The subscriber requests specific past media objects, enabling on-demand playback and rewind functionality. This mode is, for example, helpful in scenarios such as instant replay in sports streaming, where a user can quickly revisit key moments without disrupting the live experience.

When the client receives an `ANNOUNCE` message, it chooses the track it wants to subscribe to and sends a `SUBSCRIBE` or `FETCH` message to the publisher. The publisher then, respectively, follows up with a `SUBSCRIBE_OK` or `FETCH_OK` message, letting the client know their subscription or fetch was successful. The publisher then proceeds to send over the requested data.

3.2 MOQT Principles

Central to MOQT's design are two key principles: prioritisation and the use of relays. These principles work in tandem to ensure that media content is delivered efficiently, even under challenging network conditions, and can scale to meet the demands of a large audience.

The following sections explore these foundational principles in detail, highlighting how MOQT's prioritisation system ensures optimal bandwidth usage and how relays contribute to scalable and efficient media distribution.

3.2.1 Prioritisation System in MOQT

One of the standout features of Media over QUIC Transport (MOQT) is its sophisticated prioritisation system, designed to optimise media delivery under varying network conditions. This system operates on two primary levels: subscriber priority and publisher priority, each playing a crucial role in ensuring efficient and effective media streaming.

Subscriber Priority

Subscriber priority is a mechanism within MoQ that enables clients to assign varying levels of importance to the media tracks they subscribe to. This is achieved through an 8-bit unsigned integer value set in the "subscriber priority" field of the `SUBSCRIBE` message, which reflects the relative priority of the associated stream. This capability becomes particularly valuable in use cases involving multiple concurrent media streams, for instance, when delivering video alongside audio or supplementary metadata. By assigning higher priority values to more critical streams, clients can help ensure that essential content is delivered with minimal disruption, even under constrained network conditions.

Importantly, subscriber priorities are not static. They can be dynamically adjusted in response to real-time network performance or user-specific requirements. This flexibility allows systems to continuously adapt, ensuring that high-priority content, such as audio during a live event, remains uninterrupted. In scenarios of network congestion, subscriber priority serves as a mechanism for intelligent bandwidth allocation. By directing available resources toward high-priority streams, it helps maintain the playback quality of critical content, thereby improving the overall user experience.

Publisher Priority

In contrast to subscriber priority, publisher priority is defined by the media producer and is specified in the `publisher priority` field within the headers of individual media objects in a stream. This mechanism allows content publishers to convey the relative importance of different media elements, ensuring that the most critical components are transmitted first. A common application of this is in video streaming, where keyframes (I-frames) are prioritised over predicted (P-frames) and bidirectional frames (B-frames). By ensuring that keyframes are delivered ahead of dependent frames, the protocol supports smoother playback and faster recovery from interruptions such as packet loss.

Beyond frame-level prioritisation, publisher priority enables object-level control over the transmission sequence. This fine-grained prioritisation is particularly advantageous in adaptive bitrate streaming scenarios, where content quality must be dynamically tuned to match fluctuating bandwidth conditions. By assigning higher priority to essential media objects, publishers can maintain a more consistent user experience, even when network resources are constrained.

Integrated Prioritization

The combination of subscriber and publisher priorities within MoQ establishes a comprehensive framework for adaptive media delivery. By jointly considering both types of priority, MoQ enables more intelligent scheduling decisions that optimise bandwidth utilisation and reduce the likelihood of playback interruptions. This dual-priority model allows the protocol to assess the relative importance of each media object from both the sender's and receiver's perspectives, ensuring that the most valuable content is transmitted first.

Central to this approach is MoQ's scheduling algorithm, which evaluates both subscriber- and publisher-assigned priorities when determining the transmission order of media objects. This ensures that high-priority content is prioritised appropriately, regardless of whether the importance was designated by the client or the publisher. Such a mechanism is especially beneficial in scenarios where network resources are constrained or fluctuate over time.

The resulting prioritisation system is both flexible and efficient, enabling real-time adaptation to varying network conditions. By dynamically adjusting the delivery of media streams based on priority values, MOQT maintains a high quality of experience for end users, even in challenging environments. This integrated approach to prioritisation underscores MOQT's suitability for scalable, latency-sensitive media applications.

By leveraging this dual-level prioritisation system, MOQT provides a powerful tool for managing media delivery, ensuring that critical content is prioritised and that bandwidth is used efficiently. This results in a more reliable and responsive streaming experience for end-users.

3.2.2 Object Hierarchy

To support efficient, scalable, and structured delivery of media, MoQ defines a layered object model. This model consists of tracks, groups, optional subgroups, and objects. Each layer plays a specific role in ordering, dependency management, and stream handling. Figure 3.3 illustrates this hierarchy in the context of a temporal video stream.

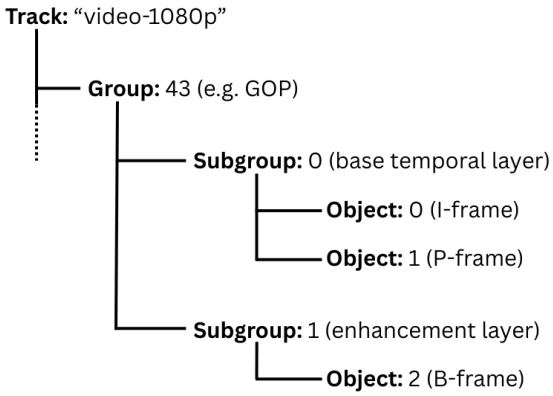


Figure 3.3: Example hierarchy for MoQ objects. At the top level, the track, which in this case is a 1080p video track. This track is divided into groups. The figure shows group 43 as an example, which is a GOP and contains subgroups with objects.

At the highest level, a Track represents a logically continuous stream of media content, such as an audio or video feed. Tracks are uniquely identified by a combination of a track namespace and a track name. Within the MoQ protocol, tracks are the primary unit of subscription and data delivery: subscribers explicitly request access to specific tracks using `SUBSCRIBE` or `FETCH` messages. A track may contain multiple Groups, each encapsulating a coherent segment of content. For example, a group might correspond to a single Group of Pictures (GOP) in a video stream.

Within each group, media content is further subdivided into Subgroups. A subgroup is a sequence of Objects that must be delivered together due to interdependencies, such as temporal or spatial enhancement layers in scalable video coding. Subgroups enable granular stream-level prioritisation and cancellation, since each is typically mapped to an individual QUIC stream. Importantly, subgroups are optional in MoQ. They are only necessary when object interdependencies or delivery semantics require them. When objects are independent and do not benefit from separate prioritisation or stream mapping, they can be sent directly within a group without subgrouping. This flexibility allows senders to reduce complexity and overhead in simple scenarios, while still supporting advanced use cases when needed.

At the lowest level is an object. This is the fundamental unit of media data in MoQ. It consists of metadata and an immutable byte payload, which may represent a frame, audio sample, or any other media fragment. Objects within a subgroup are ordered by Object ID, and their delivery order is significant. Furthermore, objects within the same subgroup may share decoding dependencies, whereas objects from different subgroups are considered independent and decodable in isolation.

3.2.3 Relays and Scalability

In the architecture of MOQT, relay nodes play a pivotal role in enhancing scalability and ensuring efficient media distribution. These relays serve as intermediaries between publishers and subscribers, performing several key functions that collectively contribute to the system's performance and reliability.

Relays are responsible for caching and distributing media objects, which helps reduce the load on origin servers. When subscribers request media content using the `FETCH` mechanism, relays can cache these objects. This caching strategy allows relays to serve subsequent requests for the same content directly rather than having each subscriber retrieve the content from the origin server. By doing so, relays significantly reduce redundant traffic and alleviate the burden on the publisher's infrastructure, leading to more efficient use of network resources and improved content delivery speeds.

Moreover, relays facilitate efficient subscription aggregation. When multiple subscribers request the same media track, instead of each client establishing a direct connection to the publisher, the relay maintains a single subscription for that track. The relay then forwards the media stream to all interested subscribers. This aggregation minimises redundant traffic back to the original publisher, optimising bandwidth usage and reducing the overall network load. By managing subscriptions in this way, relays ensure that the publisher's resources are utilised efficiently, even under high demand.

The strategic deployment of relays in various geographical locations further enhances MOQT's scalability and performance. By positioning them closer to end-users, MOQT can deliver content with reduced latency as the data travels shorter distances. This geographical load balancing ensures that the system can handle large-scale distributions effectively, providing a consistent and high-quality streaming experience to a global audience. Relays distribute the load across different regions, preventing any single server or network segment from becoming a bottleneck, which is particularly important during peak usage times.

In summary, the integration of relay nodes in MOQT enables a highly scalable and efficient media delivery system. By caching content, aggregating subscriptions, and balancing loads geographically, relays ensure that media streams are delivered with minimal latency and maximum resource utilisation, ultimately enhancing the user experience.

Relays play a crucial role in making MOQT efficient at scale, functioning similarly to CDNs in HAS-based streaming but with the added benefit of QUIC's transport optimisations. Unlike WebRTC, which relies on peer-to-peer transmission, MOQT leverages structured routing and caching, achieving both low latency and large-scale distribution.

By combining these key components, MOQT offers a highly flexible, low-latency, and scalable approach to media streaming, significantly improving on both HAS-based and WebRTC-based delivery models.

3.3 WARP: A Streaming Format for Media over QUIC

As Media over QUIC defines the transport layer for real-time and scalable media delivery, it is designed to be agnostic to how media is packaged and formatted. To function effectively, MoQ requires well-defined streaming formats that dictate how media is encoded into objects and groups, the atomic units in the MoQ data model. One such format is WARP [LCV+24], which is introduced and explained in this section.

WARP is not a standalone protocol but a media streaming format that operates on top of MOQT. It defines how media data, such as video, audio, or metadata, is organised, encoded, and mapped to MoQ's object-based delivery system. In essence, WARP complements the MoQ protocol by fulfilling the role that MPEG-TS or fMP4 would play in traditional streaming pipelines, but is designed specifically for the object-centric, publish/subscribe model of

MoQ. By standardising the packaging format, WARP enables interoperability between media producers and consumers over MoQ, while also facilitating relay caching and content-aware processing.

3.3.1 Design Principles and Structure

WARP defines several key components that work together to enable low-latency, adaptive streaming:

- **LOC Packaging (Low-overhead Content)**: Media is segmented into small objects suitable for transport as MoQ Objects. Each object may contain a frame or fragment, with minimal overhead to maximise delivery efficiency.

- **Time-Alignment**: WARP supports alignment of multiple media tracks (e.g., audio and video) to facilitate synchronised playback at the receiver. Each track is time-stamped in a shared timeline.

- **Catalogue Metadata**: Central to WARP is a "Catalogue" structure, which describes all available tracks and their associated metadata (e.g., resolution, codec, framerate). This catalogue enables clients to discover and subscribe to tracks dynamically.

- **Patch Updates**: The catalogue is designed to be updated incrementally using "patches". This allows broadcasters to add, modify, or remove tracks mid-session without resending the entire catalogue, supporting use cases such as dynamic bitrate switching or track reconfiguration.

3.3.2 Track and Object Mapping

In WARP, each media track is identified by a unique name and namespace, which are used to map the track to the underlying MOQT. The tracks are described in the catalogue using a structured metadata schema that encompasses both transport-level and media-level information.

Each track entry specifies the media codec (such as avc1.64001f for H.264/AVC), the expected resolution (e.g., 1280x720), and the target bitrate. The schema also allows for the inclusion of attributes such as temporal and spatial IDs (used in scalable video coding), framerate, mimetype, and initialisation data. These attributes help the subscriber understand the capabilities and configuration of each track and allow relays to process or prioritise tracks appropriately.

Additionally, WARP supports logical groupings such as alternate groups (for multiple quality levels of the same content) or render groups (to coordinate synchronised rendering of audio and video). Dependencies between tracks can be expressed to enable layered decoding or adaptive switching strategies.

To illustrate, the following example shows a simplified entry for a video track in a WARP catalogue:

```
{
        "video": [{
                // Transport information
                "track": {
                        "name": "720p",
                        "priority": 2
                },
                // The codec in use
                "codec": "avc1.64001f",
                //  The resolution of the video
                "resolution": {
                        "width": 1280,
                        "height": 720
                },
                // The maximum bitrate (3Mb/s)
                "bitrate": 3000000
                // etc.
        }],
}
```

Listing 3.1: Example of a 720p video stream entry in a WARP catalogue [Cur25].

This entry defines a video track labelled "720p" with medium priority of 2, encoded using H.264 at 1280x720 resolution and a target bitrate of 3 Mbps. It demonstrates how the catalogue provides enough detail for clients to evaluate the suitability of a track for a given playback context or network condition.

Chapter 4

Using qlog for logging

Structured logging is essential in network protocols to enable effective debugging, performance analysis, and troubleshooting. As modern protocols like QUIC and HTTP/3 grow increasingly complex, a well-defined and consistent logging format becomes indispensable for understanding system behaviour, identifying bottlenecks, and improving reliability. The qlog format [MNSP25] emerges as a powerful solution tailored to meet these demands. Designed specifically for network protocols, it provides a standardised, extensible, and hierarchical format that captures critical events and metrics in a machine-readable and human-friendly manner. This chapter explores qlog's features, benefits, and its application in enhancing the observability and analysis of the Media over QUIC protocol.

4.1 Overview of qlog

Before delving into the challenges of network protocol observability and qlog's specific capabilities, it is important to first understand how qlog is conceptually structured. qlog is a standardised logging framework designed to capture structured events within network protocols such as QUIC and HTTP/3. It defines a flexible and extensible model for representing logs as sequences of discrete protocol events.

A simple example is shown in Listing 4.1. This example shows the hierarchical nature of the qlog file, where the top level is the file itself. This is divided into different traces, which all contain data from a single vantage point, such as a client, server or intermediary. This data is in the form of events, which is a structured object with at least three main fields:

- **time:** a numeric timestamp relative to the start of the trace
- **name:** a string identifier, typically following the format `namespace:event_type`
- **data:** an object containing event-specific fields

```
{
  "qlog_version": "0.4",
  "title": "Sample connection",
  "traces": [
    {
      "vantage_point": { "type": "client" },
      "reference_time": "2025-05-20T14:00:00.000Z",
      "events": [
        {
          "time": 12,
          "name": "transport:packet_received",
          "data": {
            "packet_type": "initial",
            "packet_number": 2
          }
        },
        {
          "time": 34,
          "name": "recovery:congestion_state_updated",
          "data": {
            "old": "slow_start",
            "new": "congestion_avoidance"
          }
        }
      ]
    }
  ]
}
```

Listing 4.1: Basic example of a qlog file. This logging format is divided into three distinct hierarchical parts: the file itself, the traces and the events.

4.1.1 Logging Challenges and qlog's Solutions

Network protocol logging presents several challenges, particularly when dealing with encrypted protocols like QUIC and HTTP/3. The lack of visibility into packet-level behaviour and the variety of custom logging formats used across implementations make it difficult to develop standardised tools for debugging, analysis, and interoperability testing. Additionally, non-standardised formats hinder data sharing and collaboration between teams or organisations.

The qlog format addresses these challenges by providing a structured, extensible logging format for network protocols. Its design facilitates easy sharing, uniform analysis methods, and the creation of reusable tooling. By introducing schema-based logging that supports detailed event descriptions, qlog enables consistent capture of protocol behaviours while allowing for customisation specific to individual use cases.

4.1.2 Key Features of qlog

One of qlog's defining characteristics is its hierarchical structure, which organises data into three levels: files, traces, and events. A log file contains one or more traces, each representing a sequence of events recorded at a particular vantage point, such as a client or server. Events, the smallest units in the hierarchy, capture specific protocol activities, such as a packet being sent or received. The events are described by a type and are combined with a timestamp. This structured approach improves traceability and makes analysis more systematic.

The qlog format is also highly extensible. It supports protocol-specific schemas and allows users to define custom fields for specialised requirements. Using CDDL (Concise Data Definition Language) [BVB19], developers can expand existing event schemas or create entirely new event types while ensuring compatibility with the broader qlog ecosystem.

Another key feature is qlog's compatibility with multiple serialisation formats, including JSON and JSON Text Sequences. This flexibility allows logs to be used in both batch-processing scenarios and streaming contexts, addressing diverse use cases while maintaining interoperability between tools and systems.

Lastly, qlog makes use of namespaces. They play a crucial role in preventing naming conflicts by providing a structured and unique context for defining event types and their associated data. Each namespace represents a distinct domain, such as a specific protocol (e.g., QUIC, HTTP/3), and serves as a prefix for all event types within it. For example, events in the QUIC namespace might include *quic:packet_sent* and *quic:connection_closed*. By prefixing each event type with its namespace (*quic:*), qlog ensures that even if another protocol defines similar event types (e.g., *http:packet_sent*), there is no ambiguity or collision between the names.

4.2 Benefits of qlog

The adoption of qlog as a standardised logging format introduces several advantages that address longstanding challenges in network protocol observability. By combining structure, extensibility, and performance-aware design, qlog enables detailed, interoperable logging without imposing excessive overhead. This section outlines the core benefits of qlog, focusing on its role in promoting standardisation, enhancing debugging workflows, and maintaining logging efficiency in high-throughput environments.

4.2.1 Standardisation

The qlog format establishes standardisation in network protocol logging by introducing a schema-based, structured format applicable across diverse use cases. Standardisation eliminates the fragmentation caused by proprietary or ad-hoc logging systems, allowing developers, researchers, and engineers to work with a consistent format regardless of the protocol or application. This consistency makes it easier to develop universal tools for debugging, visualisation, and analysis. For example, whether logging QUIC, HTTP/3, or custom protocols, qlog's schemas ensure that events are captured, stored, and interpreted in a predictable and uniform manner. This unified approach fosters collaboration, simplifies interoperability testing, and reduces the complexity of integrating logs from multiple sources.

4.2.2 Debugging

For debugging, qlog delivers powerful tools to understand and analyse protocol behaviours. It captures detailed events like packet transmissions, retransmissions, state transitions, and connection-level metadata, providing developers with deep insights into system performance and issues. By organising these events into a hierarchical structure of files, traces, and events, qlog makes it straightforward to isolate problems, such as misconfigured network parameters or implementation errors in a specific protocol. The use of vantage points allows for correlating logs from different locations (e.g., client, server, and network middleboxes), offering a comprehensive view of interactions across the communication stack. Such clarity is invaluable in identifying the root causes of interoperability failures, performance bottlenecks, or unexpected behaviours in encrypted protocols like QUIC.

4.2.3 Efficiency

The qlog format's focus on efficiency ensures that logging does not impose significant overhead while retaining its flexibility and detail. The use of *common_fields* is a key optimisation. These fields allow attributes shared across events, such as protocol type, group IDs, or time format, to be stored once at the trace level rather than repeated for each event. This significantly reduces redundancy, particularly in scenarios where logs capture large volumes of events over time. Additionally, qlog employs delta encoding for timestamps, where each event's time is logged relative to the previous event rather than as an absolute value. This reduces the size of timestamp data while preserving the sequence and timing information necessary for analysis. For high-throughput logging scenarios, such as large-scale QUIC connections, these optimisations ensure that logs remain compact and manageable without sacrificing the granularity needed for debugging and analysis.

4.3 Challenges and Limitations

While qlog offers numerous advantages for logging and analysing network protocols, it also introduces certain challenges and limitations that need to be considered for its effective use in real-world scenarios.

One significant concern is the potential overhead introduced by qlog compared to not logging at all. Although qlog includes several optimisations, such as *common_fields* and delta-encoded timestamps, the act of capturing, serialising, and storing log data inherently consumes resources. This can lead to performance issues, particularly in real-time or high-throughput environments, such as servers handling thousands of simultaneous QUIC connections. The overhead includes increased CPU utilisation for logging operations, memory usage to buffer log data, and I/O strain from writing logs to disk or streaming them to external systems. For time-sensitive applications, the delay introduced by logging may impact user experience or system performance, especially if the implementation is not optimised or logging verbosity is set too high. Balancing the need for detailed logging with system performance remains a critical challenge, particularly for deployments in resource-constrained environments.

Another limitation lies in the area of privacy. qlog's structured format is designed to capture detailed protocol events, which may include sensitive information such as IP addresses, port numbers, connection identifiers, and session data. While this level of detail is invaluable for debugging and analysis, it raises concerns about exposing user data or violating privacy regulations such as GDPR. For instance, logs could inadvertently reveal user-specific patterns, geographic information, or session activities if shared or improperly secured. qlog provides mechanisms like metadata anonymisation and data minimisation to address these concerns, but implementing such practices requires careful planning. Organisations must establish policies for anonymising sensitive fields, controlling access to logs, and ensuring secure storage. However, these measures can add complexity to logging workflows and may reduce the utility of the logs for some analytical purposes.

A third challenge is the relative scarcity of tooling for qlog compared to more established logging systems. While the qlog format is designed to be easily parsed and used, the ecosystem of tools for visualisation, analysis, and debugging is still developing. Tools such as qvis (for visualising QUIC and HTTP/3 logs) exist but are specialised and limited in their capabilities compared to more mature logging ecosystems like ELK (Elasticsearch, Logstash, Kibana) or Splunk. Additionally, integrating qlog with existing logging pipelines may require significant customisation, especially for environments where qlog must coexist with other formats or logging systems. The limited availability of tooling also places a more significant burden on developers to create custom scripts or tools to process and analyse qlog files, which can be a barrier to adoption. Expanding the availability and capabilities of qlog-compatible tools will be essential for broader uptake in production systems.

4.4 Applying qlog to Media over QUIC

The qlog format's structured logging format aligns closely with the debugging and analysis needs of Media Over QUIC. Its ability to capture and organise detailed event data provides a comprehensive view of both transport and application-layer behaviours, enabling developers to diagnose and optimise protocol performance effectively.

One of the key challenges in MoQ is diagnosing issues in QUIC's encrypted transport layer. Traditional packet capture tools struggle to provide insights into encrypted streams. qlog addresses this by logging events directly from the endpoints, offering visibility into internal protocol operations without requiring access to decrypted packet data. For example, qlog can capture events such as packet retransmissions or stream resets, helping developers pinpoint the root causes of delivery failures or performance issues.

Moq's design requires close interaction between the transport and media layers. qlog enables this by supporting multiple namespaces within a single trace, allowing developers to log and correlate media-specific events (e.g., *moq:frame_sent*) with QUIC transport events (e.g., *quic:packet_lost*). This cross-layer insight simplifies debugging by providing a unified view of interactions, making it easier to diagnose issues like synchronisation errors or suboptimal retransmission strategies.

The qlog format's structured logging allows MoQ developers to capture metrics essential for real-time media delivery, such as packet delivery times, round-trip times, and retransmission rates. Analysing these logs can reveal inefficiencies, such as congestion control misconfigurations or high latency during media playback. These insights can guide optimisations to improve the user experience, such as adjusting retransmission timers or fine-tuning adaptive bitrate algorithms.

The qlog's format extensible framework is especially valuable for MoQ, which may require custom event types to log application-specific details. Developers can define custom schemas to capture media-related metrics, such as frame encoding parameters, synchronisation between audio and video streams, or adaptive bitrate adjustments. This customisation ensures that logs provide actionable insights tailored to the unique needs of media delivery protocols.

Moq's real-time nature benefits from qlog's support for streamed logging formats, such as JSON Text Sequences. This capability enables developers to monitor logs as they are generated, facilitating rapid feedback during debugging sessions or live performance monitoring. Streamed logs integrate seamlessly with real-time visualisation tools, helping developers identify and address issues without waiting for complete log files.

Chapter 5

Scaling a Web-based MoQ Logging System

As Media over QUIC matures into a protocol designed for large-scale, real-time media delivery, ensuring that its logging infrastructure can scale accordingly becomes crucial. Observability systems must handle high volumes of protocol events, deliver actionable insights under variable traffic loads, and remain responsive during peak conditions. This chapter explores the challenges of designing a scalable logging infrastructure for Media over QUIC, with a focus on structured logging using qlog. These challenges arise from the nature of distributed systems, the constraints of real-time data handling, and the need to store and manage large volumes of protocol-specific logging data. Ensuring the system's performance and reliability under varying load conditions is critical, especially for live and high-throughput media scenarios.

To address this, the chapter is structured around key points in the logging pipeline: ingestion, storage, and processing, where specific scalability issues are likely to emerge. Each section presents the challenges inherent to that stage and proposes targeted solutions to mitigate them. In the final section, we discuss broader architectural strategies that apply across the entire system, offering cross-cutting approaches to enhance scalability, flexibility, and resilience.

5.1 Handling High Throughput of Logs

A scalable logging system must be capable of handling a high volume of log messages without compromising performance. As the number of clients increases, so does the rate of log generation, leading to potential bottlenecks in data transmission, storage, and retrieval. This section examines the challenges associated with high log throughput and the strategies employed to ensure efficient processing, even under peak loads.

5.1.1 Event Categories and Their Logging Impact

Each client in the system generates logs at varying rates depending on factors such as video streaming activity, logging verbosity, and network conditions. The types of logs generated are crucial for understanding client behaviour, diagnosing issues, and optimising the streaming experience. For instance, a client might produce logs for the following categories:

- **Connection and Transport Events:** These logs provide insights into the underlying network conditions and the client's interaction with it. Examples include:
  - *Congestion Control Metrics:* Information about congestion window size, packet loss rates, and round-trip times helps understand network congestion and its impact on

streaming quality. These logs are essential for diagnosing network-related playback issues, such as buffering or dropped frames.

- *Jitter Measurements:* Variations in packet arrival times (jitter) can lead to audio and video glitches. Logging jitter values allows for identifying network instability and its effects on the user experience.
- *Flow Control Metrics:* Data related to flow control mechanisms, such as backpressure signals, helps analyse how the client adapts to varying network bandwidth availability. This is crucial for smooth bitrate adaptation.

The volume of connection and transport-related events, such as connection setup, path changes, or packet-level feedback, is largely influenced by network conditions. In stable environments, these events remain infrequent and predictable. However, under unstable or lossy network conditions (e.g., mobile handovers, congestion episodes), these events can spike due to path validation attempts, retries, or transport-level adjustments. This makes their frequency highly dynamic and potentially bursty during adverse conditions.

- **Media-Related Events:** These logs capture information about the actual streaming process and the client's interaction with the media stream. Examples include:
  - *Bitrate Adaptation:* Logs related to bitrate switching decisions (e.g., when the client switches from 720p to 480p) are essential for understanding how the client adapts to changing network conditions. These logs can be used to optimise adaptive bitrate (ABR) algorithms.
  - *Stream Subscribing/Unsubscribing:* Logs indicating when a client joins or leaves a stream are important for tracking viewership and resource utilisation on the server.
  - *Buffer Underruns:* These logs indicate instances where the client's buffer runs out of data, resulting in playback interruptions. They are critical for identifying network or server-side issues.

  Media-related events are typically the most frequent in MoQ-based systems, as they track the actual flow of audio/video content. The logging volume here is directly tied to the configured verbosity level. In low-verbosity modes, only key transitions (e.g., bitrate shifts, keyframe boundaries) may be logged. In contrast, high-verbosity modes may emit logs for every datagram or object parsed, which can result in tens of thousands of events per second, particularly for high-bitrate streams or fine-grained object structures. As such, careful calibration is needed to balance observability with system overhead.
- **System-Level Diagnostics:** These logs capture information about the client's resource usage and overall health. Examples include:
  - *CPU Usage:* Logging CPU utilisation allows for the identification of performance bottlenecks on the client side.
  - *Memory Consumption:* Tracking memory usage helps detect memory leaks or other memory-related issues that could affect streaming performance.
  - *Player State (Playing, Paused, Buffering):* Logging the player's state allows for the analysis of user interaction and the correlation of playback issues with specific player states.
  - *Timestamp of key events like seeking:* Logging the timestamp of key events, such as seeking, is crucial for analysing user behaviour and correlating it with potential playback issues.

  System-level diagnostic events, such as manual instrumentation markers, logging of system state snapshots, or user-triggered actions (e.g., toggling logging views or issuing manual queries), tend to be relatively infrequent. Their frequency is typically dictated by developer or operator intent rather than protocol dynamics. While they provide valuable

contextual information (especially during debugging sessions), they contribute minimally to log volume compared to transport or media-layer events. Nevertheless, their timing and correlation with other high-frequency logs can be crucial for root cause analysis.

If logging is performed at a fine-grained level, each client could generate hundreds of log entries per second. For example, a client switching bitrates multiple times during a session, experiencing occasional network jitter, and logging various player states could quickly produce this volume of data. With thousands of clients operating simultaneously, the system must efficiently ingest and process hundreds of thousands of log messages per second. Assuming, for example, 5,000 concurrent clients, each generating 200 logs per second, the system needs to handle 1 million log messages per second. Without optimisation, this influx can overwhelm the message broker, backend, storage layers, and visualisations. Therefore, careful consideration must be given to the design and implementation of each component of the logging pipeline to ensure it can handle this scale of data.

5.1.2 Burst Traffic Scenarios

One of the most significant challenges in designing a scalable logging system, especially for real-time applications like video streaming, is accommodating burst traffic patterns. Unlike a steady stream of logs, real-world scenarios often involve unpredictable spikes in logging activity. These bursts can overwhelm the system if it's not designed to handle them effectively, leading to excessive latency, dropped log messages, or even complete system failure. In the context of video streaming, several factors can trigger such bursts:

- **Network Degradation:** A sudden network degradation, such as a temporary loss of connectivity to a Content Delivery Network (CDN) server or a widespread network outage, can cause a cascade of log messages from affected clients. For example, if a CDN server becomes unreachable, all clients relying on that server will experience packet loss. This will trigger retransmission requests, increased latency, and potential connection failures, leading to a surge in logs related to these events. Furthermore, clients might attempt to switch to lower bitrates or pause playback, generating additional log entries. These correlated log messages from numerous clients can create a significant spike in log volume.
- **Client-Side Issues:** Issues on the client side, such as application crashes, unexpected behaviour, or resource exhaustion (e.g., CPU overload), can also trigger bursts of log data. For instance, a client application crash might result in a "log dump" containing the application's state and error messages at the time of the crash. If multiple clients experience similar issues simultaneously (e.g., due to a software bug), the resulting log dumps can create a substantial burden on the logging system. Similarly, resource exhaustion on the client side can cause a sudden increase in logging activity as the client struggles to maintain performance.
- **Large-Scale Events:** Large-scale video streaming events, such as live broadcasts of popular events, can lead to a massive influx of log messages. During these events, many clients connect to the streaming service simultaneously, generating logs related to connection establishment, stream subscription, playback, and other activities. Even if the average log rate per client is relatively low, the sheer number of concurrent clients can result in a very high aggregate log volume, especially during peak viewing times. Furthermore, these events often have predictable peak times, allowing for some preemptive scaling, but unexpected surges in viewership can still strain the system.
- **Server-Side Issues:** Problems on the server side, such as database failures, message broker outages, or backend service disruptions, can also lead to bursts of log data. For example, if the database used to store logs becomes unavailable, the backend might queue log messages in memory. Once the database is restored, the backend will flush the accumulated log messages to the database, resulting in a sudden spike in write activity. Similarly, a failing message broker might cause clients to retry sending messages, leading

to a surge in traffic when the broker recovers.

If the system is not designed to handle these bursts effectively, it may experience excessive latency, dropped log messages, or even complete failure due to overwhelmed resources. This can severely impact the ability to monitor the health and performance of the video streaming service, making it difficult to diagnose and resolve issues. Therefore, robust mechanisms for buffering, queuing, and processing log messages are essential to ensure the logging system remains resilient during periods of high load.

5.1.3 Message Brokering for Scalable Log Ingestion

In high-throughput environments, transmitting log data directly from instrumentation interfaces to backend systems can lead to performance degradation and reliability issues. To mitigate this, many scalable logging architectures incorporate a message broker [Ibm25], a decoupled intermediary that facilitates the ingestion, buffering, and routing of log messages between producers and consumers.

A message broker supports communication patterns such as publish-subscribe or message queuing, allowing log producers to emit events without needing to be aware of downstream processing components. This decoupling enables more resilient and modular systems, as it reduces tight coupling and improves fault tolerance.

Well-known message broker systems include Apache Kafka [1], RabbitMQ [2], NATS[3], and Eclipse Mosquitto [4]. Each is suited to different use cases depending on the performance requirements and protocol constraints. For example, Mosquitto is a lightweight MQTT broker that is particularly well-suited for scenarios where low overhead, small footprint, and push-based delivery are advantageous, common in embedded or edge-based logging scenarios. Apache Kafka, on the other hand, is designed for high-throughput, distributed messaging and persistent log storage. Its strong durability guarantees, horizontal scalability, and efficient batch processing, make it a natural fit for backend systems that need to ingest and process large volumes of structured logging data, such as those generated in MoQ observability pipelines.

By introducing a broker, the system can scale horizontally through topic partitioning, consumer replication, and load balancing. Additionally, brokers often provide features such as delivery guarantees, message persistence, and temporal buffering, all of which are critical in environments with intermittent connectivity or fluctuating traffic.

In summary, the use of a broker-based architecture offers a scalable, flexible approach to managing high-volume log flows while maintaining reliability and extensibility between the instrumentation interface and the backend infrastructure.

5.1.4 Transport Protocol Considerations for Scalable Delivery

The selection of a transport protocol to relay log messages from the instrumentation interface to the backend infrastructure plays a pivotal role in the design of a scalable logging system. This choice directly impacts performance characteristics such as latency, throughput, delivery guarantees, and fault tolerance, and can constrain or guide the choice of message broker technology.

Several protocol options are commonly used in logging systems, each with trade-offs:

- **HTTP/HTTPS**: Ubiquitous and firewall-friendly, but incurs higher overhead due to connection setup and statelessness. Suitable for batch uploads or push-to-ingest gateways.

[1] `https://kafka.apache.org/`
[2] `https://www.rabbitmq.com/`
[3] `https://nats.io/`
[4] `https://mosquitto.org/`

- **WebSockets**: Enables low-latency bidirectional communication. Useful for browser-based instrumentation or scenarios with continuous log streaming.
- **gRPC**: Offers performance efficiency with support for streaming and structured schemas (via Protocol Buffers). Often used in tightly coupled microservices environments.
- **MQTT**: A lightweight publish-subscribe protocol optimised for bandwidth-constrained and high-latency networks. MQTT is ideal for mobile, edge, or embedded devices and is fully supported by brokers like Eclipse Mosquitto.
- **QUIC-based transport**: Emerging as a viable candidate for telemetry and logging due to its low-latency and multiplexing capabilities, though adoption in logging scenarios is still nascent.

Choosing a protocol involves evaluating the operational environment (e.g., bandwidth constraints, device capabilities, NAT traversal), delivery requirements (e.g., at-most-once, at-least-once), and integration with existing broker ecosystems. For example, selecting MQTT as the transport protocol naturally leads to adopting Mosquitto or a similar lightweight broker to manage the message flow.

In essence, the transport protocol serves as the backbone of log communication and should be selected with an awareness of how it aligns with system constraints and the overall logging architecture.

5.2 Data Storage and Management

Given the high-throughput nature of MoQ media sessions and the structured verbosity of qlog-based instrumentation, the logging system must handle both real-time ingestion and efficient, cost-aware long-term storage. This section examines storage challenges and presents design strategies tailored to the needs of a MoQ-aware logging architecture.

5.2.1 Storage Capacity and Cost

Video streaming logs can consume significant storage space. Factors like the number of clients, logging verbosity, and the duration of log retention all contribute to the overall storage requirements. Estimating the required storage capacity is crucial for planning infrastructure and budgeting. Therefore, we introduce a theoretical function to get a broad calculation of how much data needs to be stored. We base this on one of the control message types which results in the most amount of messages (Bitrate Adaptation), as this will give us the most accurate estimation.

$$\boxed{L = W \times 60 \times R \times M \times B}$$

L = amount of log storage
W = watch-time in hours
R = number of ABR switches per minute
M = number of control messages sent per switch
B = average size of qlog message in bytes

To get an estimation of L, we can look at an example such as Twitch.com[5], a renowned live streaming platform and thus a potential future user of MoQ. At the time of writing this, Twitch averages around two million viewers per day [Twi25], resulting in 50 million hours of watchtime. Furthermore, the number of bitrate switches averages one per two minutes [TDM16], every switch results in two messages (`UNSUBSCRIBE` from current bitrate track and `SUBSCRIBE` to new bitrate track) and every log message in qlog format averages 200 bytes. We fill these values into the function:

[5] `twitch.com`

$$W = 5.0 \times 10^7 \text{ h/day}$$
$$R = 0.5 \text{ switches/min}$$
$$M = 2 \text{ msgs/switch}$$
$$B = 200 \text{ bytes/msg}$$

$$L \;=\; 5.0 \times 10^7 \times 60 \times 0.5 \times 2 \times 200 \;\approx\; 6.0 \times 10^{11} \text{ bytes/day} = 600 \text{ GB}$$

This simplified model illustrates the rapid accumulation of log data, even before considering peak loads due to popular streaming days, increased verbosity as a result of more message types, or larger log sizes due to richer event content. Without mitigation, this results in unsustainable storage pressure. A good solution to this challenge of large amounts of log messages that need to be stored can be found in data retention policies, which concern what data should be stored or archived, where that should happen, and for exactly how long. Most logging messages lose a lot of their value when kept for a longer period of time. An option here would be to keep all logs for a short, fixed period (e.g., 7–30 days) in fast-access storage to facilitate debugging and real-time system monitoring. This can then be useful for critical incidents that require quick investigation using recent trace data. After that time period, we could employ a hierarchical storage model (HSM) [She22] that separates logs by access frequency. For instance, "hot" logs can be stored in high-performance, document-based databases like MongoDB[6], which offer flexible schema support and optimised indexing for JSON-style records such as qlog entries. Alternatively, SQL-based databases such as PostgreSQL[7] could be used when stronger consistency or complex relational querying is required. Older or less frequently accessed logs can be compressed and migrated to "cold" storage solutions such as Amazon S3[8] or other archival platforms. This tiered approach balances accessibility, performance, and storage cost effectively.

As logs age and their relevance diminishes, retaining every individual entry becomes both unnecessary and inefficient. To further optimise storage, log aggregation and sampling can be applied as post-processing strategies. Aggregation involves summarising fine-grained events into coarser metrics, for example, compiling bitrate switches into hourly distributions or tracking average object delivery times per session. Sampling, on the other hand, reduces log density by retaining only a subset of events (e.g., every Nth entry), which can significantly cut storage volume while preserving overall behavioural trends. Though these techniques reduce granularity, they are highly effective for long-term retention without compromising the system's ability to analyse historical performance.

Finally, after logs reach the end of their defined lifecycle, they should be automatically deleted. Before deletion, compaction techniques, such as deduplicating repetitive debug entries or pruning verbose internal states, can reduce unnecessary storage use without sacrificing analytical integrity.

5.2.2 Data Retrieval and Querying

Efficient retrieval of log data is essential for debugging, performance analysis, and security investigations in a scalable MoQ logging system. As log volumes increase, the ability to query logs quickly and precisely becomes more challenging. For example, users may need to isolate all error events for a specific session within a given time window or trace the sequence of control messages exchanged during a failed subscription negotiation. To support these use cases, the system must implement mechanisms that allow fast filtering by criteria such as timestamps, client or track IDs, event types, or error codes.

[6] `https://www.mongodb.com/`
[7] `https://www.postgresql.org/`
[8] `https://aws.amazon.com/s3/`

One core challenge is the indexing of large datasets, which is critical for maintaining low-latency query performance at scale. Without carefully designed indexes, search operations over millions of entries become computationally expensive. Time-series databases such as InfluxDB or TimescaleDB offer efficient indexing for time-based queries, making them well-suited for chronological inspection of log flows. For more flexible or full-text querying, Elasticsearch and Solr provide distributed indexing and advanced search capabilities, enabling fast filtering across nested JSON structures common in qlog.

Another key difficulty lies in handling complex queries, where users combine multiple filters and conditions, for example, correlating session metadata with control-plane errors and transport behaviour. Supporting such queries requires both efficient query execution plans and a sufficiently expressive query language. Systems like ClickHouse or MongoDB (with its aggregation pipeline) can be advantageous here, depending on the data model and workload.

Finally, query performance optimisation becomes critical, especially for real-time dashboards and on-demand visualisations. Techniques such as caching frequent queries, pre-aggregating metrics over time intervals, or applying data downsampling can significantly reduce system load while maintaining responsiveness.

While general-purpose document databases like MongoDB are attractive in low-volume environments due to their native JSON support and flexible schemas, features that align well with qlog's structure, they may fall short in high-throughput scenarios. For production systems, database selection should be driven by scalability, indexing granularity, and query performance under load.

5.2.3 Data Security and Privacy

Log data generated by a MoQ observability system may contain sensitive user-related information, including IP addresses, client identifiers, session metadata, and in some cases, content-specific references. If improperly handled, this data poses significant privacy and security risks, especially under regulatory frameworks like the General Data Protection Regulation (GDPR). To address these concerns, several key challenges must be considered and mitigated through technical and procedural safeguards.

One major concern is unauthorised access to log data, which can expose identifiable user information. This can be mitigated by implementing strict access control mechanisms, such as role-based access control (RBAC), ensuring that only authorised personnel or services can view or manipulate log data based on predefined permissions.

A second critical requirement is data protection during storage and transmission. Logs should be encrypted both at rest and in transit using modern cryptographic standards (e.g., TLS for transport, AES-256 for storage). This reduces the risk of data breaches due to interception or unauthorised file access.

A further privacy challenge is the potential identifiability of users within stored logs. To address this, data anonymisation or pseudonymisation techniques should be applied, for example, hashing or tokenising user identifiers, masking IP addresses, or aggregating sensitive fields. These practices help minimise personal data exposure, especially in environments used for analytics, development, or research.

Finally, the system must ensure ongoing compliance with legal and industry standards, such as GDPR, CCPA, or ISO 27001. This includes maintaining clear audit trails of access to log data, supporting user data deletion upon request, and documenting how sensitive fields are handled across the logging pipeline.

By embedding these measures into the design of the logging infrastructure, the system ensures that privacy and security are not afterthoughts but core architectural concerns, essential both for protecting user data and for maintaining trust with stakeholders.

While implementing strong privacy and security measures is essential, it also introduces trade-offs in terms of data completeness and analytical richness. For instance, allowing users to opt out of logging or selectively disable certain types of event capture, in line with privacy regulations or personal preferences, inevitably reduces the overall volume and diversity of data available for system-wide analysis. Similarly, aggressive anonymisation or redaction may obscure patterns or correlations that are otherwise valuable for debugging or performance optimisation. This tension between individual privacy rights and collective insight is particularly relevant in open ecosystems or public research settings, where shared observability data plays a crucial role in protocol evolution and interoperability testing. Designing the system to respect user agency while still capturing sufficient aggregate data for meaningful analysis requires careful calibration of logging defaults, anonymisation strategies, and opt-out mechanisms.

5.3 Log Processing and Aggregation

The sheer volume of log data generated by a large-scale video streaming platform necessitates efficient processing and aggregation mechanisms. This section discusses the challenges and considerations involved in processing and aggregating qlog data from multiple sources.

5.3.1 Log Parsing and Normalisation with qlog

Logs from different sources (clients, servers, network devices) might have varying formats, even within a structured logging framework like qlog if extensions are used. Parsing and normalising these logs into a consistent structure is essential for easier processing, analysis, and aggregation. Fortunately, the use of qlog significantly simplifies this process. Because qlog defines a structured format for logging, it allows for consistent parsing and reduces the complexity of normalisation. Since all logs adhere to the qlog structure, the effort required to write parsers is greatly reduced. This standardisation enables easier integration with various analysis tools and facilitates efficient aggregation of log data from different sources. While qlog itself provides a base structure, protocol-specific or application-specific extensions might still require some level of normalisation to ensure compatibility with analysis tools. However, the core structure provided by qlog greatly minimises the effort required.

5.3.2 Log Aggregation

Aggregating logs from multiple clients and servers is crucial for a holistic view of the system's behaviour. This is particularly important for correlating events across different clients, servers or network devices involved in the same video streaming session. For example, to diagnose a playback issue reported by a user, you might need to aggregate logs from the subscribing client, the relay, and the origin client to identify the root cause. Qlog's structured format makes aggregation significantly easier. The consistent structure and the inclusion of common fields (like timestamps and connection identifiers) facilitate the correlation of events across different log files. This enables the creation of aggregated views of the streaming session, providing insights into the end-to-end performance and identifying potential bottlenecks.

5.3.3 Real-time vs. Batch Processing

In a scalable MoQ observability system, both real-time and batch log processing play crucial but distinct roles, each optimised for different types of insight and operational use cases.

Real-time processing is primarily focused on operational monitoring and reactive debugging. As logs are ingested, they are immediately parsed and analysed to detect anomalies, track protocol-level events, or trigger alerts. This enables engineers and operators to respond rapidly to issues such as unexpected connection drops, elevated error rates, or inconsistent control message flows. Real-time dashboards, for example, allow live tracking of stream health across active sessions, helping identify problems as they unfold and reducing mean time of detection.

Batch processing, in contrast, supports long-term, aggregated analysis of system behaviour. Rather than responding to immediate events, it enables retrospective evaluations, such as identifying usage trends across client populations, analysing patterns in subscription behaviour, or correlating streaming quality with network conditions. These insights inform performance optimisation, capacity planning, and strategic decision-making. Batch pipelines can also incorporate more computationally intensive processing (e.g., clustering, anomaly classification, report generation) that would be too resource-heavy to run live.

Because each mode serves a different analytical purpose, real-time for immediate diagnosis, batch for historical context and systemic understanding, modern observability systems typically adopt a hybrid architecture. Logs are streamed into a real-time processing layer for active observability and simultaneously persisted for batch workflows that run at scheduled intervals or on demand. This combination ensures that both urgent operational visibility and deep analytical insight are supported by the same underlying logging infrastructure.

5.4 Other Strategies for Scalable Web-Based Logging

The preceding section identified the key challenges associated with scaling a logging system for a Media over QUIC deployment. These include managing bursty traffic patterns, ensuring low-latency access to logs, controlling storage growth, and supporting concurrent access by analysis tools and visualisations. Left unaddressed, such issues can lead to degraded performance, reduced observability, and an inability to support large-scale media streaming scenarios effectively.

To address these challenges, this section proposes a set of architectural and operational strategies that align with the specific characteristics of MoQ and its logging requirements. These strategies include modular design principles, efficient storage practices, adaptive deployment methods, and effective visualisation tooling. The aim is to enable the logging system to scale in proportion to traffic without compromising responsiveness, reliability, or analytical precision.

5.4.1 Scalable System Architecture

A foundational principle for enabling scalability in a web-based logging system is the decomposition of the system into modular services. By structuring the logging pipeline into independent components, such as ingestion, buffering, processing, storage, and visualisation, each service can be scaled independently based on its load profile. This modularity also simplifies failure isolation and enhances the maintainability of the system.

To decouple producers from consumers and mitigate the effects of burst traffic, an asynchronous message queue or log buffer can be introduced. In this model, incoming log events are temporarily stored in a resilient queuing system (e.g., Kafka [9], Redis Streams [10]), which serves as an intermediary between high-throughput producers (such as MoQ relays) and downstream consumers. This design smooths out load spikes, avoids data loss during temporary slowdowns in storage backends, and simplifies parallel processing of logs.

5.4.2 Elastic Deployment and Load Adaptation

In environments where traffic is variable or difficult to predict, such as during major live events, elastic deployment is essential. Container orchestration platforms like Kubernetes support horizontal pod autoscaling based on resource metrics such as CPU, memory, or custom event rates. By dynamically adjusting the number of log ingestion or processing instances, the system can accommodate high loads without requiring permanent overprovisioning.

[9] `https://kafka.apache.org/`
[10] `https://redis.io/`

Additionally, isolating high-throughput sources can prevent them from overwhelming shared infrastructure. For instance, logs originating from relays handling a high number of subscribers can be routed to separate ingestion queues or stored in dedicated partitions. This separation preserves fairness and ensures that critical but low-volume logs are not dropped or delayed.

Chapter 6

Observability System Proposal

This chapter introduces an observability system for the MoQ protocol. It proposes a scalable and extensible architecture designed to support both real-time debugging and long-term analysis. While Chapter 5 addresses the broader scalability concerns, this chapter presents a concrete solution to those challenges. Given the evolving nature of MoQ, the system is built with adaptability at its core, ensuring compatibility with future protocol developments and enabling continued relevance as the ecosystem matures.

Beyond raw protocol mechanics, the availability of robust debugging, analysis, and deployment tooling plays a critical role in protocol adoption. Established HAS protocols such as MPEG-DASH and HLS benefit from well-defined media segment formats, standardised manifests, and mature tooling ecosystems for media packaging, manifest generation, and adaptive bitrate logic. MoQ, by contrast, is still in the early stages of ecosystem development. Tooling support remains limited, and integration with browsers or major media servers is currently non-existent. While projects such as FFmpeg do have preliminary support for MoQ-based streaming, this remains the exception rather than the rule. Technical feasibility exists, but widespread support and accessibility are lacking. This lack of tooling hampers visibility and debuggability in MoQ deployments, especially given that QUIC's encrypted transport layer restricts passive network inspection. This challenge reinforces the necessity of purpose-built observability infrastructure such as the one proposed and prototyped in this work.

Currently, for QUIC-related traffic, logging is primarily conducted using qvis[1]. This visualisation tool aids in the interpretation of QUIC-related events by providing a structured and interactive view of qlog data. While it facilitates a deeper understanding of QUIC protocol behaviours, its scope is limited to single-perspective analyses and does not offer a fully holistic overview of multi-endpoint or end-to-end interactions. This is the main reason we propose a new system aimed at filling this gap. We do this by offering a distributed platform, giving the developers free rein over what they want to log regarding MoQ. The aim here is to provide them with standardisation of how the logging happens while also giving enough freedom to allow for new insights and to ensure the platform's longevity.

The proposed logging system consists of a package that can be included in the MoQ project, offering an easy-to-use API for creating event-based logs. This side of the system ensures formatting and is in contact with a backend via a broker. When the events arrive at the backend, they are saved and can be accessed via an interactive tool. Here, the developer or operator can listen to live transfers of streams and fetches (mentioned in Section 3.1.2), save this transfer data or load previously saved data files. We opted for qlog as a data format as this has become the standard for any QUIC-related logging and offers good building blocks as well as the option for expansion.

[1] `https://qvis.quictools.info`

This chapter outlines the proposal for an observability system tailored to MoQ. It begins by defining the system's goals and requirements, followed by a description of its architecture and each of its core components. Subsequently, it presents the underlying data model, explaining how qlog was selected, how it is extended to support MoQ-specific events, and how these logs are structured and processed. The chapter concludes by detailing the data flow from instrumentation to visualisation.

6.1 System Goals and Requirements

As mentioned before, the system aims to provide developers with a platform to log and visualise a streaming setup during their testing and development process, as well as a way of gathering information on end-users utilising a streaming service. By offering a distributed way of logging, a developer can obtain a unified visualisation of the inner workings and problems of their entire setup, eliminating the need to manually combine different logging implementations and visualisations.

In creating such a system, some functional requirements are very important. First of all, logging should be possible from multiple locations, whether distributed geographically or consolidated on a single machine. This approach supports both individual developers who run local setups and work independently to improve or contribute to the protocol, and larger organisations that have access to infrastructure such as content distribution networks. It allows each to implement and evaluate parts of a MoQ deployment in a manner suited to their available resources.

Another key requirement is persistent log storage, which allows users to compare historical traces and identify long-term trends. This is crucial for recognising improvements or diagnosing recurring issues. Furthermore, persisted logs create the possibility of exporting data, allowing users to leverage external visualisation or analysis tools, thus enhancing the system's analytical capabilities and flexibility. Following this, we also state that developers should be able to upload their own formatted qlog documents to the tool, allowing them to visualise externally logged data.

Equally important is the visualisation of logged data, which transforms raw events into actionable insights. For a system such as this to deliver real value and achieve widespread adoption, it must provide users with an effective means of interpreting the information it collects. Simply generating data is insufficient; users need insight. By integrating a visualisation tool that presents the logged events in a clear and structured manner, the system empowers users to interpret complex protocol behaviour more intuitively.

Beyond the core functional requirements, the system must embody several critical non-functional attributes to ensure its operational quality, reliability, and long-term value, particularly given the dynamic nature of MoQ development and the potential scale of deployment. Scalability stands out as a primary concern; the system must be architected to gracefully handle significant and potentially fluctuating loads. This involves efficiently managing high throughput rates of log messages originating from numerous concurrent clients and absorbing unpredictable traffic bursts, which are common in streaming environments. Consequently, the entire pipeline, encompassing data ingestion, processing, storage, and querying, must be designed for horizontal scaling to maintain performance as demand grows, likely employing strategies like message brokers and distributed processing, as explored in Chapter 5.

Closely related is the need for fault tolerance. The logging infrastructure should demonstrate resilience, ensuring that failures within individual components, such as a specific logger instance, the message broker, or backend services, do not cascade into complete system failure or lead to substantial data loss. Implementing redundancy, robust queuing mechanisms and strategies for graceful degradation will be vital in maintaining system availability and the integrity of the collected log data.

Performance is another key consideration. The logging activities must impose minimal overhead on the MoQ applications under observation to avoid impacting their primary function. Furthermore, the system should provide timely feedback, especially for real-time monitoring scenarios, necessitating low-latency log ingestion and processing. Efficient data retrieval and querying are also essential for effective debugging and analysis. This requires careful optimisation throughout the system, including efficient data serialisation like qlog, effective database indexing, and streamlined data pipelines.

From the perspective of MoQ developers, usability of the logging library is critical. Instrumenting a protocol implementation should be as frictionless as possible, relying on a minimal and intuitive API. Developers require clarity in how to initialise and use the logging interface, as well as flexibility to log protocol-specific events without significant overhead or disruption to existing code structures.

For users interacting with log data, such as protocol researchers, system operators, or QA engineers, the visualisation interface must prioritise accessibility and clarity. An effective tool should support uploading and downloading logs with ease, while providing interactive visualisations that highlight temporal relationships, session dynamics, and protocol behaviours.

Finally, acknowledging the evolving nature of the MoQ protocol, maintainability and extensibility are crucial for the system's longevity. The architecture and codebase should be well-structured and readily adaptable to future modifications. This includes accommodating changes to the MoQ protocol itself, integrating logging for new features, and potentially adding new visualisation capabilities over time. Designing for modularity and leveraging standardised formats like qlog will significantly aid in achieving this necessary flexibility.

6.2 System Architecture

Having established the system's core requirements, we now explore its architectural design, detailing how each component contributes to these goals. Figure 6.1 illustrates the proposed architecture, which is organised into four discrete layers: the instrumentation interface, the logger broker, the log serialiser and store, and, finally, the visualisation front-end. Each layer satisfies a distinct subset of the stated requirements, and together they constitute a coherent end-to-end solution. In the sections that follow, we discuss the function and interconnection of each layer, while deliberately avoiding prescriptive implementation details; this intentional latitude enables adopters to choose technologies that best suit their context rather than conforming to arbitrary constraints.

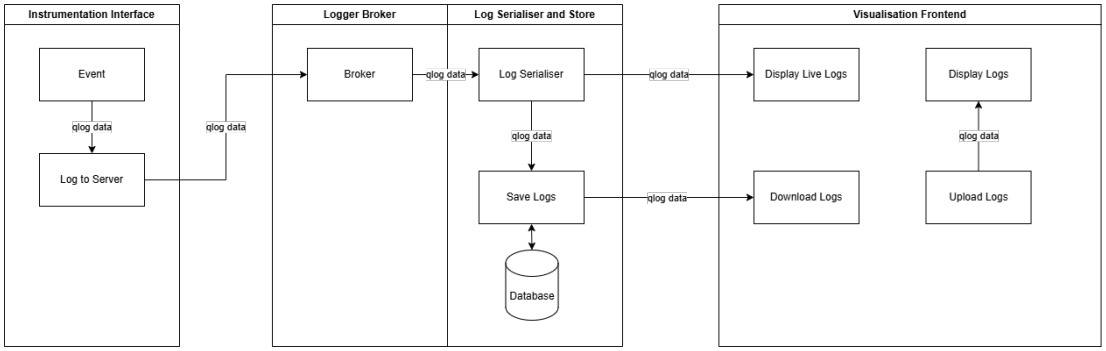


Figure 6.1: This figure shows the proposed system layout, which comprises four parts in combination with the already existing MoQ implementation. These parts are, in order from where the logging starts, the instrumentation interface, the logger broker, the log serialiser and store and the visualisation frontend.

6.2.1 Event Capture and Instrumentation Interface

This component is a modular and lightweight logging interface designed specifically for MoQ. It provides a dedicated API for capturing MoQ-specific runtime events in a structured, qlog-compliant format. Developers can use a single function call to log essential metadata, including the event type, vantage point (such as client or relay), associated message payloads, and a stream identifier. The stream identifier facilitates log grouping across distributed systems, enabling comprehensive analysis from multiple vantage points. The interface integrates seamlessly with existing MoQ implementations, including `moq-rs` and `moq-js`[2], both of which are actively developed and used within the working group.

To minimise disruption to the primary application logic and ensure non-blocking behaviour, the system employs a multi-threaded architecture. A dedicated thread handles the lifecycle of the broker connection and message queuing, thereby isolating event logging tasks from the main execution path. This design ensures that capturing and transmitting telemetry data does not impede media transport performance, especially in latency-sensitive contexts. By abstracting away the complexity of qlog formatting and network transmission, the interface enables rapid and consistent observability integration across diverse MoQ-based projects.

Crucially, this component is also responsible for assigning precise timestamps to each log entry upon capture, aligning with the qlog specification's requirement for high-resolution temporal tracking. By decoupling event capture from delivery and leveraging parallelism, this architecture ensures scalable, low-overhead logging suitable for real-time media transport environments.

6.2.2 Logger Broker

To facilitate the efficient and scalable transfer of logging data from the instrumentation interface to the backend infrastructure, the system introduces an intermediate broker component. The instrumentation interface connects to this broker using a designated stream identifier, which is supplied explicitly during the logging API call. This identifier serves as a routing key, enabling the backend system to correctly associate incoming logs with their corresponding stream contexts and subsequently retrieve them as needed. Allowing the user to log to multiple streams from one single MoQ implementation also allows for splitting or filtering certain parts of the logging data. One could, for example, split the control channel messages from the content streaming logs, allowing for a more precise grouping and a more deliberate scaling of parts of the system. The option then poses itself to send the control channel data to a weaker broker, while sending the large amounts of video streaming logs to a more scaled-up node.

The introduction of a broker into the logging pipeline is a deliberate architectural decision aimed at supporting horizontal scalability and decoupling log production from log consumption. As further elaborated in Chapter 5, this decoupling ensures that the system can handle high-throughput logging scenarios while maintaining low overhead on the instrumented applications.

To minimise performance impact and ensure reliable delivery under variable network conditions, the broker should leverage a lightweight protocol as mentioned in Section 5.1.4. This choice of protocol should align with the broader goals of the system, namely, enabling consistent and scalable observability without disrupting media transport performance. MQTT emerges as a strong candidate due to its lightweight nature and consistency, offering adequate scalability for moderate-sized deployments. These qualities align well with the system's need for low overhead and reliable delivery in early-stage or edge environments.

[2]The Media over QUIC working group utilises two main components for their implementation: moq-rs: `https://github.com/kixelated/MoQ-rs`, moq-js: `https://github.com/kixelated/MoQ-js`

As the scale of the system increases, particularly in scenarios involving numerous concurrent streams or high-frequency logging events, the limitations of MQTT may become more pronounced. Specifically, MQTT's constrained message throughput and limited durability mechanisms can pose challenges in environments demanding robust data persistence, advanced querying capabilities, and large-scale analytics.

To accommodate such scaling demands, the architecture supports the integration of a more powerful distributed streaming platform, for example, Apache Kafka. Kafka is designed for high-throughput, fault-tolerant log ingestion and is well-suited for backend systems requiring durable, replayable log streams and real-time processing capabilities.

While MQTT provides a lightweight solution for early-stage deployments, more advanced scenarios may require increased throughput and durability. In such cases, the system could be extended with a downstream Kafka integration. A bridging component might subscribe to MQTT topics and forward messages into Kafka, enabling high-throughput, replayable log streams. This hybrid approach retains MQTT's low overhead at the edge while allowing Kafka to handle backend scalability. Such an extension would be particularly valuable as MoQ adoption grows and observability demands increase.

By decoupling ingestion from long-term processing in this way, the system maintains flexibility: MQTT remains the interface for clients, while Kafka enables scalable back-end operations. This layered design ensures that the logging infrastructure can evolve in response to operational demands, without requiring intrusive changes to the client-side instrumentation.

6.2.3 Log Serialiser and Store

The logging backend component subscribes to the designated logging streams via the message broker. Its primary role is to continuously consume and process log events as the instrumentation interfaces publish them. This backend is designed to perform multiple critical functions that support both real-time monitoring and retrospective analysis.

First, incoming qlog events are durably written to a log store chosen for its ease of evolution at small scale and its capacity to grow later. For initial deployments, MongoDB offers an excellent fit. Its schemaless, document-oriented design accommodates rapid iteration without requiring migrations, which suits the evolving nature of MoQ particularly well. The low setup overhead and inherent flexibility enable developers to focus on protocol instrumentation and experimentation without being encumbered by rigid data models. This makes MongoDB not merely sufficient but well-aligned with the goals of early-stage development. As the system grows and requirements evolve, such as higher ingest rates, increased trace volumes, or more advanced analytical needs, PostgreSQL, optionally extended with TimescaleDB, provides a natural progression. The use of a JSONB column ensures continuity, allowing qlog data structures to remain consistent across both backend options. This step adds native time-series partitioning and compression, and exposes the full expressive power of SQL for later analysis, all without altering the higher-level logging API.

Second, the backend acts as an intermediary between the log store and subscribed frontend clients. It actively forwards log events from specific streams to the frontend interfaces, enabling real-time visualisation of protocol activity. This live update mechanism provides valuable insight into ongoing MoQ sessions, supporting debugging and monitoring efforts during runtime.

Third, the backend exposes a query interface to the frontend, allowing users to retrieve historical logs from a specific stream over a defined time window. Upon receiving such a request, the backend queries the database, filters the relevant events based on timestamp and stream identifier, and groups the results according to their vantage point. It then constructs a well-formed qlog file in accordance with the qlog specification, which is subsequently made available for download through the frontend interface. This enables post-mortem analysis using qlog-compatible tools and promotes consistency across the system's observability pipeline.

6.2.4 Visualisation Frontend

This component allows the user to choose a logging stream to listen to. Doing so starts the pushing of events on that logging stream from the backend to the frontend. When events arrive, they get parsed and visualised correctly so the user can interpret them.

The visualisation frontend provides an interactive interface for users to monitor and analyse Media over QUIC sessions in real time. It enables developers to select a specific logging stream of interest, thereby initiating a subscription process. Once a stream is selected, the backend starts pushing relevant event data to the frontend through a live connection. Upon arrival, incoming events are parsed and interpreted according to the qlog event schema. The frontend maps these events to appropriate visual elements, offering structured and intuitive representations that facilitate user comprehension. This includes, for instance, timelines of control message exchanges, datagram flow, and vantage point-specific activity. The visualisation logic ensures that the MoQ events are handled in a meaningful and consistent manner.

In addition to real-time inspection, the frontend offers the capability to download the current set of displayed events as a qlog-compliant file. This allows users to perform in-depth offline analysis using external qlog tools or to archive session logs for documentation or debugging purposes. The combination of live visual feedback and on-demand archival export makes the frontend a central component in the system's observability and usability pipeline.

Moreover, the frontend also supports the upload of locally stored qlog files that adhere to the schema. Developers can load previously saved event streams into the interface, enabling retrospective exploration and analysis without requiring a live connection to the backend. This dual functionality, supporting both live and offline data, enhances the flexibility of the system and facilitates a wide range of usage scenarios, from active debugging to post-mortem protocol analysis.

Real-time media protocols like MoQ generate a high volume of control and data events, distributed across multiple streams and actors. Debugging such behaviour, especially when involving prioritisation, object loss, or asynchronous control flows, can be extremely difficult by relying solely on raw logs. A dedicated visualisation layer is therefore essential, not just for user comprehension, but for protocol validation, implementation testing, and performance tuning.

The design of the visualisation frontend draws inspiration from existing tools such as qvis[3], which offers powerful transport-level visualisations based on standardised qlog traces and has a similar goal to what we want to achieve with the visualisation frontend. The qvis tool demonstrates the value of structured logs for understanding QUIC's behaviour, particularly in terms of packet transmission, loss recovery, and congestion control. However, qvis is primarily oriented toward QUIC's transport semantics and is not equipped to visualise the session-level constructs introduced by MoQ. It operates on per-endpoint traces and lacks support for analysing control messages such as `SUBSCRIBE`, `ANNOUNCE`, or `FETCH`. Moreover, it provides no correlation between related events across different endpoints, an essential capability for debugging distributed MoQ sessions. To overcome these limitations, the visualisation frontend presented in this thesis extends the qlog format with MoQ-specific events and introduces views tailored to protocol semantics, enabling a richer understanding of session-level behaviour.

[3] `https://qvis.quictools.info`

6.3 Data Model and Flow

To enable structured, extensible, and scalable logging for MoQ sessions, a clear and consistent data model is essential. This section details the internal data representation and event flow that underpin the proposed logging architecture. It outlines how MoQ-specific protocol semantics are captured, transformed, and propagated through the system using an extended qlog schema. By defining the structure and movement of data from event capture to visualisation, this model ensures coherence between system components and facilitates accurate, low-latency introspection of session behaviour.

6.3.1 Extending qlog

While the adoption of qlog provides a robust and standardised foundation for capturing transport-layer events, effectively debugging and analysing Media over QUIC requires visibility into application-level interactions specific to the protocol itself. As established in Chapter 4, qlog excels due to its structured format, established tooling, and inherent extensibility, features that make it an ideal choice for logging complex network protocols like QUIC. However, the base qlog specification primarily defines events relevant to the transport layer, e.g., QUIC packet handling and connection states. It naturally lacks predefined event types and semantic structures tailored to the unique operations and mechanisms of MoQ, such as track announcements, subscriptions, or object prioritisation. To achieve comprehensive observability for MoQ systems and facilitate targeted analysis, it becomes essential to leverage qlog's extensibility. Therefore, this section outlines a proposed extension to the qlog format, defining a dedicated set of event types and data structures specifically designed to capture the critical dynamics of Media over QUIC sessions within the standardised qlog scheme.

Namespace Design for MoQ Events

To maintain modularity and ensure future-proofing, two distinct qlog namespaces have been defined:

- **moq**: This namespace contains standardised and protocol-compliant event types that directly reflect the operations described in the MoQ transport specification, such as control messages (e.g., `subscribe_ok`, `announce`, `fetch_cancel`), session state transitions, and stream classifications. For clarification, the full list of control message types is provided in Table 1 of the appendix.
- **moq-custom**: This auxiliary namespace captures experimental, vendor-specific, or yet-to-be-standardised events that are nevertheless relevant to the MoQ ecosystem. It allows implementers to log custom behaviour (e.g., internal scheduling decisions or fallback logic) without conflicting with future official extensions.

This dual-namespace approach aligns with qlog's extensibility guidelines as defined in Section 8.2 of the main qlog schema draft [MNSP25]. It supports the evolution of MoQ while preserving compatibility with standard analysis tools.

Unified Logging Through Multi-Endpoint Traces

Another enhancement this work introduces involves structuring qlog files to incorporate multi-endpoint traces, enabling correlated observation of a full MoQ session across clients, relays, and servers. While qlog already supports the concept of trace objects (each encapsulating a single logical view with its own vantage point), our extension advocates the use of a traces array within a single qlogFile object to group synchronised logs across endpoints. This enables powerful end-to-end analysis scenarios, such as matching subscribe requests at clients with `subscribe_ok` responses from servers, or tracing the relay path of announce messages through intermediate nodes.

Each trace entry is annotated with the `vantage_point` field (as per Section 6 of the qlog schema), clearly distinguishing its role (client, server, relay) and allowing event correlation across the distributed MoQ session.

Figure 6.2 shows the overall hierarchy of the qlog file and how the traces can be used to distinguish the different vantage points. We see that the top level is the qlog file, which contains multiple traces, one for each of the vantage points. Each trace then contains the events that were logged from this vantage point.

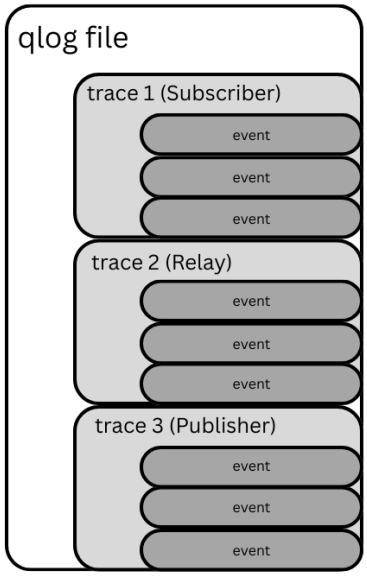


Figure 6.2: Layout of the qlog file and how it is used in the proposed system. Every file contains a trace for each of the vantage points, which each contains the events logged from that vantage point.

MoQ qlog Event Definitions Draft

After the initial implementation phase, a formal proposal for MoQ-specific qlog events was published in the IETF as an Internet-Draft [PE25]. This draft introduces a broader event vocabulary for MoQ, organised around key operational stages such as control message lifecycle, object datagram handling, and subgroup composition. Notable event types include:

- **MoQ:control_message_created and MoQ:control_message_parsed**: These define the creation and interpretation of control messages in a structured and type-aware manner.
- **MoQ:object_datagram_created, MoQ:fetch_header_parsed, and similar events**: These focus on deeper packet-level dynamics and subgroup/data stream orchestration.
- Stream and object lifecycle events, reflecting the granular structure of the MoQ data model as defined in the MoQ transport specification.

The official schema uses the structured Concise Data Definition Language (CDDL) [BVB19] and aims for integration with existing qlog tooling ecosystems such as qvis. While more comprehensive in scope, this draft emerged after the initial phases of this thesis work. Consequently, the current implementation diverges somewhat in naming conventions, event granularity, and data layout, but remains compatible in principle thanks to qlog's flexible schema extension points. Future work may involve aligning the custom logging system more closely with this proposed standard, especially as it stabilises and gains broader adoption. Until then, the current focus remains on delivering practical observability for control flow in evolving MoQ environments.

6.3.2 Serialising MoQ Events to qlog Format

When creating the system, the deliberate choice was made to adapt the event logging to follow the qlog format. From the start, when the user makes a logging call in their implementation code, they are forced to follow the qlog format. This is to, first of all, create a mental model for the user, allowing them to better understand the complete data flow of the system, but primarily because qlog is a logging scheme that has proven itself to be useful and is widely adopted to be the standard when logging QUIC connections.

As previously noted, the system requires users to utilise the qlog format for their data input. Events are logged when a user calls the logging function of the `Logger` class within their implementation code. This `Logger`'s function accepts an object structured as a qlog event. The example presented in Listing 6.1 demonstrates the format of a log message. When logging, users must provide an object containing the `eventName`, `vantagePointID`, `stream`, and `data` fields. Here, the `eventName` and `data` fields map directly onto the respective `name` and `data` fields of a standard qlog event. The `stream` field indicates the specific stream for sending the log message to the server, while the `vantagePointID` acts as a unique identifier employed both in the visualisation process and for organising logs by location into distinct traces within the qlog file.

```
{
    eventName: "subscribe-received",
    vantagePointID: "PUBLISHER",
    stream: "logging-stream",
    data: {
        message: msg,
    }
}
```

Listing 6.1: Logging "subscribe-received" on a publisher in a JavaScript environment

The `data` field within a qlog event object provides a flexible mechanism for including rich, event-specific information beyond the core metadata. Rather than enforcing a constrained schema, qlog leverages an open-ended, key-addressable structure in which each key is a string. The associated value may be any JSON-compatible type, including strings, numbers, booleans, arrays, or nested objects. This generality enables developers to log arbitrary, contextually relevant information without rigid constraints on format or structure.

This design choice directly supports the goals of extensibility and adaptability. Developers implementing event logging for a specific use case, such as Media over QUIC, can define and emit structured contextual data without modifying the underlying schema or serialisation mechanisms. Consequently, the logging system remains both broadly interoperable and future-proof. Should the nature of the logged events evolve (e.g., to include more complex diagnostics or metadata), the existing logging pipeline remains functional without requiring systemic changes.

Additionally, using native JSON types ensures natural integration with analysis tooling. Data consumers can access known fields via string keys and, depending on the application's needs, may interpret or parse values directly (e.g., treating a value as an integer, parsing a nested object, or applying domain-specific logic). This encourages the development of tooling that is both robust and capable of leveraging domain semantics.

After the data is captured and formatted as qlog data on the client side, it gets transported over to the back-end, where it gets sent over to the visualisation front-end as well as saved in a database. Data gets saved on the server to serve the developer later when they want to download the qlog file from the tool for further analysis. A developer or analyst can also upload

a qlog file directly into the visualisation tool. Doing so will only load the file locally and will never save the file elsewhere.

Chapter 7

System Proof of Concept

To show that the proposed system introduced in Chapter 6 is feasible and can result in valuable insights regarding the debugging and monitoring, as well as improve the explainability of the Media over Quic protocol, we built out a proof of concept following our findings. This is but one of many possible ways the proposal can be interpreted, and one should always choose what works best for their implementation. Choices made in this chapter are not definitive and can be altered if the system is scaled up or down.

For this proof of concept, we make use of the already existing implementation from the MoQ working group. At the start of the development process, this implementation followed draft 04 of the MOQT protocol, and thus, some more recent features were not tested. It should, however, be noted that because of the generic nature of the system and the open extension of the qlog format, newer versions of the protocol can still be visualised, and the system does not lose much of its value.

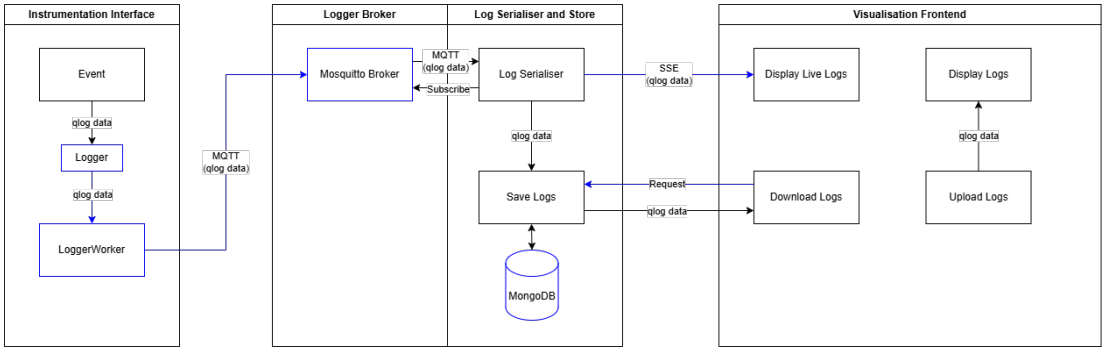


Figure 7.1: Updated architecture layout based on the exact implementation of the proof of concept. Blue-marked objects show adaptations made to Figure 6.1, which describes a more generic version.

Figure 7.1 shows an adapted version of Figure 6.1 regarding the system architecture. In the adapted figure, we include specific implementation details (marked in blue) to illustrate how the system architecture can be realised in practice. This chapter provides a comprehensive walk-through of a practical implementation of the logging system following this system architecture. It details how the components of the system interact to form a coherent, observable pipeline for MoQ sessions, bridging the theoretical architecture defined earlier with a real-world, functional prototype. Each section discusses a major subsystem, design choice, or integration challenge, offering both technical insight and rationale. Together, they demonstrate the flexibility of the proposed design and its viability in realistic settings.

7.1 Instrumentation Interface

The instrumentation interface is the framework layer that attaches to the existing implementation. It provides the user with an easy-to-use interface for logging events. In the context of the proof of concept, the instrumentation layer serves as the connection between the logging infrastructure and the operational components of the Media over QUIC system. Building on the architectural and technical foundations established in the implementation chapter, this layer exposes a set of abstractions and integration points that facilitate real-time interaction with MoQ streams and their associated qlog events. Its primary purpose is to provide a modular and extensible mechanism for capturing protocol-level events and translating them into structured logs that conform to the MoQ-specific qlog schema. Ultimately, the design of the interface layer aims to demonstrate how standardised logging can be seamlessly integrated into next-generation streaming protocols, offering visibility into session-level behaviour while maintaining the scalability and performance characteristics inherent to MoQ.

We will now introduce some choices that were made following the proposed system. Because this proof of concept makes use of the existing Media over QUIC implementation from the IETF working group, the instrumentation interface has to be implemented in both the Rust language and in JavaScript (or TypeScript). This results in some language- or platform-specific choices that had to be made. Considering these different implementations, both adaptations follow the same ideas.

7.1.1 Singleton Instance

Both implementations introduce a new Logger class. An instance of this class allows for calling the logging function and sending log messages to the backend. This class follows the singleton design pattern. This ensures that not every call creates a new instance of the class, allowing all logging operations to occur over the same backend connection without the need to maintain a separate global connection variable. We also save memory space and improve performance, not having to create new instances of the class for every location in the MoQ implementation where a log message is sent. In the context of MoQ, where multiple asynchronous and concurrent events may require logging simultaneously, the singleton pattern guarantees consistent and thread-safe access to the logging backend without introducing synchronisation issues or resource contention.

Listing 7.1 shows the function used for handling the singleton pattern. When the instance is not yet created, we create a new instance of the Logger class. Otherwise, we pass the already created instance. In Rust, we can implement a singleton pattern using the `lazy_static` crate. We can see this crate in use in Listing 7.2. It allows us to define a value (such as an instance of a class or struct) that is initialised exactly once and then shared across the whole codebase. The `Mutex` in this code example ensures serial execution of the logging functions, ensuring thread safety.

```
static getInstance() {
    if (!Logger.instance) {
        Logger.instance = new Logger()
    }
    return Logger.instance
}
```

Listing 7.1: Code for implementing the singleton design pattern in a JavaScript or TypeScript environment.

```
lazy_static! {
    pub static ref LOGGER: Mutex<MqttLogger> = Mutex::new(
        MqttLogger::new());
}
```

Listing 7.2: Code for implementing the singleton design pattern in a Rust environment.

7.1.2 Connection Thread

To minimise the computational overhead on the implementation thread, threading was employed to handle logging operations asynchronously. This design choice is particularly important in a streaming context, where delays in media processing or transmission could result in degraded user experience. Both implementations use their respective threading models to ensure that the connection with the backend, as well as the transmission of log messages, occur on a separate thread, thus avoiding blocking the main execution flow.

When a user creates a new log message, a minimal piece of code is run to capture and fill the log message with the correct time and send it to the logging thread. While the implementation is logging, the logging thread listens for these logs. The first time the thread starts up, the connection is opened with the broker connecting it to the backend. Once established, the thread remains active, asynchronously handling the transmission of subsequent log messages.

Listing 7.3 shows that the constructor of the `Logger` class immediately initiates a new Web Worker. Web Workers offer a straightforward mechanism for running scripts in background threads, thereby preventing blocking of the main thread. In this design, the Web Worker is responsible for executing the remaining logic necessary to connect to the broker via MQTT. An equivalent approach in Rust is illustrated in Listing 7.4, where a new thread is spawned within the constructor. The connection logic is encapsulated inside this thread, utilising the `move` keyword to transfer ownership of the required variables into the thread's scope.

Although Web Workers and Rust threads provide similar concurrency benefits, they differ fundamentally in their execution models and communication mechanisms. In JavaScript, the main thread communicates with the Worker via message passing, while in Rust, threads can share memory and coordinate using channels or synchronisation primitives.

```
constructor() {
    this.worker = new Worker(new URL("./loggerWorker.js", import.
        meta.url), { type: "module" })
}
```

Listing 7.3: Initialisation of a new Worker as a thread in the JavaScript implementation of the Logger

```
thread::spawn(move || {
    // The entire MQTT client connection and buffering loop
    // happens inside this closure
});
```

Listing 7.4: Initialisation of a new thread in the Rust implementation of the Logger

7.1.3 Buffering Messages

Because logging messages can be accumulated in a rapid fashion, a system was added to both logger implementations to buffer them. We can introduce a buffer like this because the logging visualisations do not have to be updated in real-time to be valuable. Another factor allowing us to do this is the timings of the events, which are registered before entering the buffer.

To implement this logic in JavaScript, we make use of the `setInterval` function, as shown in Listing 7.5. We place the logic for buffering the events and sending them to the backend within this function, which then gets called once every second, so as not to overload this backend. The buffer is a JavaScript dictionary which, for every different stream identifier passed in the `stream` field of the events, contains an array with all the buffered events for that stream. This principle is visible in Listing 7.7. Although in a local scenario the buffer will, most of the time, be filled with only one stream identifier, it allows for the buffering of events when streams should be split up. To implement this same principle in Rust, we choose to import `interval` and `Duration` from the Tokio crate. They allow us, much like the JavaScript implementation, to wait for a second, accumulate the messages in the buffer, and then transmit them. They can be seen in use in Listing 7.6.

```
setInterval(() => {
    // Logging and buffering logic happens inside this closure
}, 1000)
```

Listing 7.5: setInterval() function which executes the code inside the code block once every second.

```
let mut interval_var = interval(Duration::from_secs(1));
loop {
    interval_var.tick().await;
    // Logging and buffering logic happens inside this closure
}
```

Listing 7.6: Usage of the interval function and the Duration struct for executing the code block every second.

```
{
    "logging-stream1": [event1, event2],
    "logging-stream2": [event3, event4],
}
```

Listing 7.7: Buffer content structure example.

7.1.4 The Logging Interface

To make the logging experience as smooth as possible for the user, the interface for doing the actual logging is kept as straightforward as possible while still offering the needed flexibility for logging in typical qlog fashion and logging on multiple streams. For the JavaScript or TypeScript implementation, seen in Listing 7.8, a user starts by getting the instance of the Logger class by calling `getInstance`. They then log an event by passing a formatted object to the `logEvent` function. This object, as mentioned in section 6.3.2, is filled with the `eventType`, `vantagePointID`, `stream` and `data` fields to follow the qlog format. A similar logic applies to the implementation in Rust. Here we first create the log message as an object of the `LogMessage`

class. We fill this message with the same fields as before. Because of Rust's strongly typed syntax, we pass the data as a hashmap. The last step is using the `mqtt_log` function to send over the message to the broker.

```
Logger.getInstance().logEvent({
    eventType: "subscribe-received",
    vantagePointID: "PUBLISHER",
    stream: "logging-stream",
    data: {
        message: msg,
    },
})
```

Listing 7.8: Logging of "subscribe-received" on a publisher in a TypeScript environment

```
let log_message = LogMessage {
    eventType: format!("subscribe-received", msg.name()),
    vantagePointID: "RELAY".to_string(),
    stream: "logging-stream".to_string(),
    data: HashMap::from([
        ("message".to_string(), msg)
    ]),
};
mqtt_log(log_message);
```

Listing 7.9: Logging of "subscribe-received" on a publisher in a Rust environment

7.2 Logger Broker

The broker in the proposed system serves as the intermediary component between the instrumentation interface and the backend. Its primary functions, elaborated in Chapter 6, Section 6.2.2, include enabling straightforward horizontal scalability and facilitating the transfer of messages via streams. For the proof-of-concept implementation, MQTT was selected as the message transfer protocol due to its lightweight nature, efficient connection handling, and compatibility with WebSocket connections for browser-based clients. Furthermore, MQTT's publish-subscribe paradigm aligns conceptually with MoQ's design, enabling clients to push messages in an efficient and structured manner.

The broker utilised in this implementation is *Eclipse Mosquitto*, an open-source MQTT broker known for its lightweight footprint and ease of deployment, making it an optimal choice for the envisioned system architecture.

A Docker container with two open ports is used to deploy the broker. The first port supports raw MQTT connections, which are utilised by the Rust implementation and the backend components. The second port is configured for WebSocket connections, enabling communication with web-based clients. The broker listens on these ports and handles both `subscribe` and `publish` operations.

Listing 7.10 demonstrates how MQTT is employed in the JavaScript/TypeScript implementation. A client connection is first established, after which the `publish` function can be invoked to send messages to the broker, each identified by a specific `stream_id`.

The equivalent implementation in Rust is shown in Listing 7.11. Here, the client is initialised using a `create_options` configuration, followed by establishing a connection to the broker using these options. Messages are then published in a similar fashion by calling the `publish` method on the client instance.

```
const client = mqtt.connect(BROKER_ADDRESS);

client.publish(
    stream_id,
    JSON.stringify(msg),
);
```

Listing 7.10: Creation of the MQTT connection with the broker and sending of a message to the broker in a JavaScript or TypeScript environment.

```
let create_options = mqtt::CreateOptionsBuilder
::new()
.client_id(CLIENT_ID.to_string())
.server_uri(BROKER_ADDRESS)
.persistence(None)
.finalize();

let client = match mqtt::Client::new(create_options) {
    Ok(client) => client
};
let msg = mqtt::Message::new(
    stream_id,
    json_payload.as_bytes().to_vec(),
    mqtt::QoS::AtLeastOnce
);
client.publish(msg);
```

Listing 7.11: Creation of the MQTT connection with the broker and sending of a message to the broker in a Rust environment.

To enable the receiving of messages transmitted via the broker, the backend also establishes a connection to the broker. Because MQTT uses the publish/subscribe paradigm, it suffices to create this connection and subscribe to the broker. We can see how this subscription gets created in Listing 7.12. It starts off by creating the connection. On this connection is where we can call the `subscribe` function with the stream identifier we want to listen to. From then on, we can catch messages using a listener and handle the messages accordingly.

```
const client = mqtt.connect(BROKER_ADDRESS, {
    clientId: CLIENT_ID,
    clean: true,
});
client.on("connect", () => {
    client.subscribe(STREAM_ID);
});
client.on("message", async (topic, payload) => {
    // Handle the received message
});
```

Listing 7.12: Creating the connection to the broker, subscribing to a stream and listening for messages on that stream

7.3 Log Serialiser and Store

The log serializer and storage component is implemented as the system's backend service. The component handles the retrieval and storage of messages, as well as their transmission to the visualisation frontend. The retrieval was already handled in the previous section, where the connection with the broker was also explained. After the messages are received, they are sent to the database. For this proof of concept, MongoDB was chosen to store the events due to its flexibility in handling semi-structured data and its ease of integration with JSON-based message formats, which aligns well with the qlog event structure.

Listing 7.13 illustrates the initialisation of the MongoDB connection and the definition of the data schema used for storing log events. The database is connected using the `mongoose` library, which simplifies interactions with MongoDB in Node.js environments. The schema defined by `StreamEventSchema` reflects the structure of a typical qlog event as adapted for MoQ logging. It includes fields such as `stream_id`, `event_type`, `timestamp`, `vantagePointID`, and `data`, the latter of which is declared as `Mixed` to allow for the flexible, semi-structured nature of qlog event data. This design accommodates the heterogeneous payloads typical of qlog-formatted messages while maintaining compatibility with MongoDB's document-based storage model. The schema thus ensures consistent data ingestion and facilitates downstream querying and analysis by the visualisation component.

To ensure scalable and performant log ingestion in high-throughput environments such as Media over QUIC, the backend adopts a dual-buffering strategy. At its core, each media stream is associated with a dedicated in-memory buffer, maintained in a JavaScript `Map`. Incoming log messages, received via MQTT, are parsed and appended to their respective stream buffers. Once the buffer for a given stream exceeds a predefined threshold, it is flushed to MongoDB in bulk using the `insertMany` operation. This reduces write amplification and I/O overhead, while still preserving timely persistence.

As shown in Listing 7.14, the system supports real-time log delivery to the frontend via a Server-Sent Events (SSE) endpoint exposed at `/events`. Clients can subscribe to a specific `streamId`, upon which the server registers their connection and begins streaming relevant events. To prevent frontend overload, an additional receive-side buffering mechanism is introduced. If a batch of events surpasses a configurable size limit, they are aggregated into a single composite event before being pushed over the SSE connection. This combination of backend buffering and SSE stream management ensures a balance between latency, throughput, and resource efficiency across both development and deployment scenarios. Furthermore, the system detects when the last client for a stream disconnects and invokes a final buffer flush to ensure no data is lost.

The last notable function of the backend is retrieving the logs from the database. This can be done, as visible in Listing 7.15, by performing a GET request to `/api/events` and providing a stream identifier. The URL also contains the start and end times of your query. These will then be used to fetch the correct events from the database. Ideally, there would also be a system limiting who can fetch events from which streams, but implementing this was not a high priority for this system, as it is a proof of concept.

```
mongoose.connect(process.env.MONGODB_URI || "mongodb://localhost
    :27017/stream_logs", {
    useNewUrlParser: true,
    useUnifiedTopology: true,
});

const StreamEventSchema = new mongoose.Schema({
    stream_id: String,
    event_type: String,
    timestamp: Number,
    vantagePointID: String,
    data: mongoose.Schema.Types.Mixed,
});

const StreamEvent = mongoose.model("StreamEvent",
    StreamEventSchema);
```

Listing 7.13: Creation of the MongoDB store.

```
app.get("/events", (req, res) => {
    res.setHeader("Content-Type", "text/event-stream");
    res.setHeader("Cache-Control", "no-cache");
    res.setHeader("Connection", "keep-alive");

    const streamId = req.query.streamId || "default";
    clients.push({ response: res, streamId });

    req.on("close", () => {
        clients = clients.filter(client => client.response !==
            res);
        if (!clients.some(client => client.streamId === streamId)
            ) {
            flushStreamBuffer(streamId);
        }
    });
});
```

Listing 7.14: Setup of the SSE endpoint for the the visualisation clients.

```
app.get("/api/events/:streamId", async (req, res) => {
    const { streamId } = req.params;
    const { startTime, endTime } = req.query;

    if (activeStreams.has(streamId)) {
        await flushStreamBuffer(streamId);
    }

    const query = {
        stream_id: streamId,
        timestamp: {
            $gte: parseInt(startTime),
            $lte: parseInt(endTime),
        },
    };

    const events = await StreamEvent.find(query).sort({ timestamp
        : 1 });
    res.json({ events });
});
```

Listing 7.15: Querying historical stream events based on time filters.

7.4 Visualisation Frontend

As discussed in Section 6.2.4, the frontend can listen to logging streams by identifier. When clicking the `listen` button, the client registers itself with the backend and opens the SSE connection. Starting from that point, the events are pushed to the frontend and displayed as they come in. The visualising of the events happens in a dashboard, where every time a new event is registered, the visualisations update accordingly. These visualisations are built from scratch using D3.js, offering creative freedom but coming at the cost of slower development time. In this section, we will go over every part of the front-end visualisation tool and all the features that the visualisations include.

7.4.1 Control Bar

At the top of the visualisation interface, several controls are arranged as seen in Figure 7.2. From left to right, we see a listening toggle with an accompanying input field, an upload button, and a download button with its own input field. On the left, the first input field allows users to specify a stream identifier for active listening. When the adjacent button is pressed, the system initiates real-time monitoring of incoming events for the specified stream, updating visualisations dynamically as events are received. Centrally placed is the upload button, which permits users to submit qlog-formatted files, either previously generated by the system or created externally. Upon upload, the events in the file are parsed and loaded chronologically, again triggering dynamic updates to the visualisations based on their contents. On the right-hand side, the download button allows users to export the currently visualised events. A user specifies the desired stream via the associated input field, after which the system performs a fetch operation as described in Section 7.3, bundles the events into a qlog-compliant file, and allocates each vantage point its own trace within the output.

Figure 7.2: The controls which are located at the top of the page of the visualisation frontend. In this example, when pressing the "Start Listening" button, the frontend will start to receive events on the "logging-stream" stream and adapt the page correctly. Uploading a qlog file using the upload button will result in the file being parsed and the events being displayed correctly. The last button on the right allows the user to download a qlog file of all the events currently displayed.

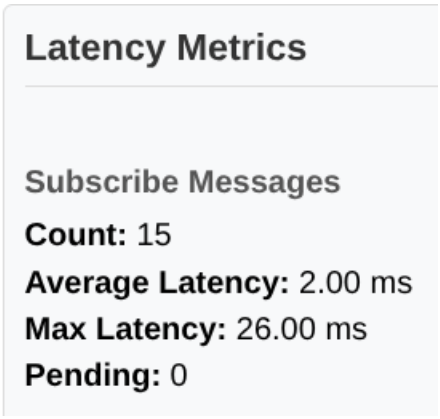


Figure 7.3: Latency measurement example for subscribe messages. The count represents the number of subscribe messages sent over in the current time period. The average and max latency give more insights into the latency values themselves. Pending, in this figure, displays the number of subscribe messages that are sent but never received.

7.4.2 Latency Measurement zone

Just below the control bar on the visualisation tool are the latency measurements. These will calculate the latency per event type. The mapping of messages can only be done when events are logged when they are sent at the publisher and when they are received at the subscriber. We can, using that same system, also log individual hops, but this will require a more granular visualisation.

Figure 7.3 shows how these latency measurements are displayed on the page. In this example, it shows the values which were calculated from the subscribe messages. This latency measurement zone is populated with information about the number of subscribe messages, the average latency, the maximum latency and the pending messages. Using the average latency measurement, the user can define if there are any immediate problems with the system. When greeted with an abnormally high average latency, they might be able to relate this to an outlier by looking at the max latency and count values. A low count value with a large outlier will inherently result in a higher average latency. Finally, the pending row of the measurements can give insights into problems that the messages have with propagating through the system. When this number is unusually high, it means that messages are getting lost somewhere in the setup, and the user can try to identify where using the other visualisations.

7.4.3 Basic Event Chart

The most basic visualisation on this dashboard is the event chart. When a new event is registered, it gets placed on this chart as a circle, grouped by event type and displayed over time. The chart starts off bare, with no registered events or event types. As events come in, they get checked for their type. If a new type is noticed, a new value on the y-axis is created, and the event is placed corresponding to their timestamp and event type. These circles are hoverable, showing the exact information that was passed to the frontend in qlog format. The user can

also zoom in on the chart, resulting in a more precise analysis. When events on the chart are near each other in terms of timestamp and are of the same event type, this would result in them being placed on top of each other and thus not providing the correct sense of scale at first glance. To omit messages being visually left out, they get staggered on top of each other, visually resulting in a group of messages, where each of them is still hoverable.

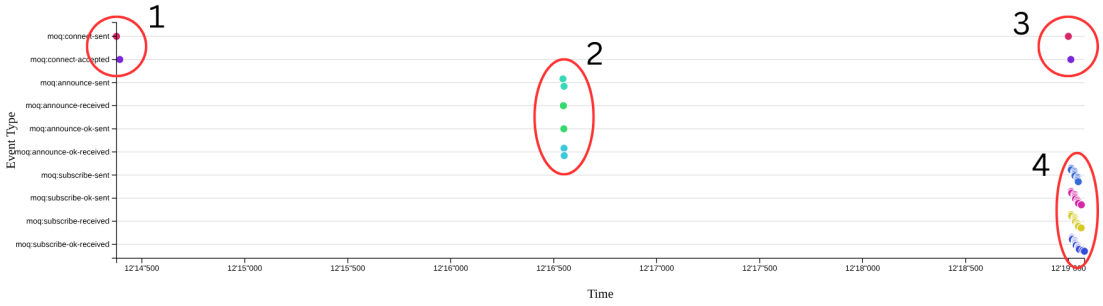


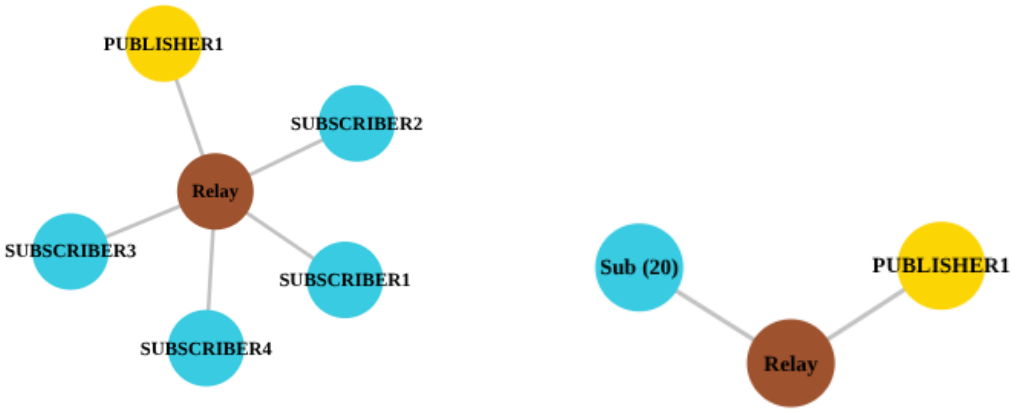
Figure 7.4: This figure demonstrates how the basic chart of the visualisation tool can display incoming or saved events. It currently displays the control messages for setting up a simple MoQ stream. First, (1) the publisher connects to the relay and (2) announces its streams. This is then followed by the (3) subscriber connecting to the relay and (4) subscribing to several streams.

Figure 7.4 demonstrates an example of the basic chart on the visualisation tool. The system at the time of this example is only logging control messages sent over the bidirectional control stream. The example is composed of the publisher connecting to the relay and announcing its different tracks. This is followed by a subscriber connecting to the same relay and subscribing to several tracks. As mentioned before, the event types are displayed on the y-axis together with their respective qlog extended namespace. The events themselves are ordered chronologically. When multiple events are in close proximity to each other, they are grouped visually, which can be seen in zone 4 of the figure.

7.4.4 Connection Graph

The second graph that is visible on the dashboard is a connections graph. Because of MoQ's complex scaling features using relays, we create value by visualising the network of participants and relays. To make connections between relays and clients, we examine the `connect-sent` and `connect-accepted` messages that are sent over the control channel. The result is then a graph that displays all entities that are relevant to the stream and their connections. Based on the `vantagePointID` field in the event message, it is possible to determine if the entity is a relay, a subscriber or a publisher and colour them accordingly, resulting in a more structured visualisation. Using the visualisation, the user can see if connections are working and create a model of the setup they are using.

Figure 7.5 demonstrates an example of the connections graph. In the first MoQ setup (a), four subscribers connect to a single relay as well as a single publisher. Whenever a new entity joins, it gets placed on the graph in the form of a circle. In this process, the tool tries to detect if the entity is a subscriber, a publisher or a relay based on the vantage point identifier. It then proceeds to give them a corresponding colour to make distinguishing the different types easier. In this example, blue is used for subscribers, yellow is used for publishers and brown is used for relays. This type recognition is also used for better labelling of the nodes. The vantage point identifier is used to give the node a meaningful label. When the identifier length surpasses a threshold, it gets reduced to its type as a label. This is to prevent large labels from causing problems in interpreting the graph, as a tradeoff for losing the identifier information.



(a) Connection graph comprising a single publisher and four subscribers connected to a relay.

(b) Connection graph comprising a single publisher and twenty subscribers connected to a relay.

Figure 7.5: Figure (a) and (b) show the dynamic nature of the connections graph. When the subscriber count, connected to a single relay, reaches five, they get bundled in a single circle to reduce overcrowding of the chart with nodes.

When connection events are logged (`moq:connect-sent` or `moq:connect-accepted`), the tool tries to find the correct links between the different nodes in the graph based on these connection log messages. These connections are displayed as grey lines between the nodes.

Because of the intention of supporting large-scale setups with plenty of different nodes, an aggregation technique is used. Nodes that are of the same type and connect to the same relay are bundled into one singular node if their count surpasses a threshold (in the current system, this is set to five). As not to lose any information in this process of aggregating, the node is clickable, which opens a pop-up that contains the vantage point identifiers of the included clients.

7.4.5 Message Sequence Chart

While messages are received, if they are filtered out as control messages 1, they become visible on the message sequence chart. When a new vantage point is detected in the logs, a new entity in the form of a vertical line is displayed on the chart. Starting from that point, every time a new control message is sent to another vantage point, this is shown as an arrow starting from the sender and pointing at the receiver. This chart will grow vertically as more control messages flow in. Using this chart, the user can see how the connections are built and how the communication between the vantage points occurs.

Figure 7.6 shows the MoQ setup process in the message sequence chart. We see, once again, a publisher connecting to the relay, followed by a subscriber connecting to the relay. This chart, however, shows the direction the messages are going as well as the time they are sent and received. Using this chart, the user can follow the complete conversation between the different actors in a MoQ setup. After the connection is made, the publisher announces its streams to the relay, and the subscriber is able to subscribe to these streams.

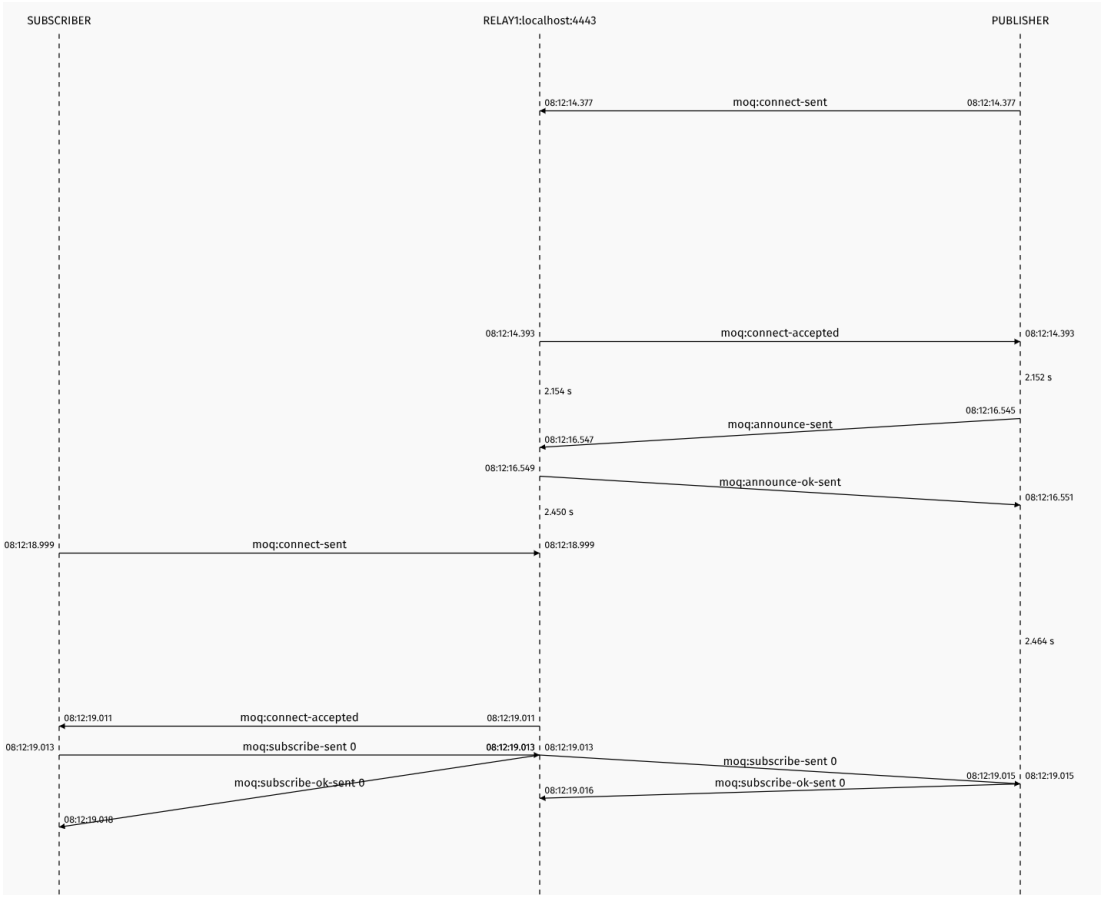
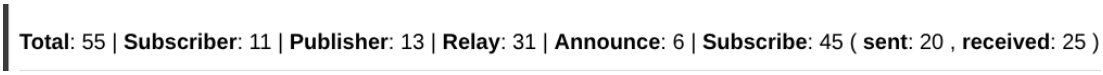


Figure 7.6: Example of message sequence chart which displays part of the control message conversation between the different actors. Messages are sent from one entity to another and displayed with direction, timestamps and event types. This example follows the start of the stream setup, where a publisher connects to a relay and announces a stream, followed by a subscriber connecting to a relay and subscribing to said stream.

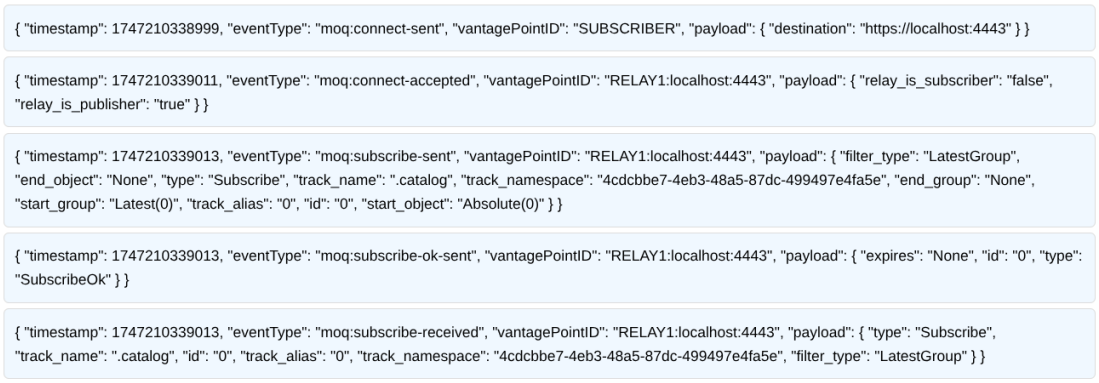
7.4.6 Event List

The last visualisation on this page is a list of all the events that are logged. This list expands every time a new event arrives and is ordered by timestamp. Events are placed here in a raw JSON format, giving the user the complete information they sent from their implementation. The value in this list lies in how it interacts with the other visualisations. Firstly, when clicking on a vantage point in the connections graph, all events sent from that vantage point will be highlighted in the event list, letting the user identify problems related to one vantage point or relate other facets specific to that vantage point. Second, when clicking on one of the events in the basic event chart, it highlights and moves the screen to that particular event. This could allow the user to get another view of where the event is located regarding the overall messages and give them more context in the event itself. This feature also works when clicking on a clustered group of events on this chart. When doing so, all events in this cluster will be highlighted instead of the one clicked on.

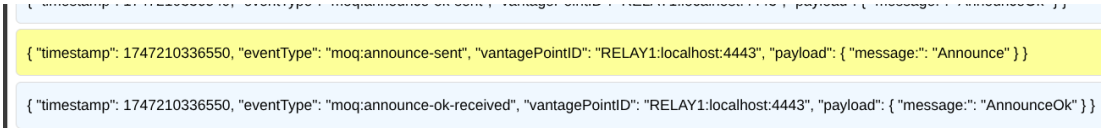
Figure 7.7 shows an example of the event list with all its different components. The first component, visible in Subfigure 7.7a, shows an example of the event list counters. These numbers represent the distribution of events next to the total number of events. This distribution is first divided into the subscribers, publishers and relays and shows how many messages each type of entity logged. Following this is the distribution of types of events, where currently the announce and subscribe event types are shown. Lastly, the subscribe type is split into sent and received messages. All of these counters can be clicked to apply a hard filter on the list. This filter will remove any logged events from the list that do not comply with the selected counter. The second Subfigure 7.7b shows the event list itself. Here, all events that come in are displayed in the format they are logged. The list itself grows downwards and can become too large to find or relate specific events, which is why there are several actions possible to aid in this process. The first action is shown in Subfigure 7.7c. It pictures the behaviour of the event list when the user clicks an event in the basic chart. When they do so, the window moves down to the specified event, and it is highlighted yellow. In case a user clicks on a bundled event in the basic chart, all events in that bundle are highlighted, and the window moves to the first event in that bundle it passes. The second action to change the event lists view is visualised in Subfigure 7.7d. Here, a user selected one or more nodes in the connections graph, resulting in the events logged by the unselected entities being greyed out. Both actions can be reset by performing a different action or by clicking a reset button in the bottom right of the window. All actions resulting in highlighting specific events in the list can be combined with each other as well as with the filters mentioned previously to further limit the number of events the user focuses on.



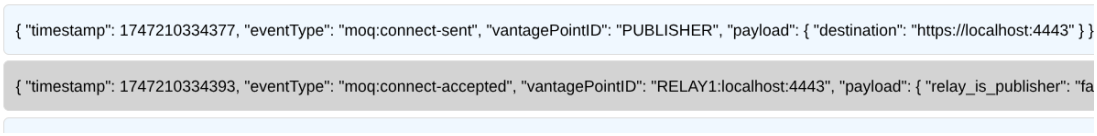
(a) The event list counters, located above the list itself. These counters show the total number of messages and how they are distributed. Clicking on one of these counters filters the list of events below.



(b) The basic visualisation of some of the events displayed in the data field on the page. These events are shown as they are sent over by the logging entities.



(c) A snippet of the event list with a highlighted event as a result of clicking the specific `announce-sent` event in the basic chart. The clicking action highlights the event as visible and moves the window to the location of the highlighted event on the page.



(d) A snippet of the event list with a highlighted event as a result of clicking the `PUBLISHER` node in the connections graph. The events logged by the entity represented by this node are shown as normal, and the other events are greyed out.

Figure 7.7: These figures represent the event list with its different components and some of its possible states and interactions. The list itself shows all events as they come in in the format that they come in to allow the user to see exactly what they have logged.

7.5 Architectural Reflections and Practical Constraints

While the core architecture and functionality of the proposed logging system align with the project's design goals, several practical considerations emerged during implementation. These factors, ranging from integration challenges within existing codebases to subtle inconsistencies in event timing, highlight the complexities of applying logging infrastructure in real-world environments. This section outlines key issues that influenced the implementation process and discusses their implications for both current and future applications of the system.

7.5.1 Integration Overhead and Retrofitting Challenges

Even with a simple interface for logging events, integrating it into an existing implementation can still require a non-negligible amount of effort. This includes managing the import, constructing the log message, and invoking the logger at appropriate points in the codebase. Accurately gathering the necessary data to construct valid log messages also introduced additional complexity. In particular, fields such as `vantagePointID` often had to be hard-coded to avoid intrusive modifications to the existing codebase. These limitations highlight the challenges of retrofitting logging into an existing implementation. In contrast, designing with logging in mind from the outset allows for cleaner integration and more consistent data capture.

7.5.2 Visualisation Value

A key strength of this proof of concept lies in its ability to generate meaningful visualisations that offer valuable insight into the system's behaviour. While the current implementation does not cover every possible use case, it already supports several useful scenarios. The frontend, developed from scratch using D3.js, provides a flexible web-based interface that enables dynamic and customisable visual representations. Although extending it with new visualisations may require some development effort, the modular structure makes it well-suited for further expansion and adaptation to specific needs.

7.5.3 Timing Problems

During the development of the system, an odd event was observed: on occasion, a `received` message would be logged before its corresponding `sent` message. Since each timestamp is recorded as close to the event's actual occurrence, this behaviour cannot be attributed to asynchronous logging or concurrent processing artefacts alone. Upon investigation, the anomaly was traced to the message being logged after the actual message was sent to the corresponding entity in one place, and it being logged before this in the other location. In our case, this was an easy fix, but it should be noted that problems like this can result in misleading visualisations, where events appear to occur in an impossible order. It should therefore be considered when interpreting logged timelines. However, this issue is typically limited to differences in the order of one millisecond or less, which rarely impacts the system's usability unless extremely fine-grained, in-depth analysis is performed.

Another timing-related problem that could occur when logging distributed devices as precisely as the system aims to do is related to internal clocks. Every device in the setup will have its own internal clock that gets updated periodically using an NTP server. It can occur that even when performing these periodic time checks, the clocks of different devices skew ever so slightly. When this happens, this can result in wrong conclusions if the analysis the user tries to execute requires these precise timings.

7.5.4 Scaling for Local Development

Several architectural choices in this system were made with an emphasis on performance and scalability in medium to large-scale deployments. For instance, incorporating a message broker facilitates horizontal scaling of the logging pipeline, and evaluating high-throughput, efficient

databases enables sustainable storage and querying of extensive qlog event datasets. While these components introduce overhead that may appear unnecessary for a lightweight, local deployment, their inclusion reflects a broader design philosophy: supporting a range of deployment scenarios, from isolated developer testing environments to distributed research platforms or production, scale monitoring systems. This multi-scenario approach inevitably leads to trade-offs. Optimisations for scalability, such as asynchronous message processing or log pre-aggregation, may introduce latency or complexity that is disproportionate in a single-user context. However, these design decisions were made deliberately to accommodate diverse use cases and to ensure the system remains robust and extensible as MoQ adoption grows and operational demands increase. The resulting architecture thus prioritises generality and future-proofing, even when this implies occasional inefficiencies in narrow contexts.

Chapter 8

Creating Value with MoQ Observability Infrastructure

For the successful adoption of the system, we require proof that shows its value and its place in the current development and the further adoption of MoQ. While MoQ builds upon QUIC's robust foundation, its layered session semantics and asynchronous control mechanisms introduce new complexities that are difficult to capture using conventional logging techniques. This system thus tries to provide structured insight into the inner workings of the protocol during a stream.

Observability in the context of MoQ refers to the ability to systematically capture, inspect, and analyse the internal state transitions, message flows, and behavioural dynamics of a MoQ session over time. Given the protocol's emphasis on real-time delivery, low latency, and publish-subscribe interaction models, as described in Chapter 3, observability must extend beyond simple logging. This, to provide structured, semantic insights into the behaviour of individual components and their interactions across the network.

Providing observability for MoQ does pose some inherent challenges. These start at the root of the protocol with QUIC. As QUIC encrypts most transport metadata, this means that normal network observability systems will not work. This, combined with the high-level asynchronous control flows that MoQ adds, makes the protocol especially difficult to envision. Another aspect that makes it difficult to observe MoQ is the many entities that are present in a setup. If we want to envision the whole streaming process, it is mandatory that we are able to capture and combine data from these multiple endpoints to get a complete overview. This also means that the tools and visualisations that we create must handle these multi-participant sessions.

As part of the solution for the challenges posed above, we introduced a structured logging scheme using an extension of qlog in Section 6.3.1. This scheme allows us to capture semantically meaningful events. By using a generalised logging format, we allow the use of external tooling and ensure the longevity of the system. This logging scheme, however, in and of itself, does not solve all the challenges. For this, we need a visualisation tool that accompanies it.

This chapter evaluates the practical value of the infrastructure by examining its role in real-world diagnostic scenarios. Through scenario-based analysis, we demonstrate how the system enhances transparency and troubleshooting capabilities during a MoQ session. These examples cover typical challenges in MoQ deployment and can range from debugging control message flows to relay forwarding failures. Each scenario contrasts the observability gap in a MoQ system without dedicated logging against the insight gained using our enriched logging and visualisation stack. The goal is to validate the system not merely as a proof-of-concept but as a necessary tool for MoQ development and operations, offering actionable feedback during protocol experimentation, interoperability testing, and deployment diagnostics. Beyond

scenario-driven diagnostics, the chapter also explores the system's contribution to learnability, how it facilitates understanding of MoQ's operational intricacies for both new and experienced developers. Finally, it examines the long-term value of this observability infrastructure in sustaining protocol evolution, supporting ecosystem tooling, and encouraging interoperability over time.

8.1 Scenario A: Preemptive Subscribe Confirmation

In this scenario, a subscriber initiates a `SUBSCRIBE` request to a relay in order to receive media content associated with a specific track. The request includes the `AbsoluteStart` filter type and a `StartGroup` value, instructing the system to begin delivery from the group identified by the given `StartGroup` identifier.

Problems arise when the publisher operates in a live-only configuration and does not retain previously published groups. In such cases, a `StartGroup` referencing an expired or pruned group will result in no data being available for delivery. This becomes particularly problematic when the relay, optimised for low-latency session setup, responds immediately to the subscriber with a `SUBSCRIBE_OK` without waiting for confirmation from the upstream publisher. Although the MoQ transport specification recommends that a relay "`SHOULD` delay its `SUBSCRIBE_OK` response until it receives a `SUBSCRIBE_OK` from the publisher" [CPN+25], this is not a strict requirement [Bra97]. As a result, some relays may acknowledge subscriptions speculatively.

This speculative behaviour can lead to silent failure if the upstream publisher does not respond, either because it lacks data for the requested `StartGroup` or because it treats the request as invalid and silently drops it. The subscriber, having received a `SUBSCRIBE_OK` from the relay, assumes the subscription is active and awaits media that will never arrive. Meanwhile, the relay, having committed to the acknowledgement prematurely, lacks a mechanism to notify the subscriber of the upstream failure.

Such failures are particularly challenging to detect and diagnose in traditional systems, as no explicit error is generated, and media delivery simply fails to materialise. The complexity increases further in multi-hop topologies or when multiple subscriptions are in flight. As shown in this evaluation, the proposed observability infrastructure enables the detection and analysis of this failure mode by correlating control message flows and their relative timing across entities.

Diagnosing this kind of failure without a dedicated observability system is extremely challenging. From the subscriber's perspective, a `SUBSCRIBE_OK` has been received, and the client assumes that media delivery should begin. The absence of subsequent media data may be misattributed to congestion, publisher inactivity, or other unrelated issues. On the relay side, unless verbose internal logging is manually enabled, there is often no indication that the upstream subscription remains pending. Comparing timings and relating the specific subscribe messages to each other over different entities can prove difficult as well. Furthermore, since QUIC encrypts control streams, passive inspection tools cannot provide any insight into the relay's interaction with the publisher.

In this context, a developer or operator is left with minimal actionable information. Identifying the root cause requires access to internal relay state and timestamped message flows from multiple entities, often reconstructed manually and with limited confidence in accuracy. In distributed deployments involving multiple relays or publishers, this diagnostic burden becomes unmanageable.

With the observability system in place, this failure scenario becomes immediately visible through the structured correlation of MoQ control events across vantage points. By inspecting the message sequence chart generated by the visualisation frontend, it becomes clear that the subscriber receives a `SUBSCRIBE_OK` before the relay has received upstream confirmation from the publisher. Figure 8.1 shows this specific message sequence chart that clearly demonstrates where and how

Figure 8.1: Subscribe conversation between the publisher, relay and subscriber. The subscriber sends its subscribe message to the relay, which responds with a confirmation. The relay passes the subscribe message to the publisher but never receives any confirmation.

it goes wrong. The subscriber starts the process by sending a subscribe message to the intermediate relay. The relay receives this and sends its own subscribe message to the correct publisher. Before receiving confirmation from the publisher, the relay sends a confirmation message to the subscriber. As mentioned, this is completely allowed according to the MoQ draft, but can cause confusion as a result. This sequence exposes the causal misalignment that leads to silent failure.

In contrast to the manual correlation required in traditional systems, the visualisation here enables near-instant diagnosis. In addition, by analysing the track identifiers of the events, a developer can isolate the affected subscription and confirm the absence of associated object events. This evidence enables targeted debugging and remediation, such as updating the relay logic to defer `SUBSCRIBE_OK` until backend confirmation is received, or at minimum, issuing a warning when upstream confirmation is delayed or dropped.

8.2 Scenario B: Undetected Connection Failures at Scale

In this scenario, a streaming platform rolls out a larger-scale test where one of the streams on their platform is implemented using MoQ. For this live stream, the publisher is expected to serve media to approximately 100 subscribers over a relay, where each subscriber is attempting to establish a MoQ session. While most subscribers successfully connect and receive content, a non-negligible number remain inactive, failing to receive any media data. From the publisher's perspective, no errors are raised, and network-level connectivity appears nominal. On the subscriber side, the connection attempt appears incomplete, and the application shows a blank screen.

Without a dedicated observability infrastructure, identifying the root cause is non-trivial. Traditional transport-layer tools such as Wireshark offer limited value in this context, particularly when working with encrypted QUIC connections. While it is technically possible to decrypt local QUIC traffic using session keys, this is not practical when dealing with large-scale or real-world deployments involving remote users. Moreover, Wireshark and similar tools lack protocol-specific awareness of MoQ control flows, making it difficult to pinpoint logical failures such as misrouted messages or invalid connection metadata.

By leveraging the observability system presented in this thesis, the issue becomes evident through two complementary visual tools. First, the connection graph, shown in Figure 8.2, reveals that certain subscriber nodes fail to establish valid links to the relay. These disconnected nodes are immediately visible as isolated or dangling entities in the graph layout, contrasting sharply with the majority of successful peer relationships.

From this visual cue, the developer can drill down into the event list for a specific failed connection. By filtering for control messages specifically coming from one of these subscribers, the root cause is revealed: the affected subscribers sent a connection request message containing an incorrect or unrecognised destination field, and the relay dropped the message.

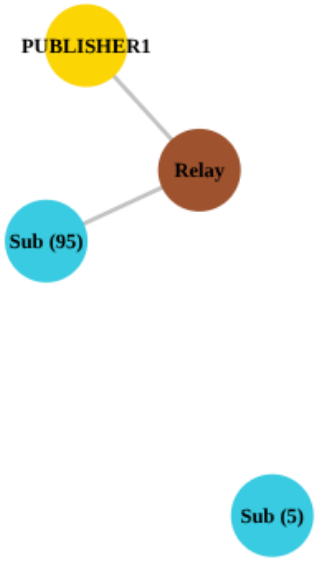


Figure 8.2: Larger scale connections graph with 95 subscribers correctly connecting to the relay and 5 subscribers being left out. The connection between the publisher, relay and 95 subscribers is correctly shown in the chart as a connected graph. The 5 dangling subscribers are left out as shown as a circle, not connected to any other nodes.

8.3 Scenario C: Latency Observation

In this scenario, a developer tries to gain insights into latency values across the MoQ protocol. In their testing, they observed that when placing a player on the sending side and one on the receiving end, side by side, there is a notable amount of delay when starting the live-stream.

Without an observability system, identifying the root cause of this latency would require extensive manual correlation of logs from multiple endpoints. This includes synchronising local clocks, parsing timestamps, and interpreting the order of control and data messages across distributed nodes. Given the asynchronous nature of MoQ control messages and the layered protocol structure, such analysis is error-prone and inefficient.

The observability system proposed in this thesis addresses this diagnostic challenge by visualising end-to-end latency metrics across protocol events. Using the latency measurement module, the developer identifies that subscribe messages consistently exhibit abnormal delays. As shown in Figure 8.3, the average end-to-end latency is measured at 786 milliseconds, with peak values exceeding two seconds, a substantial degradation for real-time media delivery.

This numeric feedback not only confirms the developer's initial suspicion but also narrows the focus to the session startup phase, particularly the propagation and processing of `SUBSCRIBE` and `SUBSCRIBE_OK` control messages. By highlighting these delays directly within the tool, the system removes the guesswork from multi-endpoint analysis and enables actionable insight into session dynamics.

Further inspection using the tool's event timeline view reveals that the delay stems from the interval between one specific `moq:subscribe-sent` and `moq:subscribe-received` interaction. With this information, we relate the problem to significant network-level delay, most likely due to congestion or buffering along the transmission path, which postpones the arrival of this message and consequently delays the initiation of media delivery.

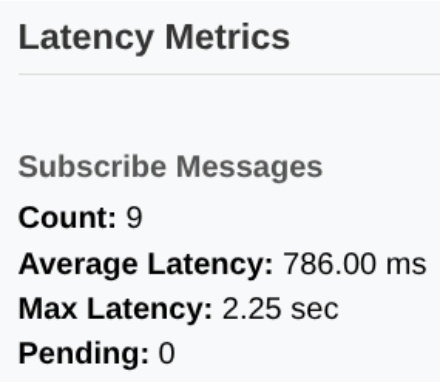


Figure 8.3: Example of a high value for the average latency for subscribe messages. This high value can be caused by the outlier, here over two seconds, and will require more in-depth analysis.

8.4 Improved Learnability of MoQ

Beyond its utility as a debugging and performance analysis tool, the proposed logging and visualisation system also contributes significantly to the learnability of the Media over QUIC protocol. By providing structured, contextualised insights into MoQ's operational flow, the system serves as an educational resource for developers, researchers, and protocol designers who are new to the protocol.

Traditionally, understanding the behaviour of emerging network protocols like MoQ requires synthesising knowledge from fragmented blog posts, early-stage specifications, and diverse codebases. This learning curve is further steepened by the protocol's complexity, particularly the publish-subscribe architecture, announcement propagation, and hierarchical stream and object structures.

The visualisations enabled by the proposed system, powered by enriched qlog data, allow users to observe in real time how connections are established, how control messages such as `ANNOUNCE` and `SUBSCRIBE` are exchanged, and how these influence session behaviour.

The connections graph, as shown in Figure 8.4, can allow future developers to further understand the topological layout of a MoQ session. This, combined with the message sequence chart in Figure 8.5, will aid in improving knowledge for the complete MoQ session setup.

Furthermore, the system can act as a pedagogical aid in academic or training contexts. Instructors can use recorded traces to explain protocol mechanics in a temporal and visual manner, thus reducing dependence on abstract textual descriptions. The standardised structure of qlog further ensures that traces are interoperable across tools and implementations, enhancing the accessibility of knowledge across the community.

Overall, the proposed observability infrastructure not only supports MoQ development and operations but also functions as a foundational resource for education and protocol adoption. By closing the gap between protocol specification and operational intuition, it facilitates a smoother entry point into the MoQ ecosystem.

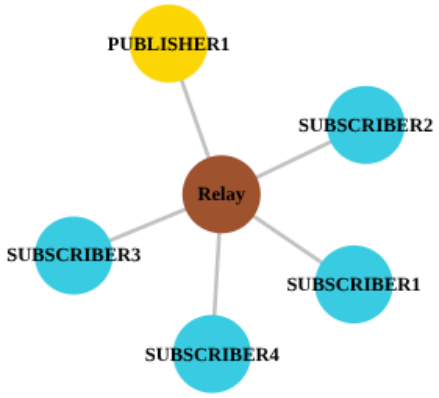


Figure 8.4: Connection graph comprising a single publisher and four subscribers connected to a relay.

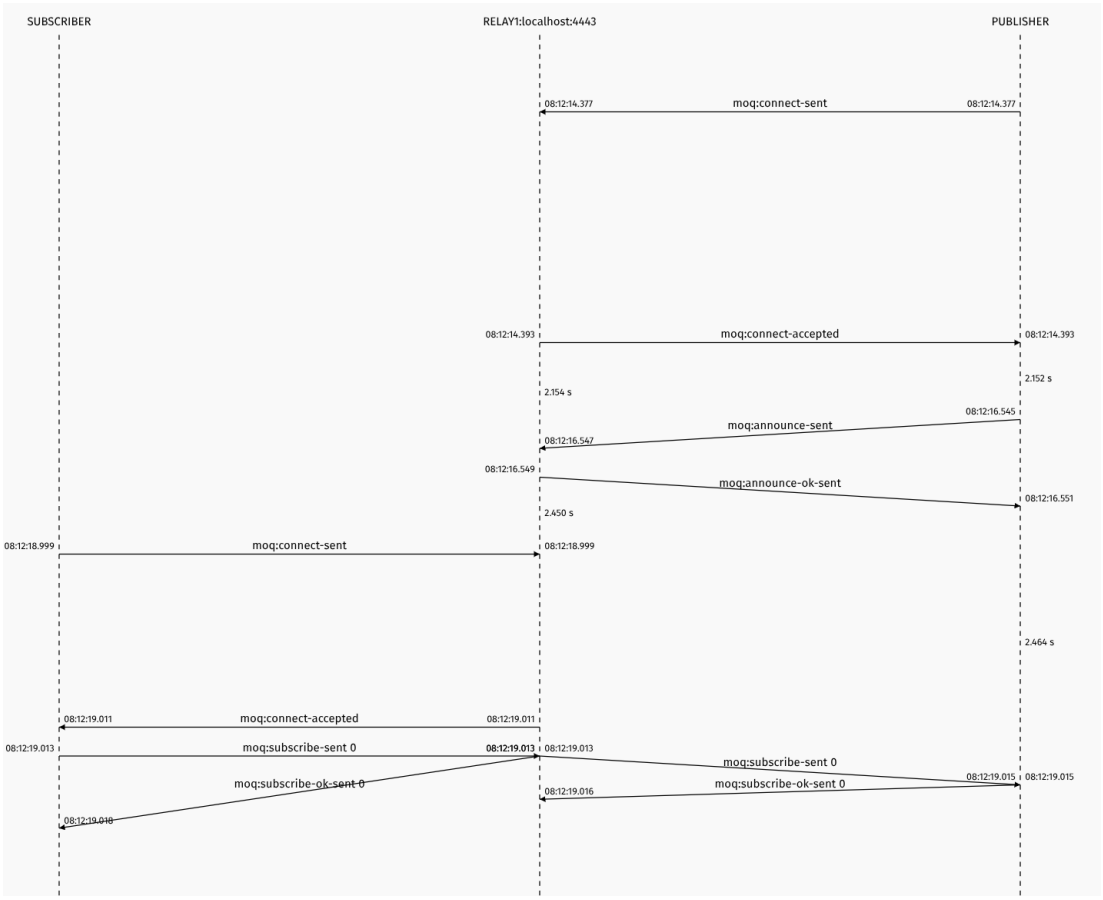


Figure 8.5: Example of message sequence chart which displays part of the control message conversation between the different actors. Messages are sent from one entity to another and displayed with direction, timestamps and event types. This example follows the start of the stream setup, where a publisher connects to a relay and announces a stream, followed by a subscriber connecting to a relay and subscribing to said stream.

8.5 Long-term Value and Future Proofing Through Standardisation and Archival

Beyond the immediate utility of visualisations in diagnosing protocol behaviour and improving session-level insights, the system proposed in this thesis also contributes to long-term value creation through its emphasis on data standardisation, extensibility, and archival capabilities.

At the core of this system lies the use of the qlog format, an extensible, structured logging schema specifically structured for modern transport protocols like QUIC and MoQ. The choice for qlog is not only a technical implementation detail, it is an important design decision that provides significant value for future development and evolution. By serialising events in a consistent and schema-compliant format, this system enables interoperability across tools and implementations. This adherence to a shared schema enables plug-and-play compatibility with both existing visualisation tools and emerging analysis frameworks.

Standardisation of log format also facilitates cross-tool visualisation. As new tools for protocol analysis emerge, the availability of logs in a widely adopted structure, such as qlog, ensures that the same dataset can be reused or reinterpreted. On the contrary, logs generated by other qlog-compliant tools can be ingested into this system, creating a modular ecosystem of tooling around MoQ.

The back-end of the system also supports long-term analysis through persistent archival of event traces. By maintaining a historical record of session behaviour of data, such as control message flows, developers can visualise protocol evolution over time. This is especially useful in a new protocol like MoQ, where iterations may rapidly reshape semantic behaviour. Through comparative analysis across archived datasets, the impact of such changes can be validated.

Moreover, archived logs create opportunities for downstream use cases such as regression detection, anomaly identification, and benchmarking. In future development cycles, historical qlog datasets can serve as reference traces against which the correctness or performance of new implementations may be tested. In operational deployments, they provide a forensic layer, enabling post-hoc analysis of failures, policy changes, or congestion events.

However, the long-term value described above is not without its limitations. The durability of archived traces and the interoperability benefits derived from standardisation depend heavily on the stability and backwards compatibility of the qlog schema and its MoQ-specific extensions. Should the qlog format undergo major revisions, whether in its core structure, serialisation conventions, or event semantics, existing datasets may become incompatible with future tools unless explicit migration or translation layers are introduced. Likewise, as MoQ itself is an evolving protocol, changes in its control message types or transport semantics may render older logs semantically ambiguous or partially obsolete. Without robust versioning and tooling support for schema evolution, these shifts risk fragmenting the ecosystem and diminishing the archival utility of early datasets. Consequently, while standardisation offers strong future-facing benefits, its practical effectiveness hinges on continued community coordination around qlog evolution and clear deprecation policies.

In summary, while the long-term utility of standardised logging and archival is contingent upon the continued stability and evolution of the qlog schema, the system as proposed nonetheless establishes a strong foundation for sustainable protocol engineering. By adhering to a shared format and preserving detailed session traces, it empowers both short-term debugging and long-term protocol analysis. Even in the face of potential schema evolution, the structured approach adopted here offers a viable path toward version-aware tooling, forward compatibility, and reproducible experimentation, which are essential qualities for maturing protocols like MoQ.

Chapter 9

Conclusion

This thesis set out to address the critical gap in the observability of Media over QUIC, a modern media transport protocol designed to deliver scalable, low-latency video delivery across diverse network environments. As MoQ introduces distinctive concepts such as publish-subscribe semantics, prioritisation logic and relay-based distribution, it also presents unique challenges for developers and researchers attempting to understand, evaluate and debug its behaviour. Given the protocol's ongoing standardisation and limited tooling support, there is a clear need for infrastructure that enables transparent introspection and analysis of MoQ behaviour in realist conditions.

In response, this work presented a modular, scalable observability system based on the extensible qlog format. The system captures protocol events in real time, structures them using a MoQ-specific qlog extension, and visualises them through an interactive frontend. By designing this toolset, the thesis aimed to enhance protocol transparency and support developers in debugging, evaluating and refining MoQ implementations.

From an architectural perspective, the system is scalable and adaptable. A decoupled logging pipeline based on message brokering, combined with lightweight serialisation and buffering techniques, enables the system to ingest and process high-throughput log streams under varied network and session conditions. Moreover, the design highlights key technical requirements for building effective observability tooling in this domain.

Through several scenarios, the system has proven to provide actionable insights that are otherwise difficult to obtain. Using detailed logging of control events, such as announcements and subscriptions, users can identify setup errors, detect inconsistencies and reason about performance characteristics. The integration of a dedicated visualisation layer further strengthens this capability, offering an intuitive view of session dynamics and asynchronous message flows.

Ultimately, this thesis contributes not only a functional observability system but also a framework that can support the broader evolution and standardisation of MoQ. Its design accommodates multiple scenarios, from debugging to analysis and protocol education, demonstrating both flexibility and long-term value. The challenges encountered during implementation have further informed future design considerations, which are explored in the next section.

9.1 Future Work

This thesis has laid the groundwork for structured observability in Media over QUIC by extending qlog to support MoQ-specific semantics and designing a scalable, web-based logging and visualisation system. While the results are promising, several avenues remain open for further exploration and development.

Expanding Event Coverage and Schema Maturity

The current implementation primarily focuses on logging control messages and some stream-based messages. Future work could extend qlog coverage to include more detailed transport-layer events (e.g., congestion interactions) and payload-level insights. Additionally, aligning the MoQ-specific qlog schema with evolving drafts and integrating feedback from broader community adoption would improve interoperability and long-term sustainability.

Real-world Deployment and Performance Evaluation

Future work should involve integrating the logging system into live MoQ environments to assess its effectiveness and operational impact. Such deployments would provide insight into usability, robustness under realistic workloads, and compatibility with different MoQ implementations. Alongside this, comprehensive performance benchmarking is essential to evaluate system overhead, log throughput limits, and responsiveness. These insights can inform optimisations in buffering, log transport, and storage strategies, ensuring the system scales effectively with high-bandwidth or bursty media traffic.

Automated Analysis and Anomaly Detection

With the recent surge in machine learning algorithms, the door is open to leverage them or to create rule-based analytics over the logged data, which could enable proactive detection of performance anomalies, session failures, or misconfigurations. This would transform the logging system from a passive diagnostic tool into a dynamic observability platform.

User-Centric Visualisation Enhancements

With production deployment in mind, more advanced, user-configurable visualisations tailored to different stakeholder roles (e.g., developers, operators, QA engineers) could improve usability. Features such as timeline compression, comparative session views, or drill-down analytics could be valuable additions.

Security, Privacy, and Data Governance

While basic considerations are discussed, further work is needed to evaluate the implications of logging sensitive metadata, especially in federated or privacy-sensitive deployments. Future iterations should implement fine-grained access control, redaction capabilities, and anonymisation policies in compliance with standards such as GDPR.

9.2 Reflection

Writing this thesis has been a challenging yet rewarding experience that allowed me to bring together technical knowledge, research skills and personal growth. While I had prior exposure to topics such as QUIC and structured logging through coursework and my bachelor's paper, applying them in the context of a new protocol like Media over QUIC exposed new complexities and forced me to rethink assumptions. I had to translate a constantly evolving standard into a working, analysable system, which required both technical and conceptual clarity.

Looking back, one of the main things I would approach differently is the degree of involvement with the broader MoQ community. Although I followed standardisation efforts, reviewed related drafts and asked for some input in open channels, I missed the opportunity to engage more actively with the working group. Earlier feedback from implementers or researchers familiar with current deployment challenges could have influenced both design decisions and evaluation priorities. I now realise that a more collaborative approach could have yielded insights that would have improved the system's relevance and usability in real-world contexts.

Another point of reflection concerns the architectural emphasis of the system. I spent a considerable amount of time ensuring the design was modular and scalable, with support for burst traffic and high-throughput scenarios. While these capabilities are important, especially when considering future large-scale experiments or deployment pipelines, I now feel that the system would have been more useful in the short term if it had focused more on enabling deep dives for developers. Providing better filtering, interactive drill-downs, or customisable trace views could have made the tool more immediately actionable for protocol implementers working through specific bugs or integration issues. These ideas emerged during the later phases of development, but by then, time constraints limited how far they could be explored.

This thesis has taught me to accept and work around feedback I received from others, whether in literature, specifications or conversations. I learned how to apply the knowledge that was presented to me while maintaining a critical perspective on areas that could still be improved. This process of finding a balance in openness to critique and having confidence in my own reasoning became a key lesson I will carry forward.

While the final system may not be exactly what I envisioned at the start, I'm proud of the result. It brings together a range of technical components and protocol knowledge in a way that is both functional and extensible. Most importantly, it reflects a learning process not just about MoQ or logging but about design trade-offs, collaboration and the reality of engineering under uncertainty.

Bibliography

[Alv21] Harald T. Alvestrand. Overview: Real-Time Protocols for Browser-Based Applications. RFC 8825, January 2021. (Last checked on 17/06/2025).

[BLA+23] Abdelhak Bentaleb, May Lim, Mehmet N. Akcay, Ali C. Begen, Sarra Hammoudi, and Roger Zimmermann. Toward one-second latency: Evolution of live media streaming. *arXiv preprint arXiv:2310.03256*, 2023. Survey under submission to IEEE Communications Surveys & Tutorials, Oct 5 2023 (Last checked on 17/06/2025).

[BLA+25] Abdelhak Bentaleb, May Lim, Mehmet N. Akcay, Ali C. Begen, Sarra Hammoudi, and Roger Zimmermann. Toward one-second latency: Evolution of live media streaming. *IEEE Communications Surveys Tutorials*, pages 1–1, 2025. (Last checked on 17/06/2025).

[Bra97] Scott O. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997. (Last checked on 17/06/2025).

[BVB19] Henk Birkholz, Christoph Vigano, and Carsten Bormann. Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures. RFC 8610, June 2019. (Last checked on 17/06/2025).

[Clo] Cloudflare. What is adaptive bitrate streaming? — cloudflare. `https://www.cloudflare.com/learning/video/what-is-adaptive-bitrate-streaming/`. (Accessed on 10/18/2024).

[CPN+25] Luke Curley, Kirill Pugin, Suhas Nandakumar, Victor Vasiliev, and Ian Swett. Media over QUIC Transport. Internet-Draft draft-ietf-moq-transport-08, Internet Engineering Task Force, February 2025. Work in Progress (Last checked on 17/06/2025).

[Cur25] Luke Curley. The MoQ Onion - Media over QUIC, May 2025. [Online; accessed 4. May 2025].

[GLFGG16] Boni Garcia, Luis Lopez-Fernandez, Francisco Gortazar, and Micael Gallego. Analysis of video quality and end-to-end latency in webrtc. In *2016 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, 2016. (Last checked on 17/06/2025).

[Goo25] Google. docs/native-code/rtp-hdrext/transport-wide-cc-02 - src - Git at Google, June 2025. [Online; accessed 8. Jun. 2025].

[Gro19] Boris Grozev. *Efficient and scalable video conferences with selective forwarding units and webRTC*. Theses, Université de Strasbourg, December 2019. (Last checked on 17/06/2025).

[IBM24] IBM. Stream control transmission protocol - ibm documentation. `https://www.ibm.com/docs/en/aix/7.3?topic=protocol-stream-control-transmission`, May 2024. (Accessed on 20/11/2024).

[Ibm25] Ibm. Message Brokers, April 2025. [Online; accessed 24. Apr. 2025].

[Int24a] International Telecommunication Union. Advanced video coding for generic audiovisual services. ITU-T Recommendation H.264 (V15) T-REC-H.264-202408-I!!PDF-E, International Telecommunication Union, August 2024. Approved on 2024-08-13. ITU-T Study Group 16. (Last checked on 17/06/2025).

[Int24b] International Telecommunication Union. High efficiency video coding. ITU-T Recommendation H.265 (V14) T-REC-H.265-202407-I!!PDF-E, International Telecommunication Union, July 2024. Approved on 2024-07-12. ITU-T Study Group 16.

[IT21] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021. (Last checked on 17/06/2025).

[LCV+24] Will Law, Luke Curley, Victor Vasiliev, Suhas Nandakumar, and Kirill Pugin. WARP Streaming Format. Internet-Draft draft-law-moq-warpstreamingformat-03, Internet Engineering Task Force, October 2024. Work in Progress (Last checked on 17/06/2025).

[MNSP25] Robin Marx, Luca Niccolini, Marten Seemann, and Lucas Pardue. qlog: Structured Logging for Network Protocols. Internet-Draft draft-ietf-quic-qlog-main-schema-11, Internet Engineering Task Force, March 2025. Work in Progress (Last checked on 17/06/2025).

[MRM10] Philip Matthews, Jonathan Rosenberg, and Rohan Mahy. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, April 2010. (Last checked on 17/06/2025).

[MRWM08] Philip Matthews, Jonathan Rosenberg, Dan Wing, and Rohan Mahy. Session Traversal Utilities for NAT (STUN). RFC 5389, October 2008. (Last checked on 17/06/2025).

[PE25] Lucas Pardue and Mathis Engelbart. MoQ qlog event definitions. Internet-Draft draft-pardue-moq-qlog-moq-events-01, Internet Engineering Task Force, March 2025. Work in Progress (Last checked on 17/06/2025).

[PM17] Roger Pantos and William May. HTTP Live Streaming. RFC 8216, August 2017. (Last checked on 17/06/2025).

[Ros10] Jonathan Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, April 2010. (Last checked on 17/06/2025).

[San23] Sandvine. Regional application popularity trends. Technical report, Sandvine Incorporated, January 2023. Accessed on 15/02/2025.

[SCFJ03] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July 2003. (Last checked on 17/06/2025).

[She22] Robert Sheldon. hierarchical storage management (HSM). *Search Storage*, February 2022. (Last checked on 17/06/2025).

[SSP15] Branislav Sredojev, Dragan Samardzija, and Dragan Posarac. Webrtc technology overview and signaling solution design and implementation. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1006–1009, 2015. (Last checked on 17/06/2025).

[Ste19] Daniel Stenberg. Tcp head of line blocking — http/3 explained. `https://http3-explained.haxx.se/en/why-quic/why-tcphol`, July 2019. (Accessed on 17/11/2024).

[Syl25] Lee Sylvester. WebRTC Issues and How to Debug Them, June 2025. [Online; accessed 9. Jun. 2025].

[TDM16] Martino Trevisan, Idilio Drago, and Marco Mellia. Impact of access speed on adaptive video streaming quality: A passive perspective. In *Proceedings of the 2016 Workshop on QoE-Based Analysis and Management of Data Communication Networks*, Internet-QoE '16, page 7–12, New York, NY, USA, 2016. Association for Computing Machinery. (Last checked on 17/06/2025).

[Tea25] I. R. Team. Debugging WebRTC: A Guide To Troubleshooting | IR, June 2025. [Online; accessed 9. Jun. 2025].

[Twi25] TwitchTracker. Twitch Viewers Statistics, May 2025. [Online; accessed 15. May 2025].

[Wie23] Fili Wiese. Http range request explained. `https://http.dev/range-request`, August 2023. (Accessed on 22/11/2024).

Appendices

Message Name	Description
CLIENT_SETUP	Initialises the session from the client side, including supported versions and parameters.
SERVER_SETUP	Server response to the client setup, selecting the version and setup parameters.
GOAWAY	Informs the peer that no further streams should be initiated, signalling session termination.
MAX_SUBSCRIBE_ID	Sets the upper bound for subscription identifiers allowed from the peer.
SUBSCRIBES_BLOCKED	Indicates the sender cannot accept new SUBSCRIBE messages temporarily.
SUBSCRIBE	Initiates a subscription to a specified track or group of media.
SUBSCRIBE_OK	Acknowledges that a SUBSCRIBE message has been accepted and is being processed.
SUBSCRIBE_ERROR	Indicates that a SUBSCRIBE request could not be fulfilled due to an error.
UNSUBSCRIBE	Terminates an existing subscription to a media stream.
SUBSCRIBE_UPDATE	Updates parameters of an ongoing subscription.
SUBSCRIBE_DONE	Publisher notifies that all objects for a subscription have been sent.
FETCH	Requests a specific range of previously published media objects.
FETCH_OK	Indicates that a FETCH request was successful and data is being delivered.
FETCH_ERROR	Indicates failure in processing the FETCH request.
FETCH_CANCEL	Cancels an ongoing FETCH request.
TRACK_STATUS_REQUEST	Asks for metadata or status information about a track.
TRACK_STATUS	Responds to a TRACK_STATUS_REQUEST with relevant metadata.
ANNOUNCE	Advertises available track namespaces to potential subscribers.
ANNOUNCE_OK	Confirms successful receipt and acceptance of an ANNOUNCE message.
ANNOUNCE_ERROR	Denies or reports an error processing an announcement.
UNANNOUNCE	Withdraws a previously advertised namespace.
ANNOUNCE_CANCEL	Signals that future subscriptions to a namespace should be disallowed.
SUBSCRIBE_ANNOUNCES	Subscribes to future announcements matching a given namespace prefix.
SUBSCRIBE_ANNOUNCES_OK	Acknowledges a successful SUBSCRIBE_ANNOUNCES message.
SUBSCRIBE_ANNOUNCES_ERROR	Indicates an error in processing a SUBSCRIBE_ANNOUNCES request.
UNSUBSCRIBE_ANNOUNCES	Terminates interest in further announcements for a namespace prefix.

Table 1: Overview of Media over QUIC Control Messages