POLITECNICO DI TORINO

Master's Degree in
Computer Engineering

Master's Thesis

Media Streaming Performance over HTTP/3

Supervisors
Prof. Antonio Servetti
supervisor's signature

. . . .

Candidate
Alessandro Bottisio
candidate's signature

. . . .

Academic Year 2024-2025

In the Name of God, the Most Compassionate, the Most Merciful.
All praise belongs to God, Lord of all worlds,
the Most Compassionate, the Most Merciful,
Master of the Day of Judgment.
You alone we worship and You alone we ask for help.
Guide us along the Straight Path,
the Path of those You have blessed,
not those You are displeased with, or those who are astray.

[ALLAH, Sura al Fatiha, Sacred Quran]

A Safa, ad Andrea, ai miei genitori,
alla nostra famiglia e ai miei amici

† A nonna Carmela

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Professor Antonio Servetti for his invaluable guidance, support, and patience throughout the course of my thesis work. His expertise, constructive feedback, and encouragement have been essential to both my academic growth and the successful completion of this project.

I would also like to thank all the faculty and staff at Politecnico di Torino who contributed to my education and personal development during my studies. Their competence and professionalism have helped and inspired me to pursue excellence.

A heartfelt thank you goes to my family, my parents and my brother Andrea for their unconditional love and constant support. Without their understanding, sacrifices, and encouragement, this journey would not have been possible.

I am sincerely grateful to my friends, who have shared this journey with me—both inside and outside the university. Their friendship, loyalty, camaraderie, advice, and sense of humor have always been a priceless gift to my life and made the challenging moments lighter and the successes even sweeter.

From the bottom of my heart, I wish to express my deepest appreciation to Safa, my fiancée, for her unwavering love, encouragement, support and patience throughout this journey. Her support has given me strength and motivation during the most challenging moments, and her presence in my life is always an endless source of joy, love, admiration, gratitude, inspiration and improvement. The light, happiness, peace and healing that she provided my soul ever since I received the invaluable gift of meeting her along with her love and companionship have been the greatest blessing that I have ever been graced with in my life. I am profoundly grateful to God to have shared this important chapter of my life with her, and I look forward to the endless more that we will, God willing, write together.

I would also like to thank all my grandparents for the love, support, inspiration and cherished memories that they have provided me over all these years, and to remember my grandmother Carmela, peace be upon her, with love and gratitude. May she receive a high station in the Hereafter and may her memory continue to bless us every day with fondness and warmth.

Last but most importantly, I would like to thank God for all my achievements, for all the good in my life and in my soul, and for the endless Love, Mercy, Guidance and blessings that He provides us in every moment of our existence.

Summary

This thesis provides a comprehensive analysis of Media over QUIC Transport (MoQT), a media transport protocol designed to enable efficient media delivery over the QUIC protocol. The study focuses on the architecture and performance evaluation of two proof-of-concept applications, moq-js and moq-rs, developed by Luke Curley. The primary objective of this research is to evaluate MoQT's effectiveness in media content delivery under varying network conditions, using a client-server-client architecture.

The architectural framework of MoQT is thoroughly examined, including its integration within the HTTP/3 stack, its object model and data hierarchy, streaming formats, and network topology. A detailed exploration of the setup, functionality, internal structure, and usage of moq-js and moq-rs is presented for the first time. Additionally, this work extends the applications to support QUIC's datagram transmission mode, facilitating a comparative analysis between communication schemes under ideal conditions, bandwidth constraints, and simulated packet loss.

Performance evaluation is based on an extensive set of measurements on both server and client sides to extract relevant key performance indicators (KPIs). This thesis contributes to a deeper understanding of QUIC's and MoQT's potential for media transport, providing insights and data that can inform future developments in this field.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis presents an in-depth analysis of Media over QUIC Transport (MoQT), focusing on the architecture and performance evaluation of some proof-of-concept applications developed by Luke Curley called *moq-js* and *moq-rs*.
The study aims to assess the capabilities of MoQT in delivering media content efficiently over the QUIC protocol, conducted in different network conditions with a client-server-client architecture.

We explore the architectural framework of MoQT, detailing it's role within the HTTP/3 stack, its object model and data hierarchy, streaming formats and adopted network topology.

Additionally, we alter the application code to implement a different transmission mode using *QUIC datagrams* on the client and server side, describing the created application protocol workflow, format and data structures in detail, along with the addition of logging functions and a client side logging server to export relevant data from *moq-js* for analytic purposes, which are then (if needed) converted and plotted using a variety of Python scripts.

The thesis includes a comparative analysis of the performance characteristics between QUIC stream and QUIC datagram modes, providing insights into their respective performance in handling media transmission under ideal, bandwidth restricted, and packet loss simulated network conditions and scenarios.

This work contributes to the broader understanding of QUIC's potential in media transport, offering insights that can guide future developments in this area.

Chapter 2

Background

2.1 HTTP

HTTP as the Foundation of Modern Web Communication

HTTP (*Hypertext Transfer Protocol*) stands as a **request-response protocol** that fundamentally shapes the nature of communication between web clients and servers. It defines the precise mechanisms through which web clients interact with web servers, orchestrating everything from simple document retrieval to complex application transactions. Indeed, HTTP is ***one of the foundation stones*** of how the modern internet operates, underpinning the exchange of information across a vast and ever-expanding network [1].

Today, **HTTP** powers the majority of web traffic globally, serving not only as the backbone for delivering ***web pages*** but also for a variety of application domains such as ***APIs***, ***real-time events***, and ***media*** streaming [30]. Operating at the application layer, HTTP has evolved into a versatile transport mechanism. For example, when users interact with dynamic content or initiate tasks such as file uploads or data retrieval, their applications typically issue background ***HTTP requests*** to communicate efficiently with backend servers.

Another increasingly significant application of **HTTP** is the ***streaming of data***. In the context of large files or continuous media, HTTP connections may be intentionally kept open for extended durations, enabling servers to ***send data in smaller chunks*** to the client as it becomes available. This technique is central to supporting both live and on-demand content consumption with minimal user-perceived delay.

Due to its **vital and widespread adoption**, HTTP has become the platform of choice for contemporary multimedia and live applications, reflecting the evolving needs and expectations of end users. As a result, one of the most **increasingly common use cases** is the efficient and adaptive **streaming of audio and video** content.

Modern ***HTTP-based streaming protocols***, such as **MPEG-DASH** and **HLS**, leverage persistent HTTP connections to deliver ***segmented media*** in manageable ***chunks*** [13]. These protocols are designed to maintain ***relatively low latency*** and support adaptive bitrate delivery, enabling high-quality, uninterrupted playback even under fluctuating network conditions. This architecture allows users to experience seamless media streaming tailored dynamically to their device capabilities and current network environment.

2.2 HTTP/2, TCP and UDP

2.2.1 HTTP/2

HTTP/2 [3] represents a major evolution of the Hypertext Transfer Protocol, specifically engineered to overcome the performance bottlenecks inherent in ***HTTP/1.1***. Its development was motivated by the need to substantially reduce latency and improve both resource utilization and the overall efficiency of web communications.

A core aspect of **HTTP/2** is its adoption of a **binary framing** layer. Unlike the textual, line-based approach of HTTP/1.1, all protocol messages in HTTP/2 are encoded in a compact binary format. This change enables more efficient parsing, reduced ambiguity, and opens the door for a richer feature set.

Another significant advancement introduced by HTTP/2 is **multiplexing**. With this feature, multiple independent messages (streams) can be sent and received simultaneously over a single TCP connection. As a result, web browsers and servers can avoid the inefficiencies and overhead associated with opening multiple concurrent TCP connections for different resources, leading to faster and more streamlined communication.

To further enhance protocol efficiency, HTTP/2 integrates **header compression** using the HPACK algorithm. This mechanism minimizes the overhead from redundant metadata in HTTP requests and responses, reducing bandwidth consumption and accelerating data transfer.

The protocol also implements **stream prioritization**, enabling clients and servers to assign relative importance to streams. This allows critical resources (such as the main HTML document) to be delivered with precedence over less time-sensitive data (such as images or background scripts), thereby improving perceived performance and user experience.

A particularly innovative feature is **server push**, which permits the server to proactively send resources to the client that it anticipates will be needed for rendering a webpage—such as stylesheets or scripts—without waiting for explicit requests. This anticipatory behavior can further decrease page load times.

From a security standpoint, HTTP/2 is typically deployed over **TLS**, providing enhanced confidentiality and integrity for all web traffic.

Despite these substantial improvements, it is important to note that **HTTP/2** continues to operate over ***TCP*** and thus inherits TCP's intrinsic limitations. The most notable among these is **head-of-line blocking**, wherein the loss of a single TCP packet results in the stalling of all concurrent streams until retransmission occurs. This characteristic, even in the presence of multiplexing, can prove detrimental for applications that require low-latency or real-time media transmission, as it undermines the seamless delivery of time-sensitive data.

2.3 TCP: Transmission Control Protocol

TCP (Transmission Control Protocol) is one of the **essential protocols** of the Internet protocol suite, first described in RFC 793 [26]. As a **connection-oriented protocol**, TCP establishes a persistent link between two endpoints and is specifically designed to ensure **reliable and ordered communication**. This reliability is a cornerstone of its design, making TCP integral to the vast majority of Internet applications.

One of the primary strengths of **TCP** is its comprehensive approach to **reliability**. By employing sequence numbers and acknowledgments (ACKs), TCP guarantees that all packets are received both accurately and in the intended order. If any packets are lost, TCP's built-in **retransmission** mechanism detects the loss and resends the missing data, ensuring that the receiving application is presented with a complete and correctly ordered byte stream.

To maintain **network stability and fairness**, TCP incorporates robust **congestion and flow control** mechanisms. Algorithms such as ***TCP Reno***, ***CUBIC***, and ***BBR*** dynamically adjust the ***transmission rate*** based on prevailing network conditions. This adaptability helps to avoid overwhelming the network and ensures that all users share available bandwidth equitably.

The process of establishing a TCP connection is initiated by the classic **three-way handshake** (SYN, SYN-ACK, ACK), which coordinates and synchronizes both endpoints before any data is transmitted. While this step is crucial for connection integrity, it does introduce some initial latency before the communication can commence.

Thanks to these properties, **TCP** is ideally suited for applications where ***delivery guarantees are more important than latency***, such as web browsing, file transfers, and email. However, the protocol's rigorous commitment to reliability comes at the ***cost of increased delay***, especially in high-latency or lossy environments—a limitation known as ***head-of-line (HoL) blocking***.

Head-of-Line (HoL) Blocking in TCP

A major limitation of TCP is the phenomenon of **head-of-line (HoL) blocking**. Due to TCP's strict requirement for ***in-order delivery***, when a single packet is lost or delayed, the receiving endpoint must ***wait*** for that specific packet to be ***retransmitted and received*** before it can process any subsequent packets, even if those packets have already arrived. This creates a bottleneck in which the ***entire stream is stalled*** by the absence of a single missing segment near the front of the receive buffer [16].

This behavior stems from TCP's design philosophy of **prioritizing *reliability and order preservation* over latency**. In practical terms, this means that, especially in lossy or high-latency networks, application performance can ***significantly degrade***. For media streaming applications, for instance, head-of-line blocking may cause noticeable delays, increased buffering, or jitter, even when only a small fraction of packets are lost.

The issue becomes even more pronounced in **multiplexed applications**—such as HTTP/2 over TCP—where multiple logical streams of data are interleaved within a single connection. In this scenario, a single lost packet can ***block the delivery of unrelated streams***, impeding the responsiveness of the entire application.

This design limitation was a **core motivator** for the development of **QUIC**, a modern transport protocol that ***avoids HoL blocking*** at the transport layer. QUIC achieves this by providing ***independent, reliable streams*** over a single connection, each of which can be delivered and processed out-of-order relative to others [14].

In summary, while **TCP** delivers strong guarantees of reliability and order, its susceptibility to head-of-line blocking renders it *suboptimal for latency-sensitive or real-time applications*—particularly in modern web environments where concurrency and rapid responsiveness are essential.

2.3.1 UDP: User Datagram Protocol

UDP (User Datagram Protocol) is defined in RFC 768 [25] and represents one of the core transport protocols of the Internet. In contrast to TCP, **UDP** is a **connectionless** protocol, offering a minimal service model that transmits discrete **datagrams** between endpoints without guaranteeing delivery, order, or data integrity. This architectural simplicity is central to UDP's appeal and underpins its distinct performance characteristics.

A primary attribute of **UDP** is its ability to provide **low latency** communication. Because UDP does not require the establishment of a connection handshake or the maintenance of state between sender and receiver, initial transmission delays are minimized. This makes UDP highly suitable for scenarios where rapid, lightweight message exchange is critical.

UDP is characterized by the complete absence of **built-in flow or congestion control**. Responsibility for managing the rate and volume of data transfer is delegated entirely to the application layer. While this design grants applications fine-grained control over their transmission behavior, it also introduces risks if flows are not appropriately regulated, potentially contributing to network congestion or unfair bandwidth usage.

Another defining property of UDP is its **unreliable delivery model**. The protocol makes no assurances regarding packet arrival: datagrams may be lost, duplicated, or received out of order. It is the prerogative of the application to implement any needed mechanisms for retransmission, error correction, or sequence reordering.

UDP is also notable for its **low protocol overhead**. The UDP header consists of only 8 bytes, significantly smaller than the minimum header size in TCP. This minimalism reduces the per-packet processing and bandwidth cost, further reinforcing UDP's suitability for delay-sensitive applications.

Collectively, these features make UDP the protocol of choice for many **real-time** and interactive use cases, such as **VoIP**, **live video/audio streaming**, **DNS queries**, and **online gaming**, where timely delivery is often more valuable than guaranteed reliability.

Importantly, UDP also serves as the **foundational transport layer** for modern protocols such as **QUIC**. By leveraging UDP datagrams as its substrate, QUIC is able to implement advanced features—such as reliable delivery, congestion control, and multiplexed streams—entirely in user space, thus **circumventing limitations imposed by TCP**, most notably the problem of **head-of-line blocking** [14]. This approach has enabled a new generation of low-latency, high-performance transport protocols tailored for web and real-time media applications.

2.3.2 TCP vs UDP: Comparative Summary

Feature	**TCP**	**UDP**
Connection model	Connection-oriented	Connectionless
Reliability	Guaranteed (ACKs, retransmissions)	Not guaranteed
Packet ordering	In-order delivery	No ordering guarantee
Congestion control	Yes	No (application-level)
Protocol overhead	Higher	Lower
Use cases	Web, email, file transfer	Streaming, gaming, DNS

2.4 Modern live transmission use cases

The demand for **low-latency media transport** has become central to the architecture of modern **streaming platforms** such as Twitch, YouTube Live, and widely-used video conferencing tools including Zoom and Google Meet. These platforms aim to deliver highly interactive experiences, where delays in media delivery can significantly degrade user engagement and responsiveness.

Traditionally, **HTTP-based streaming** solutions have been the backbone of large-scale video delivery; however, they often introduce end-to-end latency on the order of **5 to 15 seconds**. Such latency is unacceptable for interactive scenarios like live chats, virtual events, or online gaming, where near-instantaneous feedback is required. To overcome these limitations, many platforms have integrated protocols specifically designed for real-time communication, such as **WebRTC**, or have adopted optimized delivery schemes like **Low-Latency HLS (LL-HLS)** [22]. These solutions drastically reduce playback delay, bringing latency down to just a few seconds or even sub-second levels in optimal conditions.

Beyond these specialized protocols, the networking community is increasingly turning to **QUIC** as a next-generation transport layer for live media. QUIC's architectural features—such as native **multiplexing**, support for **zero round-trip time (0-RTT) connections**, and **flexible stream delivery**—make it particularly well suited for the demands of modern interactive media applications. The protocol enables multiple media streams to be delivered concurrently and independently, minimizing head-of-line blocking and reducing startup and switching times.

Furthermore, the need for **finer-grained control** over delivery properties—such as dynamically balancing latency against reliability—has driven the development of new standards and protocols. Technologies like **HTTP/3**, the Media over QUIC Transport (**MoQT**) protocol, and similar innovations have emerged in direct response to these requirements, aiming to empower both developers and platforms with advanced tools for adaptive, scalable, and high-performance live streaming.

2.5 HTTP/3

HTTP/3 represents the latest major evolution of the ***Hypertext Transfer Protocol (HTTP)***, introducing substantial improvements in *performance*, *reliability*, and *security* over previous versions [5]. Developed to address limitations inherent in HTTP implementations built upon TCP, HTTP/3 leverages a novel transport foundation—***QUIC***—to offer superior performance and enhanced robustness in modern network environments [4].

One critical advancement introduced by HTTP/3 is the **elimination of TCP head-of-line blocking**. Earlier HTTP versions, particularly HTTP/2, although introducing multiplexing, still suffered from this issue due to their reliance on TCP. In TCP-based implementations, the loss or delay of a single packet inevitably stalls all multiplexed streams within the same connection. By contrast, HTTP/3's integration with QUIC enables each data stream to operate independently. Consequently, if a packet is delayed or lost in one stream, other streams remain unaffected, significantly reducing latency and enhancing responsiveness, especially for real-time or latency-sensitive applications.

Moreover, HTTP/3 provides a substantially **faster handshake** procedure compared to its predecessors. Through tight integration with **TLS 1.3** and QUIC, HTTP/3 supports connection establishment using just one round-trip time (1-RTT), and in certain cases, even zero round-trip time (0-RTT). This streamlined handshake process notably reduces initial connection latency, offering users a perceptibly faster and smoother web browsing experience, particularly in scenarios with high-latency or unstable network connections.

Another pivotal advantage of HTTP/3 is its **enhanced mobility support**. The underlying QUIC protocol inherently supports seamless connection migration, allowing active sessions to persist uninterrupted as client devices transition between different networks or IP addresses—such as when moving from Wi-Fi to cellular networks. This capability ensures continuous service availability, significantly improving user experience in today's mobile-driven landscape.

Additionally, security in HTTP/3 is notably strengthened through **built-in encryption**. Unlike earlier protocol versions where encryption was optional, HTTP/3 mandates the use of encryption as an integral part of the handshake process. As a result, every HTTP/3 connection provides robust data confidentiality and integrity guarantees by default, significantly improving overall communication security.

Importantly, HTTP/3 achieves these advancements without changing the fundamental semantics of HTTP. Historically, ***HTTP/1.1*** operated over various transport and session layers, while ***HTTP/2*** standardized its deployment primarily atop ***TLS over TCP***. HTTP/3 continues to support these well-established semantics, ensuring compatibility with existing HTTP methods, headers, and client-server interactions, while fundamentally shifting its underlying transport to the more capable QUIC protocol [4]. Furthermore, the negotiation of HTTP versions occurs ***seamlessly***, requiring ***no modifications*** to existing website code, thereby simplifying adoption for developers and users alike [5].

Thus, HTTP/3 represents a significant step forward for web communication, addressing the performance bottlenecks and security concerns of previous protocols, while providing developers and users with a transparent, secure, and efficient protocol optimized for contemporary internet applications.

2.5.1 Comparison of HTTP revisions

Feature	**HTTP/1.1**	**HTTP/2**	**HTTP/3**
Transport Protocol	TCP	TCP	QUIC (over UDP)
Multiplexing	No	Yes (within TCP)	Yes (native, stream-level)
Head-of-Line Blocking	Yes	Yes (TCP-level)	No (stream-level independence)
Binary Framing	No (text-based)	Yes	Yes
Header Compression	No	Yes (HPACK)	Yes (QPACK)
Stream Prioritization	No	Yes	Yes
Server Push	No	Yes	Yes
Encryption	Optional (TLS/SSL)	Recommended (TLS)	Mandatory (TLS 1.3 in QUIC)
Connection Migration	No	No	Yes (via QUIC)
Handshake Latency	High (TCP handshake)	High (TCP + TLS)	Low (1-RTT/0-RTT via QUIC)

Table 2.1. Comparison of HTTP/1.1, HTTP/2, and HTTP/3

2.6 QUIC

QUIC is a secure, general-purpose transport protocol originally developed by Google and later standardized by the IETF as RFC 9000 [14]. Designed to overcome many of the inherent limitations of TCP, QUIC introduces a suite of innovations that address modern web and media application requirements.

A defining characteristic of **QUIC** is its support for **stream multiplexing**. Unlike TCP, which only allows a single ordered byte stream per connection, QUIC enables multiple, concurrent, and fully independent streams within the same connection. Each stream is flow-controlled separately, and the loss or delay of packets in one stream does not impede the progress of others. This approach effectively eliminates the problem of *head-of-line (HoL) blocking* that affects protocols built on TCP, resulting in more resilient and responsive communication—especially crucial for real-time and interactive applications.

Another significant innovation is that QUIC is implemented predominantly in **user space**, rather than as part of the operating system kernel. This architectural choice allows for faster protocol iteration, easier updates, and broad platform independence. It also means that new features and bug fixes can be deployed at application scale without requiring OS-level changes or upgrades.

Connection migration is another notable feature of QUIC. It allows ongoing connections to survive changes in network IP address—such as when a mobile device switches from Wi-Fi to cellular data—without interruption. This capability is essential for supporting seamless user experiences in mobile and unstable network environments.

For applications that demand unreliable or partially reliable data delivery, QUIC supports **datagrams** as defined in RFC 9221 [24]. This feature enables the transmission of data packets that do not require retransmission, making QUIC especially attractive for real-time media use cases where latency is more important than perfect reliability.

Security is tightly integrated into the design of QUIC. It leverages **TLS 1.3** for establishing secure channels, providing **forward secrecy**, **confidentiality**, and **authentication** by default. All payloads are encrypted and authenticated, protecting both control and user data against interception and tampering.

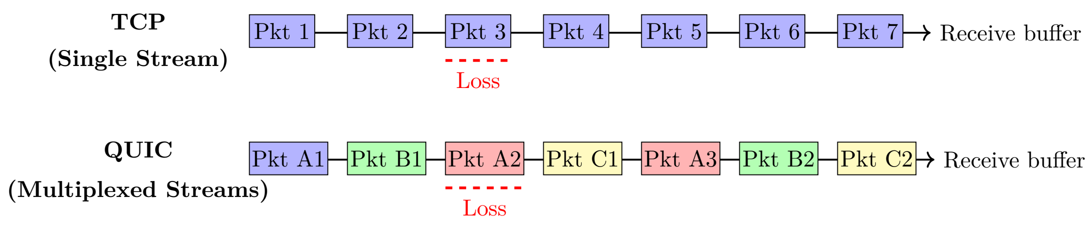
From a deployment perspective, **QUIC packets are encapsulated within UDP datagrams**, facilitating compatibility with existing network infrastructure and firewalls. The user-space nature of QUIC also enhances its deployment flexibility and adaptability across a wide variety of platforms.

QUIC's ***flexibility*** and robust feature set make it a compelling candidate to **replace TCP** in scenarios demanding **low-latency connections**, rapid session establishment, and dynamic network path migration. Applications benefit from ***flow-controlled streams***, network agility, and comprehensive ***confidentiality***, ***integrity***, and ***availability*** guarantees across diverse content distribution and media delivery scenarios.

QUIC's Approach to Eliminating Head-of-Line Blocking

A central design goal of **QUIC** is the effective elimination of head-of-line (HoL) blocking, a persistent and well-known limitation in TCP-based protocols. In traditional TCP, all data flows through a single, strictly ordered byte stream per connection. When packet loss occurs, TCP's commitment to in-order delivery forces the receiving endpoint to withhold any subsequently received data until the missing packet is retransmitted and received. As a result, the progress of all application data—regardless of their logical independence—is stalled by the loss of any single packet. This phenomenon, known as **head-of-line blocking**, can significantly increase latency, reduce throughput, and impair user experience, especially in scenarios involving multiple, interleaved data flows such as multiplexed web requests or real-time streaming.

QUIC overcomes this limitation by introducing a fundamentally different transport architecture based on **independent, multiplexed streams**. Within a single QUIC connection, applications can open and use multiple concurrent streams, each governed by its own **flow control** and delivery order. This means that if a packet belonging to one stream is lost or delayed, only that stream is affected—**all other streams continue to transmit and receive data without interruption**. The protocol's stream independence effectively isolates the impact of network loss to only the affected logical channel, ***eliminating the bottleneck*** that arises in TCP when loss or reordering occurs [14].



Note:
In TCP, all subsequent packets are blocked until the lost packet is retransmitted.
In QUIC, only stream A is affected.

Figure 2.1. Illustration of Head-of-Line Blocking in TCP vs. QUIC

This stream-based architecture is particularly advantageous for web browsers and media applications. For example, when loading a complex webpage, a browser might initiate parallel streams for HTML, CSS, images, JavaScript, and video segments. In TCP, a lost packet in a video segment could delay the rendering of unrelated images or scripts, resulting in visible lag. With QUIC, each resource is delivered on a separate stream, so network issues affecting one resource do not impede others, yielding smoother and more responsive page loads.

The benefits are even more pronounced in the context of **real-time media delivery**. Interactive applications such as live video streaming, online gaming, or video conferencing rely on continuous, timely delivery of multiple streams (e.g., audio, video, metadata). QUIC's architecture ensures that transient losses or congestion affecting, for instance, a video stream, do not delay the delivery of audio—maintaining quality of experience and minimizing the risk of synchronization issues or perceptible stalls.

Another important factor is that **QUIC is implemented in user space** and runs over **UDP**, allowing protocol evolution and bug fixes without operating system kernel updates. This not only enables rapid innovation in how streams are managed and prioritized, but also facilitates platform and application-specific optimizations. The decoupling from the operating system kernel makes it easier to deploy enhancements across diverse environments and at Internet scale.

It is precisely these advances that motivated the adoption of QUIC as the foundation for **HTTP/3**. By building on QUIC's multiplexed, loss-resilient architecture, HTTP/3 is able to provide faster, more reliable, and more adaptive web communications, even over unreliable or variable network paths. The elimination of head-of-line blocking is thus not merely a technical detail, but a transformative capability that underpins the next generation of high-performance, low-latency network applications.

Comparison of Transport Protocol Features

Feature	**TCP**	**UDP**	**QUIC**
Connection-oriented	Yes	No	Yes
Reliability	Yes	No	Yes
Ordering	Yes (global)	No	Per-stream
Congestion Control	Yes	No	Yes
Flow Control	Yes	No	Yes (per stream)
Multiplexed Streams	No	No	Yes
Head-of-Line Blocking	Yes	N/A	No
User-space Implementation	No (typically kernel)	Yes	Yes
Encryption	Optional (TLS/SSL)	Optional	Mandatory (TLS 1.3)
Connection Migration	No	No	Yes
0-RTT Handshake	No	N/A	Yes
Datagram Support	No	Yes	Yes (RFC 9221)
Standardization	IETF RFC 793 (1981)	IETF RFC 768 (1980)	IETF RFC 9000 (2021)

Table 2.2. Comparison of Transport Protocols: TCP, UDP, and QUIC

2.7 MoQT: Media over QUIC Transport

The **Media over QUIC** (**MoQ**) **Working Group** is a specialized committee established by the **Internet Engineering Task Force** (**IETF**) with the mandate to develop a unified, low-latency, scalable, and reliable transport protocol for both media ingestion and distribution. Officially chartered in **July 2022**, the group was tasked with designing a protocol that could fully exploit the advanced capabilities of **QUIC** to efficiently deliver real-time and on-demand media across a range of network conditions and deployment scenarios [11].

The impetus for MoQ stemmed from years of fragmentation in the real-time media delivery landscape, where protocols such as **WebRTC**, **RTMP**, and **DASH** addressed only subsets of the necessary requirements for latency, reliability, and transport control. The emergence of ***QUIC*** as a next-generation transport protocol—with features including multiplexed streams, rapid connection migration, and reduced handshake latency—provided the ***ideal foundation*** for a new media transport architecture capable of meeting the modern demands of live and interactive applications [14].

Within this context, **MoQT** (Media over QUIC Transport) has been conceived as a **standardized**, **end-to-end transport layer protocol** for media publishing and subscribing. Unlike previous efforts that focused on inventing new media formats, MoQT targets the delivery of media chunks—such as video frames, audio packets, and metadata—across the network with **minimal latency** and **maximum flexibility**, while supporting both **ingest** (uploading from publishers to relays) and **distribution** (downloading to viewers or edge nodes) [29].

The development of MoQT has catalyzed significant industry participation, drawing contributions from a **broad spectrum of companies and organizations**:

- **Meta (Facebook)** has played a foundational role, contributing to the protocol's core drafts and presenting experimental architectures and tools that leverage QUIC for scalable media transport [20].
- **Google**, leveraging its expertise in both QUIC and WebTransport, has been instrumental in shaping browser integration and has developed prototype implementations in Chromium [7].
- **Akamai** and **Cloudflare**, as global CDN leaders, have offered critical insights into large-scale content delivery and contributed relay protocol optimizations.
- **Daily** has explored real-time communication workflows, informing MoQT's integration with existing WebRTC-based solutions.
- **MPEG** and **BBC R&D** have engaged with the group through early research, standardization efforts, and interoperability tests.

MoQ's design is heavily influenced by experimental protocols such as **WARP** (WebRTC Asynchronous Reliable Protocol) and **RUSH** (Relayed Upload Streaming Handler). **WARP** demonstrates low-latency, chunk-based distribution for real-time media, while **RUSH** emphasizes event ordering and partial reliability, optimizing ingest workflows for live production environments [18, 27].

MoQT is engineered to operate seamlessly over both **WebTransport**—enabling browser applications to access QUIC's performance and reliability—and **raw QUIC** for native applications that require even tighter control over media delivery [7]. This dual compatibility ensures broad applicability across web, mobile, and native media workflows.

The development timeline of MoQT reflects its rapid evolution and growing maturity:

- **2022**: Working Group formation, initial charter, and scoping of core problems.
- **2023**: Publication of early drafts detailing the transport model, object hierarchy, and interoperability goals; initial interop discussions among browser vendors and media platforms.
- **2024**: Refinement of technical drafts, community-driven interop testing at IETF hackathons, and the emergence of multiple reference implementations (such as `moq-js`, `moq-rs`, and `ffmpeg-moq`).
- **2025 (ongoing)**: Work towards working group last call (WGLC), expanded real-world deployment, and integration testing with live production and streaming environments.

As of mid-2025, MoQT continues to evolve through active collaboration between browser vendors, streaming platforms, CDN operators, and media tool developers. Its long-term vision is to serve as a **unifying transport layer** for low-latency, scalable media ingestion and distribution—complementing established protocols like **DASH** and **HLS**, while meeting the heightened expectations of modern interactive and live media applications.

Chapter 3

Media Over Quic Transport

3.1 Design Philosophy of MoQT

The design philosophy of Media over QUIC Transport (MoQT) is rooted in the ambition to create a modern, future-proof framework for the delivery of interactive, real-time, and on-demand media over the internet. Drawing on lessons learned from previous generations of media protocols, MoQT seeks to balance flexibility, scalability, and low-latency performance, while accommodating the rapidly evolving requirements of next-generation media applications [28, 29].

Transport-Media Decoupling

A central tenet of MoQT's philosophy is the explicit separation between transport mechanisms and media semantics. While traditional systems such as RTP tightly couple media negotiation, timing, and delivery, MoQT deliberately avoids embedding application-level logic or codec-specific intelligence into the transport layer. Instead, it provides a generic, extensible substrate for efficiently conveying discrete media objects, allowing application developers to innovate independently at the media layer [28].

Object-Based Media Delivery

MoQT is fundamentally object-centric: all media data is packaged as independently addressable objects (e.g., video frames, audio packets, metadata). This granularity supports not only advanced routing, caching, and relaying strategies, but also enables rapid error recovery, efficient retransmission, and fine-grained prioritization. Object-based design reflects a shift toward content-centric networking and aligns with emerging industry paradigms for adaptive and interactive media [29].

Scalability and Hierarchical Organization

The protocol's hierarchical model—spanning namespaces, groups, tracks, and objects—is designed to enable deployments at internet scale. By isolating administrative domains (namespaces), aggregating related streams (groups), and providing independent logical flows (tracks), MoQT facilitates efficient resource allocation, access control, and traffic engineering across diverse network infrastructures.

Extensibility and Evolution

Recognizing the diversity of future media formats and use cases, MoQT is engineered for extensibility. The protocol incorporates reserved fields, flexible headers, and version negotiation mechanisms that allow new features—such as enhanced metadata, novel reliability schemes, or future security models—to be incorporated without disrupting backward compatibility or existing deployments [28, 29].

Low Latency and Application Diversity

MoQT is particularly attuned to the needs of ultra-low-latency and interactive applications, such as live streaming, real-time communications, and cloud gaming. By leveraging QUIC's inherent support for both reliable (streams) and partially reliable (datagrams) delivery, MoQT empowers applications to make per-object trade-offs between reliability and latency, optimizing for diverse media types and user experiences.

Interoperability and Simplicity

Simplicity and broad interoperability are guiding values in the design of MoQT. The protocol intentionally exposes minimal, generic interfaces to media applications and relays, reducing implementation complexity and fostering rapid adoption. Its metadata-centric approach enables third-party caches, CDNs, and edge services to participate in media delivery ecosystems without deep protocol or codec awareness.

Security and Privacy Considerations

Finally, MoQT inherits and extends the security properties of QUIC, ensuring confidentiality, integrity, and authentication for media flows. The protocol's architecture supports secure multi-party topologies, end-to-end encryption, and fine-grained access controls—all essential features for contemporary media distribution.

In summary, the design philosophy of MoQT is distinguished by its modular, object-based abstraction; its commitment to scalability and extensibility; and its alignment with the performance and interoperability needs of tomorrow's media applications. Through this architecture, MoQT aims to lay a robust foundation for flexible, efficient, and secure media transport over the evolving internet [28, 29].

3.2 General architecture

At the core of MoQT's architecture is a flexible and efficient **publisher/subscriber model**, which enables low-latency delivery of media from producers (e.g., live encoders, cameras, or media servers) to consumers (e.g., players, analytics systems, or relays). This model is built around the concept of hierarchical **tracks**, **groups**, and **media objects**, which together define how media is structured, announced, and delivered over the network [1].

MoQT defines three main roles in its pubisher/subscriber model:

- **Publishers**: Entities (usually servers) that create streams of media (e.g., a webcam stream or a live broadcast feed) and publish content to be consumed.
- **Subscribers**: Endpoints (clients) that select specific media content to consume via subscription, thus receiving data in real-time or near-real-time.
- **Relays**: Entities that optionally forward these streams, caching and routing multimedia content according to subscriptions and enabling network scalability and multi-hop delivery.

An **integral** characteristic of the ***effectiveness*** of a transmission protocol like MoQT is its ability to **distribute content at scale**. In particular, MoQT is designed to **allow the use of relays** to **effectively leverage** third-party networks, independent of the entity that publishes or subscribes to/from the content.

Relays can thus be used to form a **content distribution network**, conceptually similar to **Content Delivery Networks (CDNs)**. Relays treat MoQT objects as *opaque entities*, without modifying, combining, or splitting their payloads, and should prioritize them in accordance with the protocol specifications.[2]

Consumers have the flexibility to **adapt their behavior** based on current **network conditions**. For example, they may delay playback slightly to buffer more data and improve stream quality in poor conditions, or prioritize low-latency delivery at the cost of occasional loss or lower resolution.

From a ***topological perspective***, and based on MOQT's pubisher/subscriber model, the **simplest architecture** that can used by deploying **all possible entity role types** consists of a publisher that advertises content to be distributed to a relay, which in turn advertises and distributes it to one or more receiving final subscribers.
[3]

[1]MoQ Transport Draft, https://datatracker.ietf.org/doc/draft-ietf-moq-transport/

[2]https://datatracker.ietf.org/doc/draft-ietf-moq-transport/

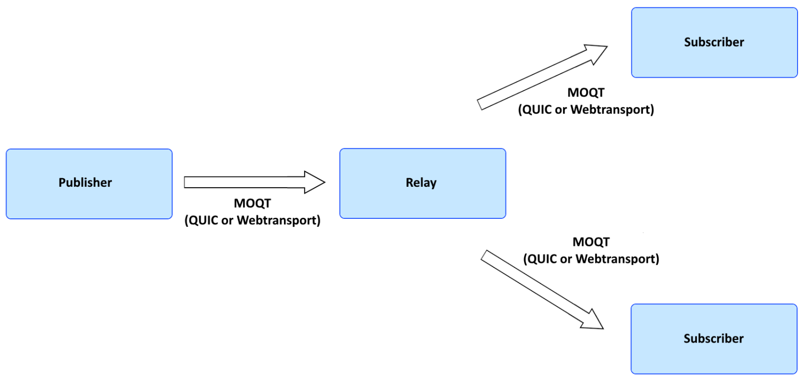


Figure 3.1. Simple MoQT architecture

More specifically, instances of the **original publisher** and **final subscriber** run as **clients in a browser environment**, which accesses a **local server** running the moq-js application, while the **relay** consists of an **instance of the moq-rs application** running locally or on a remote server to simulate varying network conditions.

MoQT streams are structured with a specific **hierarchical model** for their data objects using **Track**, **Group**, and **Object** identifiers. These enable flexible organization of media content (e.g., separating audio and video, different camera angles, quality layers, or subtitles) [4].

More specifically:

- **Object**: The **basic data unit** in MOQT; it is an addressable unit whose payload is a sequence of bytes. All objects belong to a group, which specifies their ordering and potential dependencies.
- **Group**: A **collection of objects** and a subunit of a track, acting as a join point for subscriptions. Objects within a group sholud not depend on objects from other groups.
- **Track**: A **sequence of groups**, the entity for which a consumer issues a subscription request.

An **object** is uniquely identified by its *track namespace*, *track name*, *group ID*, and *object ID*, and it must be an identical sequence of bytes regardless of how or where it is retrieved.

[3] https://www.meetecho.com/blog/moq-webrtc/

[4] IETF MoQ Transport Draft: https://datatracker.ietf.org/doc/draft-ietf-moq-transport/

Objects consist of two parts: **metadata** and **payload**. Metadata is never encrypted and is **always visible** to relays. The payload part can be **encrypted**, in which case it is only visible to the original publisher and the final subscribers. The application is **solely responsible** for the content of the object's payload. This includes the underlying encoding, compression, any end-to-end encryption, or authentication. **A relay cannot combine, split, or otherwise modify the payloads of objects**.

Objects within a group **should not depend** on objects in other groups. A **group** serves as a join point for subscriptions. For example, a new subscriber may not want to receive the entire track but may instead choose to receive only the latest groups. The publisher **selectively transmits** objects based on their **membership** in the group.

A **track** is a sequence of groups. It is the entity for which a subscriber issues a **subscription request**. A subscriber can request to receive individual tracks **starting from a group boundary**, including any new objects sent by the publisher while the track is active.

The format design is strongly motivated by the requirements of modern real-time and interactive applications. MoQ supports both fully reliable (QUIC streams) and partially reliable (QUIC datagrams) delivery, allowing applications to finely balance reliability and latency according to use case. Object-level fragmentation and reassembly, explicit signaling of dependencies, and native support for prioritization and selective subscription all contribute to minimizing head-of-line blocking and maximizing timely media delivery.

3.3 MoQ Streaming Formats

A central design feature of the **Media over QUIC Transport (MoQT)** protocol is that it defines a structure for transporting media (like video or audio) over QUIC using named media objects rather than traditional streams.

It aims to simplify caching, multiplexing, and real-time delivery across relay-based topologies. It's designed to be **codec-agnostic**, **scalable**, and **adaptable** to modern streaming needs like live events, interactive applications, and real-time communications.

To that end, MoQT supports multiple **streaming formats**[5], each defining how media data is **structured**, **named**, **prioritized**, and **delivered** over the network, along with **policies for discovery and subscription**.

[5]https://datatracker.ietf.org/doc/html/draft-ietf-moq-transport-07#name-introduction

These streaming formats are designed to be interoperable with the MoQ transport layer without mandating specific codecs or container formats.

3.4 Overview of Streaming Formats in MoQ

Unlike traditional protocols that tightly couple transport with media semantics (e.g., RTP/SDP in WebRTC), MoQ separates the transport mechanism from media representation. One of the foundational design choices in MoQ is the strict decoupling of media semantics from transport mechanics. Unlike traditional protocols such as RTP/SDP, which entangle media negotiation and transport, MoQ explicitly separates the representation of media objects from the mechanisms used to deliver them [28].

This separation enables relays and intermediaries to efficiently route, cache, or prioritize data based solely on standardized metadata, without requiring knowledge of specific media codecs or content types and allows for greater flexibility, interoperability, and scalability, particularly when distributing real-time or low-latency media over the internet.

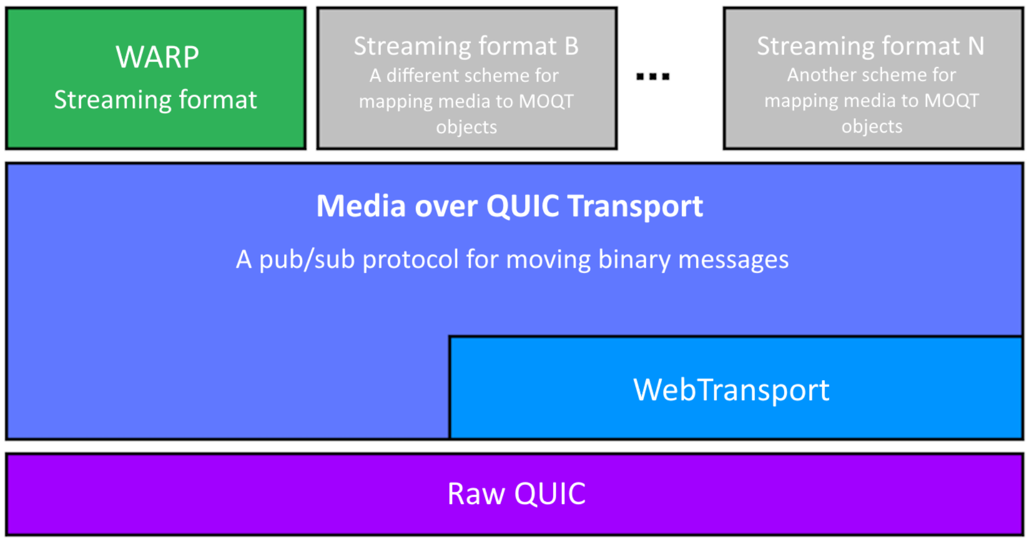

WARP
Streaming format
Streaming format B
A different scheme for mapping media to MOQT objects
...
Streaming format N
Another scheme for mapping media to MOQT objects
Media over QUIC Transport
A pub/sub protocol for moving binary messages
WebTransport
Raw QUIC

Figure 3.2. MoQT hierarchy

3.4.1 Purpose of Streaming Formats

The design of streaming formats in MoQ fulfills several pivotal roles within the protocol ecosystem.

Separation of Concerns: MoQ streaming formats are built to be media-agnostic and forward-compatible. The protocol standardizes metadata framing and signaling, but deliberately avoids encoding media-specific semantics into the transport itself. Reserved fields, extensible headers, and versioning schemes permit seamless evolution of the protocol, accommodating new media types, enhanced security, and future application requirements without disrupting existing deployments [29].

Define Object Hierarchies: streaming formats define a hierarchical organization of media, segmenting content into *namespaces*, *groups*, *tracks*, and *objects*. This multi-layered structure enables efficient relaying, caching, and prioritization without requiring interpretation of the media content itself and facilitates efficient routing, granular caching, and selective forwarding of media segments across distributed relay infrastructures [29].

Enable Interoperability: by standardizing metadata structures, MoQ ensures that disparate entities—whether they be relays, caches, or end-user players—can interpret, route, and manipulate media flows based on common logic. This approach fosters broad interoperability and removes the need for every intermediary to implement bespoke media-specific processing.

Support Customization: MoQ's approach to streaming formats is intentionally extensible and application-agnostic. This enables diverse application domains—ranging from interactive video conferencing to large-scale live event streaming and even voice-over-IP—to define domain-specific semantics atop the shared MoQ transport substrate, without sacrificing interoperability or performance [28, 29].

3.4.2 Key Streaming Formats

As the Media over QUIC Transport (**MoQT**) protocol matures, a central focus of standardization efforts has been the development of interoperable streaming formats that can leverage the unique capabilities of QUIC-based media delivery. As of 2025, two prominent streaming formats have emerged as reference models for the MoQT ecosystem: **WARP** and **RUSH**.

WARP (WebRTC Asynchronous Reliable Protocol) is designed specifically for the **downstream delivery** of media content from relays or publishers to multiple consumers. Its core innovation lies in adopting an **object-based transmission model**, in which discrete media objects—such as video frames, audio samples, or timed metadata—are delivered with low latency and can be individually prioritized or cached. WARP's architecture is optimized for **scalable, loosely coupled distribution**: relays can forward or cache objects without maintaining tight stateful coordination with all endpoints. This flexibility enables efficient media fan-out for live events, virtual conferences, and real-time collaborative environments [18].

Complementing WARP's downstream focus, **RUSH** (Relayed Upload Streaming Handler) addresses the **upstream ingestion** of live media. RUSH is tailored for workflows in which publishers—such as live event broadcasters or user devices—transmit media objects to relays, which may then redistribute or process the streams further. RUSH emphasizes **timing fidelity** (ensuring that object timing is preserved across hops) and supports **partial reliability**, allowing senders to trade off retransmissions against low latency. This is especially critical for real-time production and interactive streaming scenarios, where some packet loss is acceptable but consistent timing is essential [27].

The standardization of formats like WARP and RUSH enables **interoperability at the transport level** between different media systems and vendors. By providing clear expectations for object structure, metadata semantics, and reliability guarantees, these formats make it possible for diverse applications—ranging from professional broadcasting platforms to peer-to-peer communication tools—to exchange media over MoQT with **ultra-low latency**, high efficiency, and robust stream-aware delivery.

Moreover, the MoQ Working Group envisions that additional streaming formats will emerge in the future to accommodate evolving use cases. These may include support for **adaptive bitrate (ABR) video**, **real-time data synchronization** for multi-user applications, or advanced features such as **spatial audio delivery**. The extensibility of the MoQT protocol and its object-based hierarchy make it possible to adopt new formats without compromising on interoperability or transport efficiency.

By converging on standardized object-based formats such as WARP and RUSH, the MoQT ecosystem promises to break down vendor silos, foster innovation, and support the next generation of interactive and scalable media services.

3.4.3 Comparison with Traditional Protocols

Feature	**Traditional Protocols (e.g., RTP)**	**MoQ Streaming Formats**
Transport Coupling	Tightly bound to RTP	Decoupled from transport
Object Granularity	Packet-based	Object-based (frame, segment)
Multiplexing	Requires SDP negotiation	Built-in via QUIC streams
Prioritization	Limited (e.g., DSCP)	Per-object priority metadata
Cacheability	Poor	Cache-friendly via relayable objects
Join-in-progress	Complex (requires state)	Simple via group/track model

Table 3.1. Comparison of traditional protocols vs MoQ streaming formats

3.4.4 WARP

WARP is a streaming format designed for **low-latency**, **object-based**, and **loosely coupled** media distribution over MoQ.

It is particularly well-suited for scenarios where media needs to be delivered in **real time** to **multiple subscribers** and modern real-time applications such as **live streaming**, **interactive broadcasts**, and a**ugmented reality (AR)** or **virtual reality (VR)** media distribution.

WARP is designed to deliver multimedia content compliant with the **Common Media Application Format (CMAF)** [2] to enhance interoperability, support adaptive bitrate (ABR) switching, and facilitate distributed caching and delivery in heterogeneous network conditions [9, 17].

Design Goals

The design of the WARP format is grounded in several key objectives that directly address the challenges of modern, real-time media streaming. Foremost among these goals is the minimization of end-to-end latency, with the explicit aim of achieving sub-second delivery for live content. This capability is particularly vital for latency-sensitive applications such as live sports broadcasting, online gaming, and interactive video conferencing, where delays above one second can significantly impair user experience and interactivity [10, 6].

A second core objective is scalability, which is pursued through content-aware relaying. By structuring media with rich, machine-interpretable metadata, WARP empowers intermediary relay nodes to make intelligent, context-driven decisions—such as selectively caching, prioritizing, or dropping specific objects—based on real-time network and application demands. This adaptability not only supports graceful degradation under constrained network conditions but also enables in-network optimization strategies that go far beyond the capabilities of traditional, tightly coupled client-server architectures.

Furthermore, WARP is engineered for stream robustness and flexibility. Leveraging the underlying properties of QUIC, the format is able to deliver reliable media even over lossy or highly variable networks. Support for partial reliability, out-of-order object delivery, and targeted recovery mechanisms enables streams to maintain continuity and quality of experience even in the face of transient network failures. Collectively, these objectives reflect the broader philosophy of MoQ: to shift from monolithic client-server models toward a publish-subscribe paradigm in which media delivery is dynamically orchestrated by in-network relays and peer-aware optimizations [10].

Features

WARP distinguishes itself through a semantically rich and highly structured representation of media streams. At the heart of its approach is object-based granularity, whereby media content is segmented and transmitted as discrete, uniquely identifiable objects—such as video frames, audio samples, or metadata fragments. This enables precise control over delivery policies, dynamic prioritization, and selective retransmission or suppression based on application and network requirements.

A central feature of the WARP format is its extensive use of metadata tagging. Each media object is annotated with detailed metadata, including presentation timestamp (PTS), duration, dependency information (for example, referencing I-frame to P-frame relationships in video), and priority levels. This metadata is fundamental to enabling relays and endpoints to perform advanced stream management, synchronization, and error recovery.

The format further supports loose ordering and partial reliability, capitalizing on QUIC's native stream independence. This means that objects can arrive out of order or may be intentionally dropped without compromising the integrity of the overall stream—a property that markedly improves resiliency, especially for real-time content.

WARP is expressly optimized for relay-mediated delivery. Intermediary nodes are empowered to interpret object metadata and undertake a range of optimizations, including adaptive caching, intelligent reordering, dynamic prioritization, and even on-the-fly transcoding or prefetching. This approach not only enhances efficiency across the distribution chain but also enables more agile and adaptive media services.

Data Model and Hierarchy

In line with the broader MoQT specifications, the WARP format organizes media content within a robust four-level hierarchical structure. At the highest level, the *namespace* serves as a global scoping identifier, typically mapped to a service or domain (e.g., `live.example.com`), thereby facilitating global routing and content isolation.

Within each namespace, the *group* functions as a session-level identifier, such as a unique code for a particular live event or stream instance (e.g., `concert2025`). This allows for the aggregation of all media flows and resources associated with a specific session, supporting coordinated access control and lifecycle management.

The *track* represents a logical channel for an individual media type—be it video, audio, or captions. This granularity allows heterogeneous content to be multiplexed efficiently within the same group, supporting complex media experiences and seamless adaptation to varying user and device capabilities.

Finally, at the base of the hierarchy, the *object* constitutes the atomic transmission unit. An object may represent, for instance, a Common Media Application Format (CMAF) fragment, a group of pictures (GOP), or a single audio frame. Each object encapsulates both the raw media data and its associated metadata, enabling granular addressing, efficient transport, and dynamic optimization throughout the delivery path [6, 10].

Each object encapsulates a **segment** (e.g., a GOP or CMAF chunk) and is mapped onto a **QUIC stream**, allowing interoperability with existing encoders and decoders. This

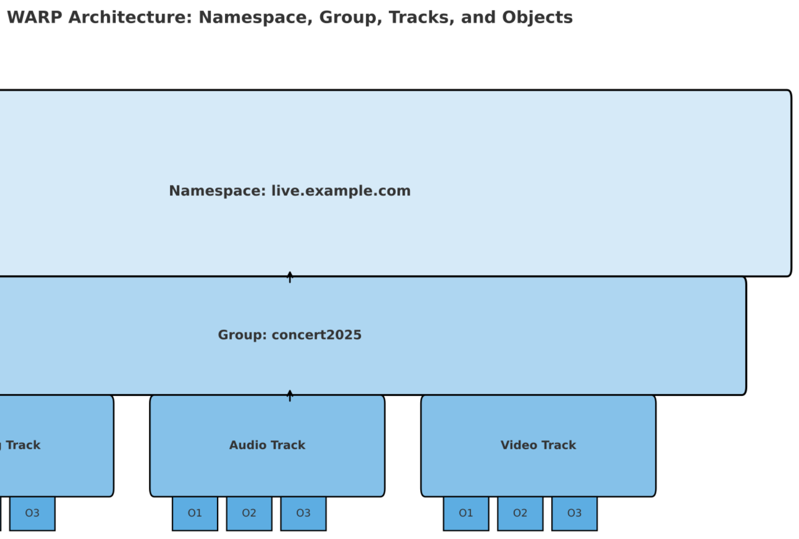

WARP Architecture: Namespace, Group, Tracks, and Objects
Namespace: live.example.com
Group: concert2025
Catalog Track
Audio Track
Video Track
O1
O2
O3
O1
O2
O3
O1
O2
O3

Figure 3.3. Warp data model hierarchy

ensures that WARP streams can be readily generated using commodity media pipelines and easily adapted into broader MoQ workflows [17].

Stream and Object Management

Streams are prioritized based on **media type** (e.g., audio is typically given higher priority than video), **recency** (newer samples are preferred over older ones), and **dependency information**. This facilitates **graceful degradation** in congested environments—allowing relays or clients to skip lower-priority frames or stale objects.

Objects are inserted into QUIC streams in the order produced by the encoder and consumed by the decoder, preserving **decoding semantics** while allowing **transport flexibility**. This approach aligns with modern adaptive streaming techniques where chunk-based segmentation (such as CMAF's use of ISOBMFF fragments) allows time-aligned, codec-independent delivery [2, 19].

Each codec bitstream must be packaged into a sequence of objects within a separate track, while media tracks should be **media-time aligned**.

Track Types and Publishing Rules

WARP distinguishes between two fundamental track types:

- **Media Track:** Carries media bitstreams such as H.264, AAC, or Opus, and represents a continuous sequence of time-aligned objects.
- **Catalog Track:** Provides metadata about the availability and structure of media tracks. It must be published before any media tracks, effectively bootstrapping the media session.

Catalog tracks facilitate **discovery**, **session negotiation**, and **late-join scenarios**. For example, a subscriber joining an ongoing stream can retrieve the latest catalog object to determine which media tracks are currently available and where to begin consumption.

WARP aligns with the architectural goals of MoQ: it is **inherently decoupled**, **publisher-subscriber oriented**, and **optimized for distributed, scalable delivery via QUIC**. Its metadata-first approach and flexible stream organization make it well-suited for **interactive and high-performance media applications** that cannot tolerate the latencies introduced by traditional HTTP-based delivery.

3.4.5 RUSH

In contrast to WARP, which is primarily oriented towards efficient downstream media distribution, the **RUSH** (Reliable Upload Streaming over HTTP/3) format has been designed to optimize upstream media ingestion within the Media over QUIC (MoQ) architecture. RUSH addresses the unique requirements of media publishers who must reliably, yet efficiently, transmit real-time or near-real-time content from the edge to in-network relays or ingest servers [21, 10].

A central tenet of RUSH is the preservation of precise timing and sequencing information for media objects as they are uploaded. This guarantees that the temporal structure of the original media production—such as the order and spacing of video frames or audio packets—is faithfully maintained from the producer through to the relay infrastructure. Accurate sequencing is critical for supporting seamless downstream playback, low-latency handoff, and precise synchronization across multiple tracks or modalities.

RUSH empowers media producers to leverage object-based partial reliability, a feature especially valuable in scenarios where network conditions are variable or retransmission opportunities are constrained. By enabling selective retransmission and targeted delivery of critical objects (for example, video keyframes), RUSH allows workflows to prioritize the most essential content and gracefully degrade non-essential or time-sensitive data when necessary. This design is particularly well suited for live production environments, where the retransmission window is intrinsically limited by the requirements of low-latency, real-time streaming.

Overall, RUSH represents a deliberate effort to provide a flexible, structured, and efficient approach to upstream media ingestion within the MoQ framework. By maintaining tight control over timing and sequencing, while supporting advanced reliability strategies, RUSH enables professional live production workflows and high-quality real-time applications to scale across modern internet infrastructures [21].

Comparison: WARP vs. RUSH

A clear understanding of the technical distinctions between the **WARP** and **RUSH** streaming formats is essential for system architects and developers working with the MoQT protocol suite. While both formats are designed to leverage QUIC's low-latency capabilities, they target fundamentally different directions of media flow within distributed streaming systems. The following table summarizes the principal aspects in which WARP and RUSH diverge in design, application, and technical implementation.

Both **WARP** and **RUSH** rely on an object-based abstraction, allowing flexible and fine-grained control over media delivery and ingest, but they apply these concepts to optimize opposite directions of the media pipeline. **WARP** is tailored for scalable distribution to a large number of receivers, allowing selective retransmission and object prioritization to achieve ultra-low-latency fan-out. In contrast, **RUSH** is optimized for reliable, time-preserving media ingestion from diverse sources, with special consideration for event sequence and partial reliability in high-pressure, live production scenarios.

By standardizing on these two formats within the MoQT framework, the working group enables interoperability and innovation, fostering an ecosystem in which both upstream contribution and downstream distribution can be independently optimized while benefiting from the common capabilities of QUIC—such as multiplexing, encryption, and connection migration.

Table 3.2. Detailed Comparison of WARP and RUSH Streaming Formats

Feature	**WARP**	**RUSH**
Primary Direction	Downstream (relay to consumer, publisher to audience)	Upstream (publisher to relay or server)
Design Focus	Scalable, low-latency media distribution to many consumers; object-based fan-out	Reliable, structured media ingestion with accurate event/-timing fidelity
Reliability Model	Partial reliability; allows selective retransmission and dropping of less important objects	Partial reliability; prioritizes timing and event order over strict retransmission of all objects
Timing Guarantees	Optimized for real-time and near-real-time delivery; can support deadlines and object priorities	Preserves publisher's timing and event sequencing for accurate upstream workflows
Object Model	Object-based (e.g., video frames, audio packets, timed metadata); supports prioritization	Object-based (e.g., media fragments, audio frames, events); emphasizes ingest order
Target Use Cases	Live event streaming, real-time collaboration, broadcast distribution, large-scale fan-out	Live production ingest, cloud relay, contribution feeds from field devices, real-time upstream workflows
Multipath/Relay Support	Designed for efficient relay, caching, and in-network fan-out	Supports relay and contribution models, with event and timing integrity
Partial Reliability	Yes (application can drop or deprioritize late or non-critical objects)	Yes (tolerates some loss to maintain timing and lower latency)
Integration in MoQT	Reference downstream distribution format	Reference upstream ingest format
Interoperability	Promotes interoperability between publishers, relays, and diverse consumer applications	Enables standardized ingest from heterogeneous devices to shared relays
Typical Data Types	Video frames, audio samples, captions, timed events	Encoded video/audio fragments, real-time events, control messages

3.4.6 Stream Management

Stream management in the context of Media over QUIC (MoQ) is inherently multi-layered, with each layer serving a distinct role in the end-to-end delivery of media content. At the highest layer, audio and video streams correspond directly to the content captured or rendered by applications. These streams encapsulate the primary media tracks as perceived by end-users during capture, playback, or real-time interaction [28].

Beneath the application layer, there exist intermediate streams which represent the encoded or decoded versions of the original audio or video content. For instance, an application may maintain separate streams for raw video capture, encoded compressed video (e.g., H.264), and subsequently decoded output for rendering or further processing. This separation allows for flexible handling of transcoding, format adaptation, and optimization workflows, all within the application domain.

At the MoQ protocol level, streams are further abstracted as input or output flows that correspond to the protocol's object-based transport primitives. MoQ streams are responsible for segmenting, encapsulating, and delivering media objects—such as video frames or audio packets—across the network, ensuring that content is uniquely identifiable and addressable throughout its journey from producer to consumer [29].

At the lowest layer, underlying transport mechanisms such as WebTransport and QUIC provide the foundation for reliable or partially reliable multiplexed delivery of streams. Here, streams are implemented as independent, bidirectional flows within a QUIC connection, supporting both reliable and datagram-based semantics. This enables MoQ to achieve highly efficient, low-latency media distribution while allowing for granular control over reliability, ordering, and flow prioritization [8, 15].

3.4.7 Use Cases

The layered stream management approach enabled by MoQ unlocks a wide array of modern media use cases. One prominent application is live streaming, encompassing real-time broadcasts of sports, concerts, and large-scale virtual events. In these scenarios, MoQ's object-centric, scalable architecture supports both low-latency delivery and efficient distribution across diverse network topologies [10].

Another significant use case is video conferencing, where the ability to scale distribution from small peer-to-peer calls to massive multi-party sessions is crucial. MoQ's flexible stream management enables selective forwarding, dynamic adaptation, and robust error recovery—features that are vital for maintaining media quality and interactivity in real-time communication environments.

Furthermore, MoQ's capabilities are highly applicable to real-time education and training platforms, where interactive content, live feedback, and low-latency streaming are essential. The protocol's support for hierarchical stream management and adaptive reliability empowers educators and learners to participate seamlessly in synchronous virtual classrooms and collaborative training exercises [28, 23].

Chapter 4

moq-js

On the client side, ***moq-js*** is used, a proof-of-concept application written in TypeScript, JSX, and Astro, developed by Luke Curley.

moq-js is a modern web application for live media publishing and playback, built around the **Media over QUIC (MoQ)** protocol. Its architecture exemplifies a clear separation of concerns, leveraging browser-native APIs and modular TypeScript code to orchestrate high-performance, low-latency streaming.

It runs within the browser and allows for the publishing or consumption of content via the MoQT protocol. The content in question can be captured via screen capture or from a microphone and webcam.

4.1 File and Module Structure Overview

The main logic is split across several layers:

- The UI (in `web/src/components/` and `web/src/pages/`)
- The core MoQ protocol implementation and media pipeline (in `lib/`)

Below is a quick map of the key modules and their architectural role:

- **UI Components:**
  - `web/src/components/publish.tsx`: Media publishing logic/UI
  - `web/src/components/watch.tsx`: Media playback logic/UI
- **Core Library (under `lib/`):**
  - `lib/contribute/`: Media capture, encoding, and publishing
  - `lib/playback/`: Media subscription, decoding, playback
  - `lib/transport/`: QUIC transport (WebTransport) and protocol objects
  - `lib/common/`: Utilities, logger, ring buffer, error handling

4.2 High-Level Workflow

1. **Publishing:**
   User starts a broadcast (UI: `publish.tsx`) → `Broadcast` class manages capture, encode, and sending chunks via `Publisher` → Chunks are packaged as `MoqObject` and transmitted over a `WebTransport` connection.

2. **Playback:**
   User selects a stream to watch (UI: `watch.tsx`) → `Player` class manages connection and subscription → Receives `MoqObject` chunks, which are decoded (using WebCodecs) and rendered to <video>/<AudioContext>.

3. **Transport/Protocol:**
   All media objects (audio/video) are sent as discrete protocol objects over QUIC, supporting loss recovery, reordering, and efficient subscription (object-based, not stream-based).

4.3 moq-js architecture - data flow

The media pipeline in `moq-js` is based on the object-based delivery model defined by the IETF [12].

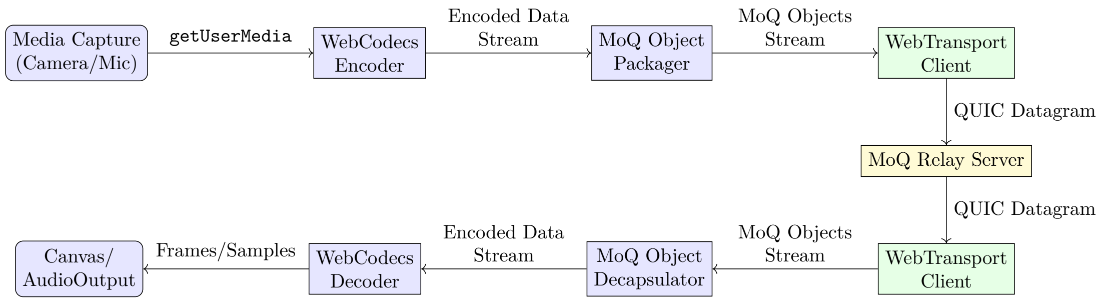


Figure 4.1. End-to-end MoQ Media Flow using `moq-js`

4.3.1 Publisher Workflow

1. **Capture:** Uses the MediaStream API to acquire video/audio input (`getUserMedia`).

2. **Encoding:** Encoded using WebCodecs VideoEncoder or AudioEncoder.

3. **Packaging:** Encoded chunks are wrapped in MoQ media objects, tagged with metadata (object ID, group ID, expiry).

4. **Transmission:** Sent via WebTransport datagrams for low-latency delivery [32].

4.3.2 Playback Workflow

1. **Connect:** WebTransport session is established with a MoQ relay.
2. **Subscribe:** Player subscribes to a track/namespace such as `moq.example/live/stream`.
3. **Receive:** Media objects are received over QUIC datagrams or streams.
4. **Decode:** Decoding is done with WebCodecs [31].
5. **Render:** Video is rendered on a canvas, and audio via AudioContext.

4.4 moq-js architecture — code level flow

The internal operation of **moq-js**, as illustrated in Figure 4.2, is characterized by a highly modular and event-driven architecture that elegantly separates concerns between user interface, protocol logic, and computationally intensive media processing. This section provides a comprehensive discussion of the code-level flow, focusing on the initialization, orchestration, and synchronization of media playback on the client.

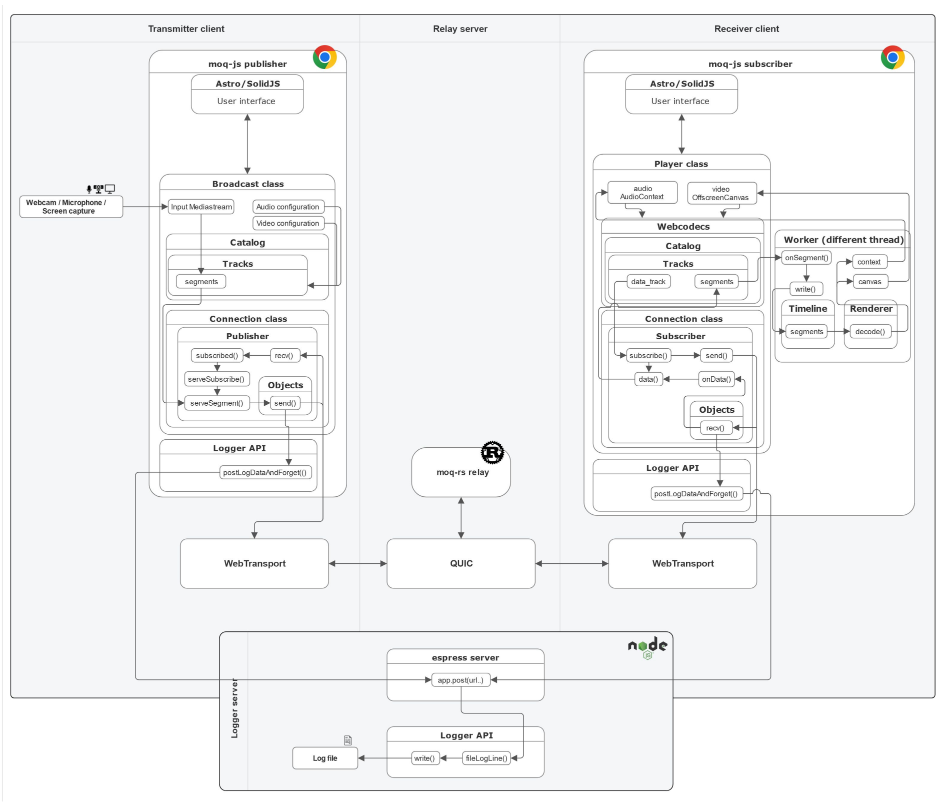


Figure 4.2. High-level architectural diagram of moq-js

Initialization and Reactive Binding

The entry point for playback in the client-side application is the **"Watch"** page, which is responsible for rendering the video interface and orchestrating the playback pipeline. In contemporary frameworks such as Astro or Solid.js, lifecycle management is often realized through reactive primitives such as `createEffect`. Within this effect, a video DOM element is instantiated, and a new instance of the **Player** class is created. This binding ensures that the player's state remains synchronized with the lifecycle of the UI component, facilitating both proper resource allocation and cleanup upon navigation or teardown.

Connection Establishment and Catalog Discovery.

Upon instantiation, the **Player** class is tasked with establishing a low-latency connection to the MoQ relay using the WebTransport API. Once the connection is confirmed, the Player fetches the media **Catalog** from the relay, which contains detailed metadata about available tracks, such as stream identifiers, codecs, language, and configuration required for proper subscription. The **Catalog** abstraction encapsulates track discovery, management of available media streams, and enables subsequent subscription and playback control.

Track Subscription and Worker Offloading.

For each track described in the catalog—beginning with initialization tracks (such as codec-specific headers or configuration objects), followed by continuous media tracks (e.g., video or audio streams)—the Player communicates with a dedicated JavaScript **Worker**. These workers are instantiated per-backend (e.g., for WebCodecs), enabling off-main-thread processing of decode and render operations. The communication pattern involves passing serialized track information and initialization messages from the Player to the Worker via `postMessage`. This not only improves responsiveness by offloading computationally intensive tasks, but also supports concurrent decoding and synchronization of multiple media tracks, such as multi-language audio or multiple video resolutions.

WebCodecs Backend and Audio Context Pipeline.

When the backend selected is **WebCodecs**—the most efficient option in modern browsers—the Player constructs a specialized instance of the backend's Player implementation. For audio playback, this entails the creation of a **Context** object, which orchestrates an **AudioContext** and an **AudioWorkletNode**. Upon initialization, the Context loads the audio worklet module and connects it to the browser's audio output path. This configuration enables sample-accurate, low-latency audio playback and allows for advanced processing, such as mixing, effects, or custom buffering strategies, all with minimal main-thread intervention.

Renderer Management and Threaded Media Pipeline.

Within each Worker, a dedicated **Renderer** class is responsible for the core rendering logic. For video, the Renderer decodes incoming segments and renders frames onto an offscreen or onscreen canvas element; for audio, it decodes chunks and pushes samples into a circular buffer (ring buffer) consumed by the AudioWorkletNode. The separation of decoding, buffering, and rendering into worker-based Renderers enables precise timing, minimizes audio/video drift, and significantly reduces the likelihood of UI stalls or dropped frames.

Audio Jitter Buffering and Smooth Playback.

A key architectural consideration in **moq-js** is the use of an audio ring buffer as an intermediary between the asynchronous arrival of media segments and the deterministic, sample-rate-driven pull model of browser audio playback. The Renderer is responsible for managing this buffer: upon receiving and decoding frames, it writes them to the buffer, while the AudioWorkletNode consumes them at regular intervals. This approach provides resilience against network jitter and transient decoder delays, ensuring continuous and artifact-free audio output.

Summary and Architectural Rationale.

The code-level flow of **moq-js** is a paradigmatic example of modern, event-driven web architecture: initialization is tightly coupled to UI state via reactive primitives; network and media logic is encapsulated in protocol-aware classes (Player, Catalog, Renderer, Context); and all heavy computation is safely delegated to Web Workers, enabling optimal responsiveness and scalability. The design achieves an elegant decoupling of concerns, robust handling of concurrency, and a clear path to future enhancements (such as adaptive bitrate, track switching, or multi-stream mixing).

This multi-layered, asynchronous approach underpins the library's ability to deliver low-latency, resilient, and high-quality media experiences directly in the browser, all in full alignment with the emerging Media over QUIC standards.

4.5 moq-js architecture - code level analysis

4.5.1 UI Integration: `web/src/components/publish.tsx`

The user interface (UI) in `moq-js` is designed to provide a seamless and highly interactive experience for live broadcasting, tightly coupling user actions to the media pipeline. The `publish.tsx` component is the entry point for users to initiate the process of capturing, encoding, and publishing audio/video content in real time.

At a high level, this UI component provides controls (e.g., a button) that, when activated, trigger a complete pipeline for live streaming. The core logic leverages imported abstractions from the library's protocol and media layers, specifically the `Broadcast`, `VideoEncoder`, and `AudioEncoder` classes, as well as the `Client` and `Connection` classes for network communication:

```
import { Broadcast, VideoEncoder, AudioEncoder } from "@kixelated/moq/contribute"
import { Client, Connection } from "@kixelated/moq/transport"
// ... UI state and event hooks ...
const broadcast = new Broadcast({ ...mediaOptions })
await broadcast.start()
```

Detailed analysis:

Abstraction and Modularity

The use of the `Broadcast` class demonstrates a clear separation of concerns. Rather than directly manipulating browser APIs in the UI logic, the component delegates media pipeline responsibilities (capture, encode, package, transmit) to reusable, well-encapsulated abstractions. This approach enhances code maintainability, supports code reuse, and aligns with the principles of modular software engineering.

Reactive and Event-Driven Workflow

User interaction (e.g., clicking a "Go Live" button) triggers asynchronous operations. This is typically managed through React or Solid.js hooks, where UI state transitions (such as "broadcasting: true/false") control component lifecycle and effect triggers. The asynchronous nature of media capture and transport is handled via promises and async/await, ensuring UI responsiveness and robust error handling.

MediaOptions and Dynamic Configuration

The `Broadcast` constructor accepts a configuration object (`mediaOptions`) that encapsulates user-selected parameters, such as audio/video sources, resolution, frame rate, and codec preferences. This pattern allows for dynamic adaptation to user or device capabilities, and supports advanced features like selecting between screen capture and camera, toggling audio, or switching between hardware and software codecs.

Orchestration of the Pipeline

Calling `await broadcast.start()` triggers a series of chained operations:

1. The `Broadcast` class invokes the MediaStream API (`getUserMedia`) to request permissions and access hardware input devices.

2. It then creates instances of `VideoEncoder` and/or `AudioEncoder` from the `@kixelated/moq/contrib` module, configuring them according to the selected options.

3. As media frames and audio samples are captured, they are asynchronously passed to the encoders, which output compressed media chunks suitable for network transmission.

4. Each encoded chunk is immediately packaged with metadata (timestamps, group/object IDs) in preparation for transport.

5. The `Client` and `Connection` classes—from `@kixelated/moq/transport`—are used to establish and maintain a WebTransport (QUIC) session to the relay, supporting real-time delivery of encoded objects.

This architecture enables a low-latency, continuous pipeline from camera/microphone to the network, with minimal main-thread blocking.

Error Handling and User Feedback

At each stage, potential errors (permissions denied, codec unsupported, transport failure) are captured and surfaced to the UI, enabling robust user feedback and facilitating rapid troubleshooting.

Internals of the `Broadcast` Class

The `Broadcast` class is the heart of the publishing pipeline. It encapsulates all logic needed for media capture, encoding, and object-based packaging for transmission via MoQ. Below is a representative excerpt from `lib/contribute/broadcast.ts`, illustrating key constructor logic and lifecycle methods:

```
// Broadcast class internals (example, real code may differ)
constructor(options) { /* ...initialize state... */ }

async start() {
  // Acquire MediaStream, initialize encoders, connect to relay, start sending objects
}

async stop() {
  // Gracefully tear down pipeline: close encoders, stop tracks, close connection
}
```

In-depth Class Explanation:

- **Constructor:** The constructor initializes the pipeline based on provided configuration options. It prepares references to the media input (audio/video), codec configuration, and sets up internal state, including object counters and encoder instances. Error checks and fallbacks are put in place to handle unsupported device or codec conditions gracefully.

- **start() method:** The asynchronous `start` method orchestrates the complete publishing flow. It acquires a MediaStream using the browser's media APIs, initializes encoders for video and audio (as needed), and establishes a connection to the MoQ relay using the `Client` abstraction. The method enters a loop or attaches event listeners for handling frame/audio data as it becomes available, ensuring minimal latency between capture and network transmission. The use of async/await ensures that resource acquisition and errors are managed robustly, and the UI can be kept informed of pipeline status.

- **stop() method:** This method gracefully tears down the pipeline. It closes the encoders, stops the media tracks, and ensures that the network connection is closed cleanly. This guarantees that no resources are leaked, and the user can reliably start and stop broadcasting sessions without stale or conflicting state.

4.5.2 Transport and Protocol (From `lib/transport/`)

The `lib/transport/` directory in `moq-js` forms the backbone of the client's real-time communication capabilities. Here, the implementation brings together the Media over QUIC (MoQ) object protocol, session and object management, and the low-level QUIC transport (via WebTransport) into a modular, event-driven layer that is both standards-aligned and extensible.

QUIC/WebTransport Client Abstraction: `client.ts`

At the core of the transport system is the client abstraction, which encapsulates all connection lifecycle logic using the browser's `WebTransport` API. This abstraction hides the complexities of session negotiation, bidirectional stream management, and datagram delivery.

```
// lib/transport/client.ts

export class Client {
  #transport: WebTransport | undefined;
  #url: string;

  constructor(url: string) {
    this.#url = url;
  }

  async connect() {
    this.#transport = new WebTransport(this.#url);
    await this.#transport.ready;
  }

  sendDatagram(data: Uint8Array) {
    if (!this.#transport) throw new Error("not connected");
    this.#transport.datagrams.writable.getWriter().write(data);
  }

  close() {
    this.#transport?.close();
  }
}
```

Session Lifecycle and Multiplexing

The `Client` class manages the lifecycle of the transport connection. It provides methods for connection establishment, reliable datagram transmission, and clean teardown. Multiplexed media objects and control messages are sent and received over the same session, utilizing QUIC's built-in stream and datagram abstractions for parallel, low-latency communication.

Publisher Role: `publisher.ts`

The publisher module provides a high-level interface for packaging and sending encoded media data as protocol objects over the QUIC transport. It abstracts over MoQ-specific metadata (e.g., groupings, expiration) and manages the details of packaging objects and delivering them via the established client session.

```
// lib/transport/publisher.ts

export class Publisher {
  #client: Client;

  constructor(client: Client) {
    this.#client = client;
  }

  async publish(object: MoqObject) {
    const buffer = object.toBuffer();
    this.#client.sendDatagram(buffer);
  }
}
```

Subscriber Role: `subscriber.ts`

On the receiving side, the subscriber manages the subscription to particular tracks or namespaces, and handles the dispatch and decoding of received MoQ objects. Subscription requests are serialized and sent as control messages, and data flows back via the same transport infrastructure.

```
// lib/transport/subscriber.ts

export class Subscriber {
  #client: Client;

  constructor(client: Client) {
    this.#client = client;
  }
  subscribeTrack(trackId: number) {
    const msg = { type: "subscribe", trackId };
    this.#client.sendDatagram(encodeControlMessage(msg));
  }

  handleObject(buffer: Uint8Array) {
    const object = MoqObject.fromBuffer(buffer);
    // Pass object to playback/decoding layer
  }
}
```

4.5.3 `lib/transport/object.ts` — The MoQ Object Layer

The file `lib/transport/object.ts` is pivotal within **moq-js**: it defines the core representation and serialization logic for media objects as they are sent and received over QUIC. The abstraction here is crucial: **MoQ objects** serve as the atomic units for media transport, encapsulating both payload (media chunk) and the metadata required for ordering, deduplication, reliability, and protocol compliance.

Object Representation

A simplified and annotated version of the core class:

```
// lib/transport/object.ts
export class MoqObject {
  readonly group: number
  readonly object: number
  readonly expires: number | undefined
  readonly payload: Uint8Array

  constructor({ group, object, expires, payload }) {
    this.group = group
    this.object = object
    this.expires = expires
    this.payload = payload
  }
}
```

Detailed explanation:

- `group`: Identifies the media group (e.g., a video or audio track) that this object belongs to. It enables grouping of related objects for subscription and delivery.
- `object`: A strictly increasing integer that uniquely identifies this object within its group. This is vital for ordering, loss detection, deduplication, and caching by the relay/server.

- `expires`: An optional timestamp (milliseconds since epoch) marking when this object should be considered expired or non-cacheable. This supports temporary caching and replay for late joiners.
- `payload`: The encoded media bytes (e.g., a VP8 keyframe or Opus audio packet). It's stored as a `Uint8Array` for efficient transmission.

Design rationale: This minimal representation enables flexible and protocol-compliant serialization, rapid deserialization, and extensibility for future fields. Object-level encapsulation is fundamental to MoQ's ability to support object-based media, unlike legacy stream-oriented transports.

Serialization and Deserialization

For network transport, objects must be converted to and from wire format. `lib/transport/object.ts` typically provides static methods for this purpose (here shown conceptually):

```
static fromBuffer(buffer: Uint8Array): MoqObject {
  // Parse group, object, expires, and payload from the wire buffer
  const group = ... // parse uint
  const object = ... // parse uint
  const expires = ... // optional, parse if present
  const payload = ... // rest of buffer
  return new MoqObject({ group, object, expires, payload })
}

toBuffer(): Uint8Array {
  // Serialize fields into a buffer for transmission
  // ... (write group, object, expires, payload)
  return buffer
}
```

Detailed explanation:

- **Deserialization (`fromBuffer`)**: Reconstructs a `MoqObject` from a QUIC datagram or stream by parsing fields in protocol order. This allows any receiver (relay, subscriber) to reconstruct the object metadata and media payload.
- **Serialization (`toBuffer`)**: Encodes the `MoqObject` instance for transmission. This step ensures the on-the-wire format is compact and fully compliant with the MoQ IETF specification, supporting interoperability.

Why is this important? Object-level serialization/deserialization ensures that every hop (browser, relay, server, subscriber) can handle, cache, and potentially replay media objects independently—enabling the MoQ protocol's low-latency and robust semantics.

Usage and Cross-Reference in the Pipeline

Packaging at the Publisher:

```
// lib/transport/publisher.ts
const obj = new MoqObject({
  group: videoTrackId,
  object: nextObjectId++,
  expires: Date.now() + 2000,
  payload: encodedChunk
});
await this.connection.sendDatagram(obj.toBuffer());
```

Decoding at the Subscriber:

```
// lib/transport/subscriber.ts
const obj = MoqObject.fromBuffer(receivedBuffer);
// ... Pass obj.payload to decoder, use obj.group/object for ordering and
↪  deduplication
```

Extended explanation:

- On the **sending side**, after a media chunk is encoded, it is immediately wrapped as a `MoqObject` and serialized for network transmission, ensuring all required metadata (track group, object order, expiration) travels with the payload.
- On the **receiving side**, every incoming datagram is first parsed into a `MoqObject` so that track management, loss handling, and decoding logic can be applied based on protocol metadata.

Extensibility and Protocol Compliance

The class is designed for future extension: additional fields (such as priority, dependencies, or encryption metadata) can be added with backward-compatible changes, as the parsing logic can adapt to optional or versioned fields.

This layer is also responsible for:

- Enforcing object-level integrity (e.g., discarding duplicates or late objects based on object ID and expiration)
- Enabling caching/replay at both relay and client
- Supporting the full MoQ protocol's requirements as tracked by the IETF draft [12]

4.5.4 Track Management and Subscription

The concept of **tracks** is foundational to the Media over QUIC (MoQ) protocol and central to the design of `moq-js`. Unlike traditional monolithic media streams, MoQ treats each audio or video stream—whether a main video feed, an audio language track, or a timed metadata channel—as an independently addressable *track*. This architectural choice brings several benefits: it enables selective subscription, granular flow control, parallel media delivery, efficient object caching, and robust support for multi-language or adaptive scenarios.

Code Structure and Responsibilities Track discovery, subscription, and management in `moq-js` are encapsulated within distinct modules, each responsible for a layer of the end-to-end playback pipeline. As exemplified in the code from `lib/playback/index.ts` and `lib/transport/subscriber.ts`:

```
// lib/playback/index.ts
export class Player {
  private tracks = new Map();

  subscribe(trackInfo) {
    const track = new Track(trackInfo);
    this.tracks.set(trackInfo.id, track);
    this.connection.subscribeTrack(trackInfo.id);
  }
}

// lib/transport/subscriber.ts
export class Subscriber {
  subscribeTrack(trackId: string) {
    this.connection.sendControl({ type: 'subscribe', trackId });
  }
}
```

Track Registry and Dynamic Management

The `Player` class maintains a dynamic registry (`tracks`) of all active tracks as a `Map`, keyed by unique track identifiers. This registry supports efficient lookup, addition, and removal of tracks at runtime—a necessity for scenarios like dynamic language switching, adding or removing camera feeds, or live multi-angle switching. Each value is a `Track` instance, encapsulating state and configuration for that media stream.

Subscription Orchestration

The `subscribe(trackInfo)` method in `Player` orchestrates the track subscription process. Upon receiving metadata (e.g., from a session catalog), it constructs a new `Track` object, registers it, and delegates the actual protocol-level subscription to the transport layer. This decoupling of UI/session logic from protocol mechanics supports both modularity and testability.

Protocol-Level Subscription

The transport abstraction, embodied by the `Subscriber` class, exposes a `subscribeTrack(trackId)` method. Here, the subscription intent is serialized into a control message and sent via the active connection (typically over QUIC datagrams or streams). The control message structure— `type: 'subscribe', trackId` —aligns with the MoQ protocol specification, ensuring interoperability with relays and other clients.

Asynchronous and Event-Driven Design

Subscriptions are inherently asynchronous: upon issuing a subscribe request, the client must wait for media object availability (possibly signaled by server messages or track announcements). This event-driven paradigm allows for responsive user interactions, dynamic UI updates (e.g., enabling or disabling controls), and seamless integration of late-joining or on-the-fly track changes.

Selective and Adaptive Media Delivery

Independent tracks empower clients to selectively subscribe only to those streams relevant to the user. For example, in a live conference, a viewer could subscribe only to the main video and a chosen language channel, reducing bandwidth and decoding costs. This design also lays the foundation for advanced features like per-track adaptive bitrate, track prioritization, and bandwidth estimation.

Standards Alignment

By making tracks explicit first-class objects in both API and protocol, `moq-js` maintains alignment with the IETF MoQ drafts [12], ensuring that the client will remain compatible as the standard evolves and as new relay implementations are developed.

4.5.5 Worker Communication Architecture

One of the most critical architectural choices in `moq-js` is the extensive use of Web Workers to offload heavy media processing from the main UI thread. In modern web browsers, the main thread is responsible for user interaction, DOM updates, and general UI responsiveness. However, real-time media tasks such as video decoding, audio processing, and complex synchronization can introduce noticeable latency or UI "jank" if performed on the main thread. To address this, `moq-js` delegates compute-intensive tasks to dedicated workers, thereby maintaining fluid and responsive user experiences even under high processing loads.

Worker Initialization and Main Thread Interface

Workers are instantiated directly from the main thread—typically in the playback UI component—using the standard Web Worker API. The worker script is specified as a module to support modern ES module imports:

```
// Main thread: setting up a worker for media playback
const playerWorker = new Worker(
new URL('../worker/player.worker.ts', import.meta.url),
```

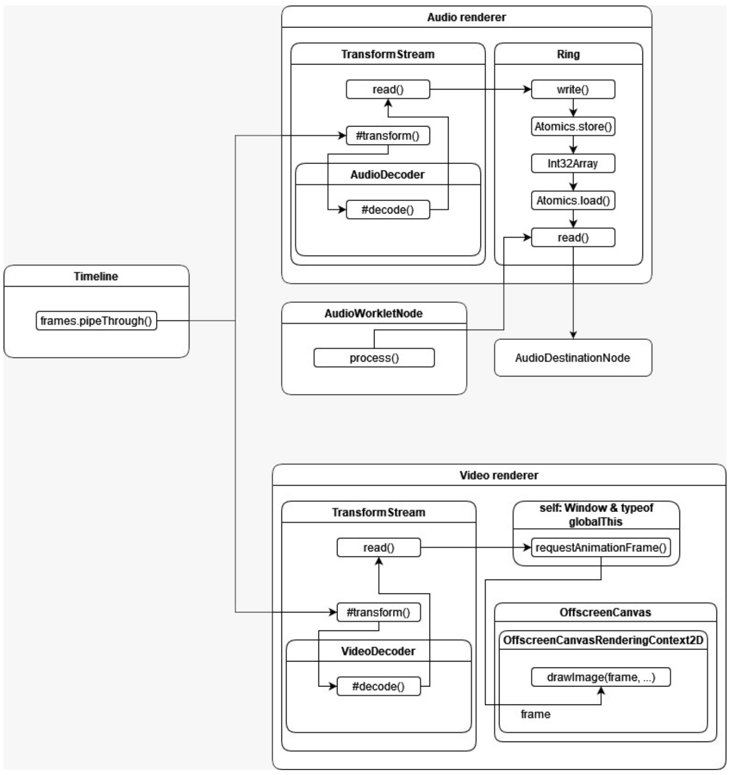


Figure 4.3. Workers architectural diagram

```
{ type: 'module' }
);

// Registering a listener for messages from the worker
playerWorker.onmessage = (event) => {
if (event.data.type === 'frame') {
renderFrame(event.data.payload);
} else if (event.data.type === 'error') {
displayError(event.data.reason);
}
};

// Sending an initialization message to the worker with stream metadata
playerWorker.postMessage({ type: 'init', tracks, codecConfig });
```

This initialization process achieves several goals:

- The worker is loaded as a module, enabling modular code structure and import of TypeScript/ESM dependencies.
- A listener is set up on `onmessage` to handle frames (for rendering), errors, and any additional notifications (e.g., statistics or state updates).

- The first message posted is an `init` message, delivering all configuration needed to begin processing, including track information and codec configurations.

Worker Event Loop and Decoding Pipeline

Inside the worker script (`web/src/worker/player.worker.ts`), an event-driven architecture is used. The worker listens for commands from the main thread, such as `init`, `decode`, or control commands (pause, resume, stop), and responds by processing and posting results back.

```
// player.worker.ts
self.onmessage = async (event) => {
if (event.data.type === 'init') {
await setupDecoder(event.data.codecConfig);
// Optional: notify main thread we're ready
self.postMessage({ type: 'ready' });
} else if (event.data.type === 'decode') {
try {
const frame = await decodeChunk(event.data.chunk);
self.postMessage({ type: 'frame', payload: frame }, [frame]);
} catch (error) {
self.postMessage({ type: 'error', reason: error.message });
}
} else if (event.data.type === 'flush') {
// For codecs that support flushing buffered frames (e.g., at end-of-stream)
await flushDecoder();
self.postMessage({ type: 'flushed' });
}
};
```

Key points:

- The worker is fully asynchronous and event-driven. Each incoming message is handled independently, allowing for concurrency and responsive operation.
- Upon initialization, the worker sets up the appropriate decoder (e.g., `WebCodecs.VideoDecoder`) based on the supplied codec configuration.
- When a `decode` message is received, the worker attempts to decode the chunk, handling both success (post frame to main thread) and failure (post error) cases. This ensures robust error handling and recovery.
- Large binary objects (such as decoded frames) are transferred using the structured clone algorithm with transferable objects for performance.
- The architecture supports additional commands such as flush, pause, or reconfigure, allowing for advanced playback scenarios.

Data Flow: Main Thread ↔ Worker

The overall data flow between the main thread and the worker can be summarized as follows:

1. **Initialization**: Main thread sends an `init` message with tracks and codec configurations.
2. **Subscription**: When a new media object arrives (from network), the main thread sends a `decode` message with the media chunk to the worker.
3. **Processing**: The worker decodes the chunk (video or audio frame).
4. **Result Return**: Worker posts back the decoded frame or an error message. Frames are rendered (video: `canvas`, audio: `AudioContext`) or buffered.
5. **Control**: Additional commands (flush, reset, statistics, etc.) can be sent and responded to as required.

Example: Video Decoding with WebCodecs in Worker

A typical video decoding pipeline in the worker may look like this:

```
let videoDecoder;

async function setupDecoder(codecConfig) {
videoDecoder = new VideoDecoder({
output: (frame) => {
// Send frame back to main thread (transferable)
self.postMessage({ type: 'frame', payload: frame }, [frame]);
},
error: (err) => {
self.postMessage({ type: 'error', reason: err.message });
}
});
videoDecoder.configure(codecConfig);
}

async function decodeChunk(chunk) {
return new Promise((resolve, reject) => {
videoDecoder.decode(chunk);
// Output callback will post frame
resolve();
});
}
```

Performance, Scalability, and Best Practices

The use of Web Workers for media pipelines in `moq-js` yields several benefits:

- **Responsiveness**: By moving heavy computation off the main thread, the user interface remains interactive and animation smooth, even during periods of intensive decoding or buffering.
- **Parallelism**: Multiple tracks (e.g., main video, secondary video, audio, data channels) can be handled in parallel, each in its own worker, further boosting throughput and reliability.
- **Scalability**: This design allows for more advanced streaming features, such as multi-view (picture-in-picture), adaptive streaming, or live overlays, without degrading the user experience.
- **Maintainability**: Encapsulating decoding logic in workers isolates it from UI logic, making the codebase easier to maintain, debug, and extend.

Error Handling and Robustness

A robust worker communication architecture must anticipate failures at multiple levels: unsupported codecs, decoding errors, buffer overflows, or communication breakdowns. By standardizing error messages (as shown in the code above) and ensuring the main thread can react (e.g., pausing playback, prompting the user, or switching codecs), `moq-js` delivers a resilient user experience.

4.5.6 Audio and Video Encoding

The `moq-js` project handles media encoding entirely in the browser using the [WebCodecs API][31]. Unlike earlier approaches which might rely on slower JavaScript codecs or server-side preprocessing, this architecture allows for real-time, low-latency encoding and direct transport of compressed video and audio streams. The encoding logic is modularized into dedicated classes for each media type, each designed to abstract browser quirks and provide a unified interface to the rest of the publishing pipeline.

Video Encoding: `lib/contribute/video.ts`

The video encoding pipeline is implemented in the `VideoEncoder` class, which encapsulates the setup, feeding, and lifecycle of a WebCodecs `VideoEncoder` instance.

```
// lib/contribute/video.ts

export class VideoEncoder {
  #encoder: globalThis.VideoEncoder;
  #keyframe: boolean = false;
  #config: VideoEncoderConfig;
  #output: (chunk: EncodedVideoChunk, meta: VideoEncoderEncodeOptions) => void;

  constructor(config: VideoEncoderConfig, output: (chunk: EncodedVideoChunk, meta:
  ↪  VideoEncoderEncodeOptions) => void) {
    this.#config = config;
    this.#output = output;
    this.#encoder = new VideoEncoder({
      output: this.#output,
      error: (e) => console.error("VideoEncoder error:", e),
    });
    this.#encoder.configure(this.#config);
  }

  encode(frame: VideoFrame, options?: VideoEncoderEncodeOptions) {
    this.#encoder.encode(frame, options ?? { keyFrame: this.#keyframe });
    this.#keyframe = false;
  }

  requestKeyFrame() {
    this.#keyframe = true;
  }

  flush(): Promise<void> {
    return this.#encoder.flush();
  }

  close() {
    this.#encoder.close();
  }
}
```

Abstraction and Modularity

The VideoEncoder class is an abstraction over the WebCodecs API, shielding the rest of the codebase from direct API usage. This abstraction provides a stable and easily testable interface for video encoding operations. By holding internal state (such as keyframe requests and configuration), it supports adaptive encoding and easy pipeline extension, such as simulcast or dynamic reconfiguration.

Keyframe Control and Real-Time Adaptation

A crucial feature of real-time video streaming is the ability to force keyframe (intra-frame) generation, which is essential for stream recovery after packet loss or for supporting new subscribers in a live session. The class tracks whether a keyframe should be generated on the next encode call and provides a `requestKeyFrame()` method, which is invoked based on network or protocol feedback.

Output Callbacks and Pipeline Integration

Instead of handling encoded output internally, the class expects an output callback, provided at construction, which is called every time a chunk is encoded. This design decouples the encoding process from network transmission, allowing for future enhancements such as chunk reordering, batching, or analytics.

Error Handling and Resource Management

All errors from the WebCodecs encoder are captured and logged, providing a single place to implement advanced error recovery or fallback logic. The class also exposes `flush()` and `close()` methods, which ensure that all pending frames are processed and that browser resources are released properly—a critical concern for long-lived browser applications.

Audio Encoding: `lib/contribute/audio.ts`

The audio encoding pipeline mirrors the structure of the video encoder but is tailored for audio-specific requirements.

```typescript
// lib/contribute/audio.ts

export class AudioEncoder {
  #encoder: globalThis.AudioEncoder;
  #config: AudioEncoderConfig;
  #output: (chunk: EncodedAudioChunk, meta: AudioEncoderEncodeOptions) => void;

  constructor(config: AudioEncoderConfig, output: (chunk: EncodedAudioChunk, meta:
  ↪  AudioEncoderEncodeOptions) => void) {
    this.#config = config;
    this.#output = output;
    this.#encoder = new AudioEncoder({
      output: this.#output,
      error: (e) => console.error("AudioEncoder error:", e),
    });
    this.#encoder.configure(this.#config);
  }

  encode(audioData: AudioData, options?: AudioEncoderEncodeOptions) {
    this.#encoder.encode(audioData, options);
  }

  flush(): Promise<void> {
    return this.#encoder.flush();
  }

  close() {
    this.#encoder.close();
  }
}
```

Unified Callback Pattern

As with the video encoder, the audio encoder expects an output callback, which receives encoded audio chunks. This enables easy composition in the overall publishing pipeline—chunks can be packaged, buffered, or sent over the network as soon as they are produced.

Dynamic Configuration and Codec Flexibility

The encoder is initialized and configured at construction with a configuration object, allowing the application to choose codecs, sample rates, and channel counts dynamically, depending on user preferences, device capabilities, or protocol negotiation.

Error Handling and Adaptability

All errors encountered by the audio encoder are reported via the error callback, where they can be logged, surfaced to the user, or used to trigger fallback mechanisms (such as switching to a different codec). The class also provides `flush()` and `close()` methods for safe pipeline teardown and resource management.

Integration with the Broadcast Pipeline

Both audio and video encoders are used within the broader broadcast (publishing) pipeline of `moq-js`. For each captured frame or audio sample, the appropriate encoder's `encode()` method is called, and the resulting encoded chunk is passed to the transport packaging and transmission layers. This architecture cleanly separates media capture, encoding, and network logic, and enables the use of workers or other concurrency models for scalability.

4.5.7 Audio and Video Decoding (WebCodecs Backend)

The decoding process in `moq-js` leverages the WebCodecs API, but its design goes far beyond a simple wrapper. Both video and audio decoding are orchestrated within specialized `Renderer` classes, making use of transform streams, precise decoder configuration, and advanced resource management. The classes are designed to run inside a Worker context, ensuring smooth and jank-free user experience even during high-bitrate or multi-track playback.

Video Decoding: `lib/playback/webcodecs/video.ts`

The `Renderer` class for video handles the full pipeline from encoded video frames to rendering on a canvas. The core flow includes a transform stream, dynamic decoder configuration, and real-time frame rendering:

```
// lib/playback/webcodecs/video.ts

export class Renderer {
  #canvas: OffscreenCanvas
  #timeline: Component
  #decoder!: VideoDecoder
  #queue: TransformStream<Frame, VideoFrame>

  constructor(config: Message.ConfigVideo, timeline: Component) {
    this.#canvas = config.canvas
    this.#timeline = timeline

    this.#queue = new TransformStream({
      start: this.#start.bind(this),
      transform: this.#transform.bind(this),
    })

    this.#run().catch(console.error)
  }

  async #run() {
    const reader = this.#timeline.frames.pipeThrough(this.#queue).getReader()
    for (;;) {
      const { value: frame, done } = await reader.read()
      if (done) break

      self.requestAnimationFrame(() => {
        this.#canvas.width = frame.displayWidth
        this.#canvas.height = frame.displayHeight

        const ctx = this.#canvas.getContext("2d")
        if (!ctx) throw new Error("failed to get canvas context")

        ctx.drawImage(frame, 0, 0, frame.displayWidth, frame.displayHeight)
        frame.close()
      })
    }
  }

  #start(controller: TransformStreamDefaultController<VideoFrame>) {
    this.#decoder = new VideoDecoder({
      output: (frame: VideoFrame) => {
        controller.enqueue(frame)
      },
      error: console.error,
    })
  }

  #transform(frame: Frame) {
    if (this.#decoder.state !== "configured") {
      const { sample, track } = frame

      const desc = sample.description
      const box = desc.avcC ?? desc.hvcC ?? desc.vpcC ?? desc.av1C
      if (!box) throw new Error(`unsupported codec: ${track.codec}`)

      const buffer = new MP4.Stream(undefined, 0, MP4.Stream.BIG_ENDIAN)
      box.write(buffer)
      const description = new Uint8Array(buffer.buffer, 8)

      if (!MP4.isVideoTrack(track)) throw new Error("expected video track")

      this.#decoder.configure({
        codec: track.codec,
        codedHeight: track.video.height,
        codedWidth: track.video.width,
        description,
      })
    }

    const chunk = new EncodedVideoChunk({
      type: frame.sample.is_sync ? "key" : "delta",
      data: frame.sample.data,
      timestamp: frame.sample.dts / frame.track.timescale,
    })

    this.#decoder.decode(chunk)
  }
}
```

TransformStream Pipeline

Decoding is performed via a `TransformStream`, which acts as an asynchronous, composable processing pipeline. This design enables natural backpressure management between the timeline of received frames and the decoder. The use of `TransformStream` ensures that if decoding slows down (for instance, due to a heavy workload or slow rendering), the system can signal the frame source to adjust accordingly, preventing buffer bloat and memory exhaustion. It also allows each part of the pipeline—from frame acquisition to decode and render—to remain loosely coupled and independently scalable.

Dynamic Decoder Configuration

Unlike static decoder initialization, the decoder in `moq-js` is only configured when the first frame is available. This allows the system to extract crucial codec and track metadata (such as the codec type, frame dimensions, and container-specific initialization boxes) at runtime. Such a lazy configuration approach is essential for supporting adaptive streaming, multiple codecs (H.264, VP9, AV1), and for handling track switches without unnecessary resource consumption or pipeline restarts.

MP4 Codec Extraction

A standout feature is the extraction and direct use of codec initialization data from the underlying MP4 sample. This includes parsing initialization boxes such as `avcC` for H.264/AVC or `av1C` for AV1. The relevant binary blobs are injected directly into the decoder configuration, ensuring compatibility with a wide variety of source media and container formats. This approach also lays the groundwork for easy extensibility to additional codecs or formats as browser and WebCodecs support grows.

Real-time Rendering

Decoded frames are rendered to an `OffscreenCanvas` within a `requestAnimationFrame` callback. This not only decouples heavy decode workloads from the UI, but also ensures that rendering occurs in sync with the browser's display refresh cycle, minimizing visual artifacts and ensuring smooth playback. By using `OffscreenCanvas`, the system is ready for worker-thread rendering or WebGL acceleration as required for advanced use cases.

Error Handling

Decoder errors (such as codec incompatibility, corrupted frames, or resource exhaustion) are captured via dedicated error callbacks and logged to the console. While the reference implementation logs these errors, this design allows future integration of more advanced error handling strategies, such as codec fallback, error concealment, or dynamic user feedback.

Resource Management

Each decoded frame is explicitly closed after rendering, which is vital for memory management in the browser environment. Explicit resource management prevents memory leaks and ensures that the garbage collector can reclaim resources promptly, a critical concern for long-running or multi-track sessions.

Audio Decoding: `lib/playback/webcodecs/audio.ts`

The audio decoding pipeline, similarly, is encapsulated in an audio `Renderer` class, which handles decoder setup, streaming, and buffering via a ring buffer:

```typescript
// lib/playback/webcodecs/audio.ts

export class Renderer {
  #context?: AudioContext
  #ring: Ring
  #timeline: Component
  #decoder!: AudioDecoder
  #stream: TransformStream<Frame, AudioData>

  constructor(config: Message.ConfigAudio, timeline: Component) {
    this.#timeline = timeline
    this.#ring = new Ring(config.ring)
    this.#context = config.context

    this.#stream = new TransformStream({
      start: this.#start.bind(this),
      transform: this.#transform.bind(this),
    })

    this.#run().catch(console.error)
  }

  #start(controller: TransformStreamDefaultController) {
    this.#decoder = new AudioDecoder({
      output: (frame: AudioData) => {
        controller.enqueue(frame)
      },
      error: console.warn,
    })
  }

  #transform(frame: Frame) {
    if (this.#decoder.state !== "configured") {
      const track = frame.track
      if (!MP4.isAudioTrack(track)) throw new Error("expected audio track")

      this.#decoder.configure({
        codec: track.codec,
        sampleRate: track.audio.sample_rate,
        numberOfChannels: track.audio.channel_count,
      })
    }

    const chunk = new EncodedAudioChunk({
      type: frame.sample.is_sync ? "key" : "delta",
      timestamp: frame.sample.dts / frame.track.timescale,
      duration: frame.sample.duration,
      data: frame.sample.data,
    })

    this.#decoder.decode(chunk)
  }

  async #run() {
    const reader = this.#timeline.frames.pipeThrough(this.#stream).getReader()
    for (;;) {
      const { value: frame, done } = await reader.read()
      if (done) break

      const written = this.#ring.write(frame)
      if (written < frame.numberOfFrames) {
        console.warn(`dropped ${frame.numberOfFrames - written} audio samples`)
      }
    }
  }
}
```

Ring Buffer

Decoded audio data is written into a dedicated ring buffer. The ring buffer serves as a robust jitter buffer, absorbing irregularities in network or decoding delays and ensuring that audio output remains smooth even under real-time constraints. If the ring buffer becomes full, additional frames are dropped—a strategy that prioritizes real-time playback over perfect fidelity, in line with the needs of live streaming.

Decoder Configuration

Like the video pipeline, the audio decoder is configured only when the first frame is available, allowing the extraction of essential parameters such as codec type, sample rate, and channel count. This makes the pipeline highly adaptable to different stream types and robust to mid-stream track changes or upgrades.

TransformStream Use

The use of a `TransformStream` for audio decoding decouples frame ingestion from decode and buffering. This enables modular, testable pipelines and supports future features such as multi-channel, spatial, or multi-track audio decoding, as well as backpressure and adaptive streaming logic.

Backpressure and Dropped Samples

If the ring buffer is full when a decoded frame arrives, excess samples are dropped and a warning is logged. This mechanism ensures that the system always prefers low-latency, real-time playback, in keeping with user experience expectations for live or interactive media.

Error Logging

Any errors encountered during audio decoding are logged with a warning, making it easy to diagnose issues and laying the groundwork for user-facing feedback or automated recovery in future iterations.

Architectural Patterns and Best Practices

Lazy Configuration

Decoders are not configured until sufficient initialization info is available. This lazy approach enables adaptation to dynamic track changes and different codecs, making the playback pipeline robust and future-proof.

Worker Compatibility

Both renderer classes are explicitly designed for execution inside a Web Worker context. This ensures that decoding and rendering workloads do not block or degrade the responsiveness of the user interface, a crucial concern for real-time, interactive applications.

Explicit Resource Management

Frames are explicitly closed after rendering or writing to the ring buffer. Decoders themselves can be closed and reconfigured as needed. This disciplined resource management is vital for browser applications that may run for extended periods and handle multiple streams.

Separation of Concerns

Timeline management, decoding, rendering, and buffering are all handled in composable, well-defined modules. This clean separation not only supports code maintainability and testability but also allows for easy extension—for example, supporting new codecs, adding visual effects, or integrating advanced audio processing modules in the future.

4.5.8 Jitter Buffering and the `Ring` Class (`lib/common/ring.ts`)

A central component enabling smooth, resilient playback in `moq-js` is the ring buffer, implemented as the generic `Ring<T>` class in `lib/common/ring.ts`. The ring buffer addresses the core challenge of decoupling variable-rate packet arrival from deterministic playback requirements. This is vital in object-based media transport over QUIC, where frames or audio samples may arrive with burstiness or jitter due to network fluctuations.

Implementation and Design Rationale

```
// lib/common/ring.ts
export class Ring<T> {
#buffer: Array<T | undefined>
#mask: number
#start = 0
#end = 0
#size = 0

constructor(capacity: number) {
let len = 1
while (len < capacity) len <<= 1
this.#buffer = new Array<T | undefined>(len)
this.#mask = len - 1
}

get length() { return this.#size }

write(item: T): boolean {
if (this.#size === this.#buffer.length) return false
this.#buffer[this.#end] = item
this.#end = (this.#end + 1) & this.#mask
this.#size++
return true
}

read(): T | undefined {
if (this.#size === 0) return undefined
const item = this.#buffer[this.#start]
this.#buffer[this.#start] = undefined
this.#start = (this.#start + 1) & this.#mask
this.#size--
return item
}

clear() {
this.#start = this.#end = this.#size = 0
this.#buffer.fill(undefined)
}
}
```

Design rationale: The buffer is always a power-of-two size, allowing for efficient index wrapping via bitwise operations (e.g., `(index & mask)`), eliminating slow modulo arithmetic. This is especially advantageous in real-time pipelines with high throughput or tight resource constraints.

Private fields (`#buffer`, `#start`, etc.) provide strong encapsulation, preventing accidental corruption by outside code. The bounded design prevents unbounded memory usage, crucial for browser-based or embedded environments.

Motivation and Typical Use Cases

The `Ring` buffer is used throughout `moq-js` as a jitter buffer—most notably in the audio playback pipeline, but also suitable for video and other data streams. It enables the system to:

- **Absorb bursty arrivals**: Incoming frames or samples are written into the buffer as they arrive, regardless of network-induced variability.
- **Serve deterministic playback**: The rendering or audio output logic reads from the buffer at a steady rate, minimizing the impact of network jitter.
- **Control latency and memory**: By bounding the buffer size, latency is kept predictable and the application never over-allocates memory.

Example in pipeline:

```
// lib/playback/webcodecs/audio.ts (excerpt)
const ring = new Ring<AudioData>(bufferSize)
...
const written = ring.write(decodedAudioData)
if (!written) {
// Buffer full: drop frame or trigger underrun logic
}
...
const next = ring.read()
if (next) {
// Schedule for playback via AudioWorkletNode
}
```

Behavior Under Load and Real-Time Constraints

If frames arrive faster than they are consumed and the buffer fills, new arrivals are dropped, favoring low-latency, real-time playback over perfect fidelity. This is especially important for live streaming, where old frames are less valuable than keeping up with the latest data. Conversely, if the buffer empties due to network loss or CPU delays, playback components can detect underflow and respond (e.g., by signaling "buffering" to the user).

Why a Ring Buffer?

Compared to dynamic arrays or lists, a ring buffer:

- Uses no dynamic allocation after construction—crucial for minimizing GC pressure.
- Guarantees O(1) reads and writes, supporting real-time streaming.
- Naturally supports overwriting/dropping oldest or newest data based on policy.

4.6 Codec Integration

A core architectural principle of `moq-js` is codec agnosticism: the library is designed to handle any browser-supported codec without hardwired assumptions about media formats. This design decision is central to its long-term interoperability, resilience to ecosystem changes, and adherence to emerging standards such as the IETF Media over QUIC (MoQ) protocol.

Browser-Native Codec Handling via WebCodecs

At the heart of codec integration in `moq-js` lies the use of the modern [WebCodecs API][31]. WebCodecs exposes a standardized, high-performance interface for encoding and decoding a broad range of audio and video codecs, directly leveraging the browser's native media pipeline and hardware acceleration capabilities. This approach yields several important benefits:

- **Performance:** Hardware-accelerated encode and decode for real-time, low-latency streaming.
- **Compatibility:** Out-of-the-box support for the codecs natively available in the browser, with graceful fallback as the ecosystem evolves.
- **Security:** By avoiding custom JavaScript-based codecs, the attack surface and risk of vulnerabilities are minimized.
- **Maintenance and Portability:** The codebase remains free from legacy codec baggage or third-party binary dependencies.

Supported Codecs and Flexibility

- **Video:** Typical supported codecs include H.264/AVC, AV1, and VP8, with the exact set determined by the browser and underlying operating system. H.264 remains the most widely available, while AV1 and VP8 offer higher efficiency and are increasingly available in modern browsers.

- **Audio:** Opus is the primary codec for real-time interactive audio, due to its low latency and robustness; AAC is supported for compatibility and broad device coverage. Other codecs may be supported as browser implementations expand.

The following excerpt illustrates dynamic configuration of the encoder or decoder at runtime, based on track metadata and available codec information:

```
// Example: VideoDecoder configuration
this.decoder.configure({
codec: track.codec, // e.g., "avc1.42E01E" (H.264), "vp09.00.10.08" (VP9),
↪  "av01.0.04M.08" (AV1)
codedWidth: track.width,
codedHeight: track.height,
description: track.initData, // codec-specific initialization (e.g., avcC box)
});
```

Chapter 5

moq-rs

The **moq-rs** project, primarily authored by Luke Curley, is a performant, modular implementation of a **Media over QUIC (MoQ) relay** written in Rust. The relay system, comprising the `moq-relay` and `moq-transport` crates, embodies advanced architectural principles of protocol layering, asynchronous IO, and scalable media routing. This chapter provides a file-level and function-level breakdown, with code examples, to illuminate the relay's design and the roles of each component.

5.1 moq-rs - Dependencies and Architectural Roles

The `moq-relay` crate directly depends on both `moq-transport` and `moq-api`, as specified in its `Cargo.toml`:

```
[dependencies]
moq-transport = { path = "../moq-transport" }
moq-api = { path = "../moq-api" }
```

Crate	**Purpose**	**Key Dependencies**	**Core Components**
moq-relay	QUIC relay, fanning/caching	moq-transport, quinn, webtransport-quinn, axum, tokio	QUIC server, session, TLS, dev web tools
moq-transport	MoQ protocol, sessions, messages	quinn, webtransport-quinn, tokio, bytes, indexmap	Msg types, session mgmt, protocol engine
moq-api	HTTP API, Redis metadata	axum, hyper, tokio, redis, serde	Server, API endpoints, Redis DB

The architectural roles and relationships between these crates are as follows:

- **`moq-relay`:** This crate implements a relay server responsible for connecting publishing clients to subscribing clients. It manages the deduplication and caching of subscriptions, enabling a single publisher to efficiently serve multiple subscribers. Clients connect using WebTransport over QUIC, specifying a broadcast name (path). The relay orchestrates session management and ensures that subscriptions to the same broadcast are served efficiently.

- **`moq-transport`:** This crate provides the transport protocol implementation over QUIC/WebTransport. It is responsible for the core mechanics of media object transmission, including session establishment, message and stream handling, object fragmentation and reassembly, caching, and both reliable and unreliable delivery. Essentially, it serves as the backbone for data transport within the architecture.

- **`moq-api`:** This crate defines shared API structures, data models, and common logic used across both the relay and other components. It includes reusable types, error definitions, and potentially client/server API logic, ensuring consistency and code reuse throughout the project.

In summary, `moq-relay` serves as the high-level orchestrator, relying on `moq-transport` for protocol and data handling, and on `moq-api` for shared models and API consistency. The dependency relationships are illustrated in Figure 5.1.

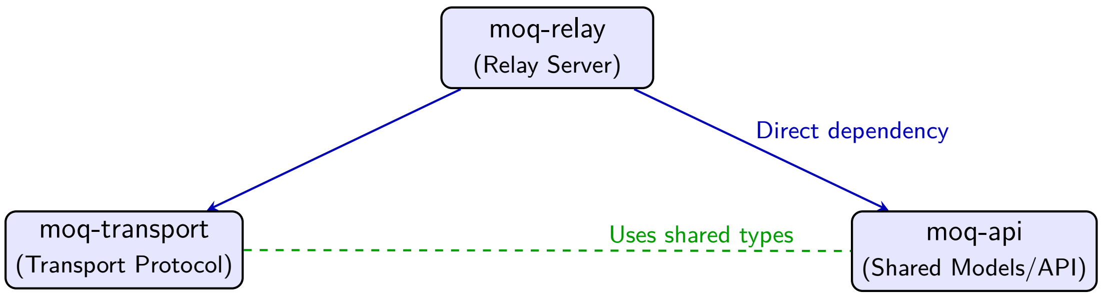


Figure 5.1. Architectural dependencies: `moq-relay` depends on both `moq-transport` and `moq-api`; `moq-transport` also utilizes shared types from `moq-api`.

5.2 moq-relay

moq-relay is a server application that **forwards subscriptions from publishers to subscribers**, performing ***caching*** and ***deduplication*** along the way. It serves as a bridge between media publishers and subscribers, mediating their interactions through QUIC transport and the MoQ application protocol, and it's designed to run in a data center, forwarding media through multiple hops to deduplicate and improve quality of service (QoS).

More specifically:

- It acts as an intermediary between publishers and subscribers; publishers send messages to the relay, while subscribers subscribe to receive these messages.
- It integrates with the QUIC transport layer using the `quinn` crate, enabling low-latency media object transmission over unidirectional QUIC streams.
- It ensures protocol compliance with the IETF Media over QUIC (MoQ) specification by implementing session setup, subscription, and publication workflows via the `moq-transport` crate.

- It employs a form of content caching and temporarily stores messages received from publishers to ensure that any subscribers receive all the requested content, providing replay functionality for late-joining subscribers and resilience against jitter.
- It manages publisher and subscriber lifecycles through asynchronous tasks that coordinate session initialization, teardown, and error recovery in a fault-resilient manner.
- It enables communication across multiple nodes (multi-hop), allowing for scalability and distributed data transmission.
- It distributes incoming media to multiple subscribers using Tokio's bounded broadcast channels, enabling scalable fan-out with automatic backpressure and message dropping for overloaded receivers.
- It allows for the implementation of various QoS policies and automatically manages the priorities of messages to be retransmitted.

5.2.1 moq-relay - Architectural Overview

General architecture

The architecture of moq-rs is deliberately designed to be both **efficient** and **extensible**, and it is composed of several **key components** that collaborate to **manage media session lifecycles**, perform **stream multiplexing**, and **orchestrate content dissemination**.

The relay acts as a **logical hub** where publishers and subscribers are interconnected through shared **namespaces** and **tracks**. Publishers register track names and begin sending **discrete media objects** — each encapsulated within its own unidirectional QUIC stream — to the relay. These objects consist of **transport metadata and payloads**, and are processed by the object ingestion logic within the relay core.

Data structures and caching

Upon arrival, each object is handled by the relay's **track registry**, which maps the incoming track names to an **internal routing structure**. This mapping ensures that the correct subscribers, identified by their subscription queries, receive the corresponding data. The registry maintains a real-time view of the active publishers, subscribers, and the tracks they are associated with, enabling **rapid forwarding decisions** and **dynamic stream switching**.

To support **stream replay** and **late joins**, moq-rs features a **lightweight object cache mechanism**. This in-memory structure stores a limited number of recent media objects per track, making it possible for subscribers who join mid-session to request and receive previously published content. The cache plays a crucial role in providing **resilience** against **short-term network jitter** or **disconnection**, although it is explicitly non-persistent and bounded by memory policies.

Tokio library

Data movement within the relay is coordinated using **asynchronous message channels**, primarily implemented with bounded broadcast channels provided by the **Tokio framework** [@tokio]. These channels **decouple the publishing rate from the subscribing rate**, enabling **concurrent fan-out** of media objects without tightly coupling publisher performance to subscriber responsiveness. **Backpressure** is **enforced naturally** by these channels: when a subscriber cannot keep up with the media rate, **messages are dropped** from its buffer, prompting it to reissue object requests or perform a fresh resubscription.

Control messages management

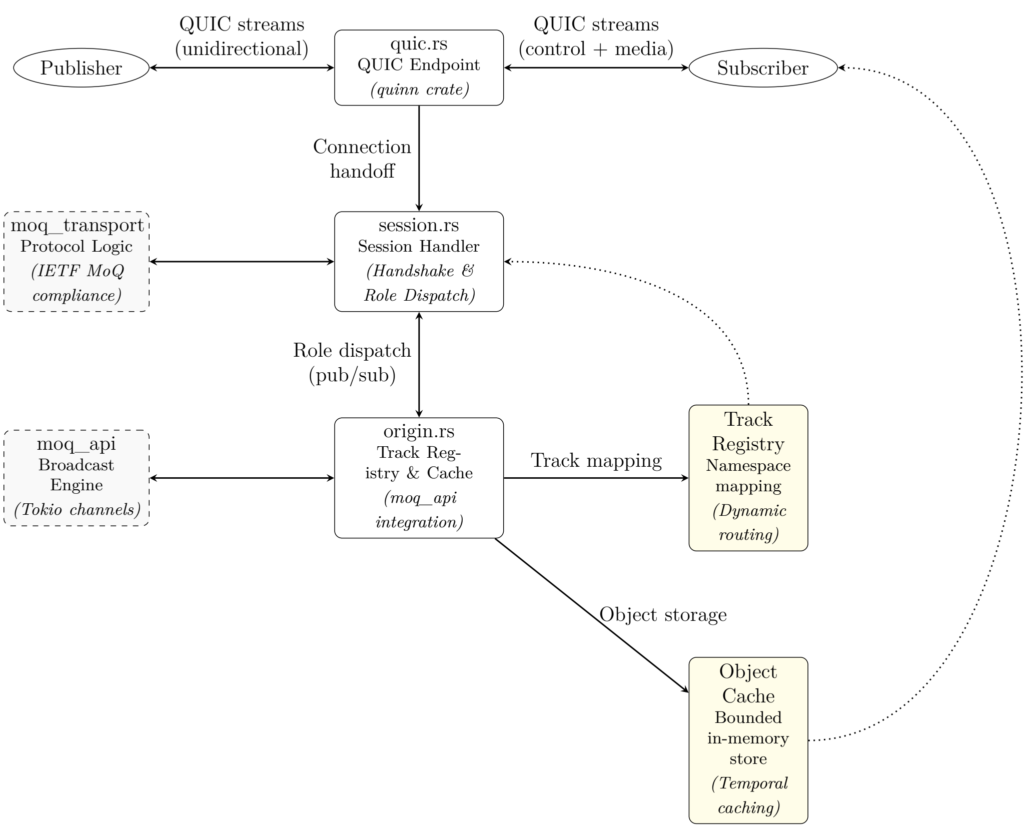
Additionally, moq-rs includes an **internal control protocol engine** that interprets control messages exchanged over the MoQ control stream. These include **session setup**, **track announcements**, **subscribe requests**, and **error conditions**. The **parsing** and **serialization** of these control frames follow the **canonical encoding** defined in the **MoQ protocol draft** and are implemented in the **moq-transport** crate — a shared library reused across moq-rs components.

Relay media semantics agnosticism

An important architectural choice in moq-rs is the relay's statelessness regarding long-term media semantics. The relay is **not aware of codecs, media types, or synchronization across tracks**; its responsibility is **strictly transport-layer delivery of objects** tagged with track and object identifiers. This agnosticism enables moq-rs to support a wide variety of use cases, from low-latency live video to fragmented media file delivery, without requiring changes to the relay logic.

`moq-rs` cleanly separates:

- **Connection management** (QUIC + TLS)
- **Session logic** (handshake + role dispatch)
- **Track & namespace management** (Origin)
- **Media forwarding** (respecting backpressure and optional caching)



5.2.2 moq-relay - Dependency Breakdown

```
[dependencies]
moq-transport = { path = "../moq-transport" }
moq-api = { path = "../moq-api" }
quinn = "0.10"
webtransport-quinn = "0.6"
axum = { version = "0.6", features = ["tokio"] }
tokio = { version = "1", features = ["full"] }
rustls = { version = "0.21", features = ["dangerous_configuration"] }
clap = { version = "4", features = ["derive"] }
# and others for crypto, logging, etc.
```

5.2.3 moq-relay - Main Components

- `main.rs`
  - Parses CLI flags (`config.rs`)
  - Initializes TLS (`tls.rs`) and web dev server (`web.rs`)
  - Calls `quic.serve()` or (`web.serve()`) concurrently
- `quic.rs`

  - Builds a `quinn::Endpoint`
  - Accepts incoming QUIC connections and spawns `Session` tasks

- `session.rs`
  - Performs the MoQ SETUP handshake (via `moq_transport`)
  - Dispatches `serve_publisher` or `serve_subscriber`
  - Each handler invokes `Origin` APIs to publish or subscribe

- `origin.rs`
  - Manages the global track registry and broadcast caches via `moq_api`
  - Handles API calls to create/delete origin entries

- `config.rs`, `tls.rs`, `web.rs`
  - CLI parsing, self-signed cert generation, and optional HTTP dev UI

5.2.4 `main.rs`: Application Entrypoint

Handles server initialization, config, TLS, and concurrent launching of QUIC and optional web services.

```rust
#[tokio::main]
async fn main() -> anyhow::Result<()> {
    env_logger::init();

    // Set up tracing for logs
    let tracer = tracing_subscriber::FmtSubscriber::builder()
        .with_max_level(tracing::Level::WARN)
        .finish();
    tracing::subscriber::set_global_default(tracer).unwrap();

    let config = Config::parse();
    let tls = Tls::load(&config)?;

    // Create a QUIC server for media
    let quic = Quic::new(config.clone(), tls.clone())
        .await
        .context("failed to create server")?;

    // Optional dev HTTP server for fingerprint endpoint
    if config.dev {
        let web = Web::new(config, tls);
        tokio::select! {
            res = quic.serve() => res.context("failed to run quic server"),
            res = web.serve() => res.context("failed to run web server"),
        }
    } else {
        quic.serve().await.context("failed to run quic server")
    }
}
```

Explanation:

- Starts async runtime, logging, and tracing.
- Loads configuration and TLS certificates.
- Instantiates QUIC server for MoQ media relay.
- In dev mode, also starts an HTTP server for certificate fingerprinting (using `axum`).
- The main relay logic is modularized in submodules: `config`, `quic`, `session`, `origin`, `tls`, `web`.

5.2.5 `config.rs`: Configuration Management

The `config.rs` module handles command-line argument parsing and configuration using the `clap` crate. It supports advanced options for certificate/key lists, dev mode, and integration with the MoQ API.

```rust
use std::{net, path};
use url::Url;

use clap::Parser;

/// Search for a pattern in a file and display the lines that contain it.
#[derive(Parser, Clone)]
pub struct Config {
    /// Listen on this address
    #[arg(long, default_value = "[::]:4443")]
    pub listen: net::SocketAddr,

    /// Use the certificates at this path, encoded as PEM.
    /// You can use this option multiple times for multiple certificates.
    /// The first match for the provided SNI will be used, otherwise the last cert
    ↪   will be used.
    /// You also need to provide the private key multiple times via `key`.
    #[arg(long)]
    pub tls_cert: Vec<path::PathBuf>,

    /// Use the private key at this path, encoded as PEM.
    /// There must be a key for every certificate provided via `cert`.
    #[arg(long)]
    pub tls_key: Vec<path::PathBuf>,

    /// Use the TLS root at this path, encoded as PEM.
    /// This value can be provided multiple times for multiple roots.
    /// If this is empty, system roots will be used instead
    #[arg(long)]
    pub tls_root: Vec<path::PathBuf>,

    /// Danger: Disable TLS certificate verification.
    /// Fine for local development and between relays, but should be used in caution
    ↪  in production.
    #[arg(long)]
    pub tls_disable_verify: bool,

    /// Optional: Use the moq-api via HTTP to store origin information.
    #[arg(long)]
    pub api: Option<Url>,

    /// Our internal address which we advertise to other origins.
    /// We use QUIC, so the certificate must be valid for this address.
    /// This needs to be prefixed with https:// to use WebTransport.
    /// This is only used when --api is set and only for publishing broadcasts.
    #[arg(long)]
    pub api_node: Option<Url>,

    /// Enable development mode.
    /// Currently, this only listens on HTTPS and serves /fingerprint, for self-signed
    ↪  certificates
    #[arg(long, action)]
    pub dev: bool,
}
```

Usage in Main Entrypoint

```rust
let config = Config::parse();
// Now config.listen, config.tls_cert, etc. are available
```

Explanation:

- Each certificate/key can be specified multiple times for SNI support.
- Default and required fields are managed by `clap`.
- The struct design makes it clear how to run the relay for different environments.
- The `Config` struct centralizes all startup configuration.
- Each field is annotated for `clap`, providing documentation and default values.
- `Config::parse()` reads arguments from the CLI or environment, supporting flexible deployment.
- Used throughout initialization (e.g., QUIC server, TLS, API client).

5.2.6 `tls.rs`: Certificate Loading/Generation

The `tls.rs` module is responsible for handling TLS certificate and private key loading for the QUIC and development HTTP servers in `moq-relay`. This module ensures secure communication by correctly parsing certificate files, and, in development mode, it can generate or use self-signed certificates for rapid local setup.

```rust
use anyhow::Context;
use ring::digest::{digest, SHA256};
use rustls::server::{ClientHello, ResolvesServerCert};
use rustls::sign::CertifiedKey;
use rustls::{Certificate, PrivateKey, RootCertStore};
use std::io::{self, Cursor, Read};
use std::path;
use std::sync::Arc;
use std::{fs, time};
use webpki::{DnsNameRef, EndEntityCert};

use crate::Config;

#[derive(Clone)]
pub struct Tls {
    pub server: rustls::ServerConfig,
    pub client: rustls::ClientConfig,
    pub fingerprints: Vec<String>,
}

impl Tls {
    pub fn load(config: &Config) -> anyhow::Result<Self> {
        let mut serve = ServeCerts::default();

        // Load the certificate and key files based on their index.
        anyhow::ensure!(
            config.tls_cert.len() == config.tls_key.len(),
            "--tls-cert and --tls-key counts differ"
        );
        for (chain, key) in config.tls_cert.iter().zip(config.tls_key.iter()) {
            serve.load(chain, key)?;
        }

        // Create a list of acceptable root certificates.
        let mut roots = RootCertStore::empty();

        if config.tls_root.is_empty() {
            // Add the platform's native root certificates.
            for cert in rustls_native_certs::load_native_certs().context("could not
            ↪ load platform certs")? {
                roots.add(&Certificate(cert.0)).context("failed to add root cert")?;
            }
        } else {
            // Add the specified root certificates.
            for root in &config.tls_root {
                let root = fs::File::open(root).context("failed to open root cert file")?;
                let mut root = io::BufReader::new(root);
                let root = rustls_pemfile::certs(&mut root).context("failed to read root cert")?;
                anyhow::ensure!(root.len() == 1, "expected a single root cert");
                let root = Certificate(root[0].to_owned());

                roots.add(&root).context("failed to add root cert")?;
            }
        }

        // Create the TLS configuration we'll use as a client (relay -> relay)
        let mut client = rustls::ClientConfig::builder()
            .with_safe_defaults()
            .with_root_certificates(roots)
            .with_no_client_auth();
        // ... (code continues with further setup)
```

Usage in Server Startup

```
let tls = Tls::load(&config)?;
// Pass tls.cert_chain and tls.private_key to Quinn or Axum as needed
```

Explanation:

- Reads the certificate and private key files as specified by the configuration.
- Ensures the number of certs and keys match.
- Uses `rustls_pemfile` to parse both the certificate chain and the PKCS8 private key.
- Assembles the parsed data into a convenient `Tls` struct, ready for use by the QUIC and HTTP servers.
- Provides error handling for common mistakes such as missing or malformed files.
- Loads all server certificates and private keys for SNI.
- Loads system root certs by default, or custom roots if specified.
- For development, the default paths typically point to self-signed certs generated for localhost.
- Builds Rustls client and server configs ready for secure QUIC/WebTransport.

5.2.7 `quic.rs`: QUIC Endpoint Creation

The `quic.rs` module is responsible for instantiating the QUIC endpoint for the relay, configuring both client and server QUIC stacks (for relay-to-relay and client-to-relay traffic), and managing incoming WebTransport connections. It also integrates congestion control and origin management.

Key Responsibilities

- Configure and instantiate the QUIC endpoint for both client and server use.
- Set ALPN for WebTransport.
- Set advanced transport parameters (e.g., BBR congestion control, keep-alive, idle timeout).
- Bind the UDP socket and build a generic endpoint.
- Integrate with the `Origin` and `Session` logic for MoQ relaying.

```rust
use std::{sync::Arc, time};
use anyhow::Context;
use tokio::task::JoinSet;
use crate::{Config, Origin, Session, Tls};

pub struct Quic {
    quic: quinn::Endpoint,
    // The active connections.
    conns: JoinSet<anyhow::Result<()>>,
    // The map of active broadcasts by path.
    origin: Origin,
}

impl Quic {
    // Create a QUIC endpoint that can be used for both clients and servers.
    pub async fn new(config: Config, tls: Tls) -> anyhow::Result<Self> {
        let mut client_config = tls.client.clone();
        let mut server_config = tls.server.clone();
        client_config.alpn_protocols = vec![webtransport_quinn::ALPN.to_vec()];
        server_config.alpn_protocols = vec![webtransport_quinn::ALPN.to_vec()];

        // Enable BBR congestion control
        let mut transport_config = quinn::TransportConfig::default();
        
        ↪  transport_config.max_idle_timeout(Some(time::Duration::from_secs(10).try_into().unwrap()));
        transport_config.keep_alive_interval(Some(time::Duration::from_secs(4)));
        
        ↪  transport_config.congestion_controller_factory(Arc::new(quinn::congestion::BbrConfig::default()))
        transport_config.mtu_discovery_config(None); // Disable MTU discovery
        let transport_config = Arc::new(transport_config);

        let mut client_config = quinn::ClientConfig::new(Arc::new(client_config));
        let mut server_config =
        ↪  quinn::ServerConfig::with_crypto(Arc::new(server_config));
        server_config.transport_config(transport_config.clone());
        client_config.transport_config(transport_config);

        // There's a bit more boilerplate to make a generic endpoint.
        let runtime = quinn::default_runtime().context("no async runtime")?;
        let endpoint_config = quinn::EndpointConfig::default();
        let socket = std::net::UdpSocket::bind(config.listen).context("failed to bind
        ↪  UDP socket")?;

        // Create the generic QUIC endpoint.
        let mut quic = quinn::Endpoint::new(endpoint_config, Some(server_config),
        ↪  socket, runtime)
            .context("failed to create QUIC endpoint")?;
        quic.set_default_client_config(client_config);

        let api = config.api.map(|url| {
            log::info!("using moq-api: url={}", url);
            moq_api::Client::new(url)
        });

        if let Some(ref node) = config.api_node {
            log::info!("advertising origin: url={}", node);
        }

        let origin = Origin::new(api, config.api_node, quic.clone());
        // ...more session spawning logic omitted for brevity...
```

Explanation:

- Clones and modifies TLS client/server configs for WebTransport ALPN.
- Sets advanced transport options: BBR congestion control, keep-alive, and disables MTU discovery for stability.
- Binds a UDP socket to the configured address and builds a `quinn::Endpoint`.
- Attaches MoQ API and origin advertisement for full relay operation.
- Foundation for spawning MoQ sessions on incoming connections (not shown here).

5.2.8 `session.rs`: Session Lifecycle and Role Dispatch

The `session.rs` module is responsible for managing individual MoQ protocol sessions. It handles QUIC connection establishment, WebTransport upgrade, MoQ setup/handshake, and dispatches publisher or subscriber logic according to the client's role.

Key Responsibilities

- Await and accept incoming QUIC connections.
- Perform WebTransport handshake and extract the broadcast path.
- Accept the MoQ protocol setup handshake and determine client role.
- Dispatch session to publisher or subscriber handlers.
- Integrate with origin tracking.

```rust
use anyhow::Context;
use moq_transport::{session::Request, setup::Role, MoqError};
use crate::Origin;

#[derive(Clone)]
pub struct Session {
    origin: Origin,
}

impl Session {
    pub fn new(origin: Origin) -> Self {
        Self { origin }
    }

    pub async fn run(&mut self, conn: quinn::Connecting) -> anyhow::Result<()> {
        log::debug!("received QUIC handshake: ip={:?}", conn.remote_address());

        // Wait for the QUIC connection to be established.
        let conn = conn.await.context("failed to establish QUIC connection")?;

        log::debug!(
            "established QUIC connection: ip={:?} id={}",
            conn.remote_address(),
            conn.stable_id()
        );
        let id = conn.stable_id();

        // Wait for the CONNECT request.
        let request = webtransport_quinn::accept(conn)
            .await
            .context("failed to receive WebTransport request")?;

        // Strip any leading and trailing slashes to get the broadcast name.
        let path = request.url().path().trim_matches('/').to_string();

        log::debug!("received WebTransport CONNECT: id={} path={}", id, path);

        // Accept the CONNECT request.
        let session = request
            .ok()
            .await
            .context("failed to respond to WebTransport request")?;

        // Perform the MoQ handshake.
        let request = moq_transport::session::Server::accept(session)
            .await
            .context("failed to accept handshake")?;

        log::debug!("received MoQ SETUP: id={} role={:?}", id, request.role());

        let role = request.role();

        match role {
            Role::Publisher => {
                if let Err(err) = self.serve_publisher(id, request, &path).await {
                    log::warn!("error serving publisher: id={} path={} err={:#?}", id,
                    ↪  path, err);
                }
            }
            // ...Subscriber and other roles handled similarly...
        }
        Ok(())
    }
}
```

Explanation:

- Handles the entire session lifecycle: QUIC handshake, WebTransport upgrade, MoQ protocol negotiation.
- Extracts the path (broadcast name) from the WebTransport CONNECT request.
- After MoQ SETUP, dispatches the session to publisher or subscriber handlers based on negotiated role.
- Centralizes session logging and error context for robust production debugging.

Publisher handler: Receives media objects, stores them in the cache, and broadcasts to all subscribers.

Subscriber handler: Subscribes the client to a track, replays cached objects, and forwards live objects.

Key mechanisms:

- **Backpressure:** Uses `tokio::sync::broadcast` to fan out data with buffer limits, preventing slow receivers from blocking.
- **Cache Replay:** Late joiners get recent objects for seamless catch-up.
- **Clean teardown:** Properly closes connections and frees resources.

5.2.9 `origin.rs`: Track Registry, Cache, and Routing

The `origin.rs` module manages information about broadcast origins. It handles discovery and registration of broadcasts using the MoQ API, caches active broadcasts, and integrates with the relay's QUIC endpoint for remote origin fetching. This supports multi-relay and distributed topologies.

Key Responsibilities

- Register and discover broadcast origins via the MoQ API.
- Track active broadcasts locally using a cache.
- Provide helpers for publishing new broadcasts and subscribing to existing ones.
- Integrate with the QUIC endpoint for remote origin connections.

```rust
use std::ops::{Deref, DerefMut};
use std::{
    collections::HashMap,
    sync::{Arc, Mutex, Weak},
};

use moq_api::ApiError;
use moq_transport::cache::{broadcast, CacheError};
use url::Url;
use tokio::time;
use crate::RelayError;

#[derive(Clone)]
pub struct Origin {
    // An API client used to get/set broadcasts.
    // If None then we never use a remote origin.
    api: Option<moq_api::Client>,

    // The internal address of our node.
    // If None then we can never advertise ourselves as an origin.
    node: Option<Url>,

    // A map of active broadcasts by ID.
    cache: Arc<Mutex<HashMap<String, Weak<Subscriber>>>>,

    // A QUIC endpoint we'll use to fetch from other origins.
    quic: quinn::Endpoint,
}

impl Origin {
    pub fn new(api: Option<moq_api::Client>, node: Option<Url>, quic: quinn::Endpoint)
    ↪  -> Self {
        Self {
            api,
            node,
            cache: Default::default(),
            quic,
        }
    }

    /// Create a new broadcast with the given ID.
    /// Publisher::run needs to be called to periodically refresh the origin cache.
    pub async fn publish(&mut self, id: &str) -> Result<Publisher, RelayError> {
        let (publisher, subscriber) = broadcast::new(id);

        let subscriber = {
            let mut cache = self.cache.lock().unwrap();

            // Check if the broadcast already exists.
            // TODO This is racey, because a new publisher could be created while
            ↪  existing subscribers are still active.
            if cache.contains_key(id) {
                return Err(CacheError::Duplicate.into());
            }

            // Create subscriber that will remove from the cache when dropped.
            let subscriber = Arc::new(Subscriber {
                // ... fields omitted for brevity ...
            });
            // ...more logic...
        };
        // ...more logic...
    }
    // ...more methods...
}
```

Explanation:

- Holds the MoQ API client, node URL, and a thread-safe cache of current broadcasts.
- Integrates with QUIC for fetching remote origins.
- The `publish` method creates and registers a new broadcast, ensuring no duplicates.
- Uses `Arc<Mutex<HashMap<...»>` for safe concurrent access to the broadcast cache.
- Designed for multi-relay environments and easy extension.

Key APIs:

- `register_publisher`: Associates a publisher with one or more tracks.
- `subscribe`: Adds a subscriber to a track, replays cached objects, and pipes live updates.
- `publish_object`: Inserts media into cache and pushes it to subscribers.

5.2.10 `web.rs` and `error.rs`: Optional Web UI and Errors

- `web.rs`: Exposes an HTTP dashboard for debugging, live session stats, and health checks.
- `error.rs`: Defines a relay-wide error enum for uniform error propagation.

5.3 moq-relay - Publisher–Subscriber Data and Object Flow

The core of the MoQ transport involves the structured delivery of ***media objects***—discrete, self-contained units such as video keyframes, audio segments, or subtitles. **`moq-relay`** handles these flows symmetrically, connecting publishers and subscribers through a shared namespace registry and optional cache layer.

Publisher Flow

1. Publisher connects via **QUIC** and performs the **MoQ `SETUP`** handshake.
2. It registers one or more `TrackName`s using `PublisherTrack` requests.
3. For each media object:
   - A **QUIC unidirectional stream** is opened.
   - **Headers** and **payload** are sent as a single flow.
   - The relay **stores the object** (if cache enabled) and **fans it out to active subscribers**.

Subscriber Flow

1. Subscriber **connects** and performs the **`SETUP`** handshake.
2. It subscribes to a given `TrackName`.
3. The relay:
   - **Sends** any **cached media objects** immediately.
   - **Pipes** all **live objects** from the publisher to the subscriber as they arrive.
   - **Applies backpressure** using broadcast channels and buffer limits.

5.4 moq-transport

The `moq-transport` crate is the **protocol engine** of the Media over QUIC (MoQ) ecosystem. It implements all protocol logic and primitives needed for the Media over QUIC (MoQ) ecosystem, and it is responsible for all wire protocol logic, including message encoding/decoding, session state machines, and role-specific behavior for publishers and subscribers. It is a pure Rust library designed to be reusable across clients, relays, and test tools. Its main responsibilities are:

- Session state management for publishers and subscribers
- Protocol-compliant encoding/decoding of all MoQ control and data messages
- In-memory hierarchical caching and object replay logic
- Typed error handling and protocol negotiation

5.5 Architectural Role of `moq-transport`

Within the modular design of the `moq-rs` project, the `moq-transport` crate plays a foundational role as the core transport layer upon which higher-level components, such as `moq-relay`, are built. The relationship between `moq-transport` and `moq-relay` is characterized by a clear separation of concerns, where each crate is responsible for a distinct layer of functionality within the overall system architecture.

`moq-transport` is responsible for implementing the Media over QUIC (MoQ) protocol, encapsulating all essential aspects of data transport over QUIC and WebTransport. Its core responsibilities include:

- **Session Management:** Establishing, maintaining, and terminating sessions between endpoints, supporting both publishers and subscribers.
- **Message Framing and Parsing:** Defining and handling protocol messages, including serialization, deserialization, and adherence to protocol specifications.
- **Media Object Transmission:** Handling the fragmentation, reassembly, and sequencing of media objects to ensure reliable and efficient delivery.
- **Caching and Deduplication:** Providing mechanisms for caching media data and eliminating redundant transmissions to optimize network usage.
- **Error Handling:** Managing protocol-level errors and providing robust error reporting to higher layers.

Building on this transport layer, the `moq-relay` crate serves as a high-level server component that connects publishing clients with subscribing clients. Its main responsibilities include orchestrating publisher-subscriber relationships, routing data streams, enforcing access policies, and implementing broadcast session management. `moq-relay` relies extensively on `moq-transport` to offload all protocol-specific logic, allowing it to focus on application-level concerns.

Layer/Crate	**Main Responsibility**
`moq-transport`	Protocol logic, session state machines, message encoding/decoding, cache logic, role transitions
`moq-relay`	Orchestration, connection and session management, registry, routing and policy
`moq-api`	Federation, discovery, registry for origins/tracks, cross-node coordination

Table 5.1. Role of `moq-transport` within the MoQ relay architecture.

5.5.1 moq-transport - Dependency Breakdown

```
[dependencies]
quinn = "0.10"
webtransport-quinn = "0.6"
tokio = { version = "1", features = ["macros", "io-util", "sync"] }
bytes = "1"
indexmap = "2"
thiserror = "1"
async-trait = "0.1"
paste = "1"
```

5.5.2 moq-transport - Module Structure

The crate is organized into the following submodules:

- `session/` — MoQ session state machines for different endpoint roles
- `message/` — Each control/data message with its encoding/decoding logic
- `setup/` — Protocol handshake, version, and role negotiation
- `cache/` — Hierarchical, per-track object caching and broadcasting
- `coding/` — Low-level serialization/deserialization utilities (varints, strings, params)
- `error.rs` — Unified protocol and library error types
- `lib.rs` — Crate root, re-exports all main components

5.5.3 Session State Machines (`session/`)

Session management is central to protocol compliance and reliability. Each file under `session/` implements a key state machine:

- `client.rs`, `server.rs`: Accepts connections, negotiates setup, and dispatches to role handlers
- `publisher.rs`, `subscriber.rs`: Implements the flows for publisher and subscriber endpoints
- `control.rs`: Encapsulates control stream parsing and writing (e.g., handling MoQ control frames)

Code sample: Handshake State Machine (from `server.rs`) The following code, from `server.rs`, succinctly exemplifies the handshake state machine. It shows the server-side logic for accepting a new session, reading the SETUP frame, and initializing a `Request` object that binds together the session's IO and negotiated parameters:

```rust
pub struct Server<S> {
    io: S,
}

impl<S: AsyncRead + AsyncWrite + Unpin> Server<S> {
    /// Accept the handshake from the client
    pub async fn accept(mut io: S) -> Result<Request, MoqError> {
        // Read and parse SETUP message
        let setup = Setup::decode(&mut io).await?;
        Ok(Request { io, setup })
    }
}
```

In this construct, the server not only validates the handshake, but also sets the stage for all subsequent session behavior, handing off the validated state to appropriate session logic for either publication or subscription. The strong typing ensures protocol correctness and makes errors in negotiation or transport instantly actionable.

5.5.4 Session Negotiation Example

Negotiation of protocol parameters is the cornerstone of establishing a compliant MoQ session. The handshake is unidirectional in its initiation: the client sends a SETUP frame specifying its supported protocol version, desired role, and optional extensions, after which it awaits a SETUP_OK response from the server. The following sample, distilled from `setup/client.rs`, demonstrates the essential sequence:

```rust
// setup/client.rs (simplified)
pub async fn setup_handshake(conn: &mut QuicConn) -> Result<Setup> {
    // Send SETUP frame with version/role info
    let setup = Setup { version, role, extensions };
    setup.encode(&mut conn).await?;
    // Await SETUP_OK from the server
    let resp = SetupOk::decode(&mut conn).await?;
    Ok(resp)
}
```

Here, the handshake's structure is not merely a technicality but a formal contract: both sides must agree on all relevant parameters before proceeding to any media transfer. This approach ensures that protocol upgrades, extensions, and role assignments are explicit and verifiable, supporting extensibility and future-proofing.

5.5.5 Message Encoding and Decoding (`message/`)

At the heart of any transport protocol lies its message encoding. The `message/` submodule of `moq-transport` provides a highly-structured, strongly-typed definition for every control and data message that can be sent across a session. Each message—be it for

stream announcements, subscriptions, segment transmission, or control—is defined as a Rust struct or enum with precise encode and decode methods, allowing for round-trip correctness and preventing malformed wire formats.
This modularization is apparent from the module declaration itself (`message/mod.rs`):

```rust
//! Low-level message sent over the wire, as defined in the specification.
//!
//! All of these messages are sent over a bidirectional QUIC stream.
//! This introduces some head-of-line blocking but provides ordering guarantees.

mod data;
mod frame;
mod id;
mod publish;
mod request;
mod subscribe;
mod util;

pub use data::*;
pub use frame::*;
pub use id::*;
pub use publish::*;
pub use request::*;
pub use subscribe::*;
pub use util::*;
```

Code sample: Message Types and Encoding (from `frame.rs`)

```rust
pub enum Frame {
    Publish(Publish),
    Subscribe(Subscribe),
    Data(Data),
    // ...
}

impl Frame {
    pub fn encode<B: BufMut>(&self, buf: &mut B) { /* ... */ }
    pub fn decode<B: Buf>(buf: &mut B) -> Result<Self, Error> { /* ... */ }
}
```

Central to this system is the `Frame` enum, representing all possible top-level messages. Its encode and decode methods handle precise, protocol-compliant binary serialization:

```rust
pub enum Frame {
    Publish(Publish),
    Subscribe(Subscribe),
    Data(Data),
    // ...
}

impl Frame {
    pub fn encode<B: BufMut>(&self, buf: &mut B) { /* ... */ }
    pub fn decode<B: Buf>(buf: &mut B) -> Result<Self, Error> { /* ... */ }
}
```

This approach yields several critical benefits: first, the strong typing of wire messages ensures that all frames are parsed and validated before reaching higher layers, minimizing error surfaces. Second, the separation into individual modules allows independent evolution and exhaustive testing of each message type, a necessity for robust protocol evolution.

The design philosophy here is explicit: by making all wire messages strongly typed and individually validated, the crate achieves both high reliability and extensibility. Any decoding or protocol error is surfaced as a distinct error case, enabling graceful recovery or reporting at higher layers.

5.5.6 Wire Serialization Utilities (`coding/`)

The correct serialization of protocol primitives is a non-negotiable requirement for interoperability in any transport protocol. The `coding/` module in `moq-transport` implements this via a clean separation of encode and decode routines for the various primitives—variable-length integers, parameter lists, and protocol strings.

This module defines MoQ wire primitives, such as variable-length integers (used extensively in QUIC and MoQ), string types, and message framing:

```rust
mod decode;
mod encode;
mod params;
mod string;
mod varint;

pub use decode::*;
pub use encode::*;
pub use params::*;
pub use string::*;
pub use varint::*;
```

Code sample: Variable-Length Integer Codec (from `varint.rs`)

One of the most significant utilities is the QUIC-style variable-length integer codec. The importance of varint encoding cannot be overstated: it is both space-efficient and a requirement of the MoQ and QUIC specifications. The struct and its core API are as follows:

```rust
pub struct VarInt(pub u64);

impl VarInt {
    pub fn decode<B: Buf>(buf: &mut B) -> Result<Self, Error> { /* ... */ }
    pub fn encode<B: BufMut>(&self, buf: &mut B) { /* ... */ }
}
```

Explanation:

- Encapsulates all protocol serialization/deserialization, such as QUIC-style varints, strings, and parameters.
- Cleanly separated encode/decode modules for maintainability.

The actual implementation in `varint.rs` carefully follows the QUIC variable-length integer encoding algorithm, which changes encoding width dynamically to minimize space for common values while accommodating the protocol's wide numeric ranges:

```rust
// coding/varint.rs
pub fn encode_varint(val: u64, buf: &mut BytesMut) {
    if val < 0x40 {
        buf.put_u8(val as u8);
    } else if val < 0x4000 {
        // ...
    }
    // (implements the QUIC variable-length integer encoding)
}
```

This code ensures both correctness and efficiency, laying the foundation for every higher-level protocol feature, from stream IDs to object lengths.

5.5.7 Hierarchical Object Caching (`cache/`)

Caching within MoQ is designed not as a mere optimization, but as a core architectural pillar enabling reliable, low-latency media delivery. The `cache/` module implements a hierarchical object cache that mirrors the structure of live media: broadcasts contain tracks, tracks contain segments, segments contain fragments, and fragments encapsulate media objects.
The cache subsystem is composed of:

- `track.rs`: Per-track object cache
- `fragment.rs`, `segment.rs`: Sub-division of objects for efficient lookup
- `broadcast.rs`: Tokio-broadcast based fan-out to subscribers
- `watch.rs`: Mechanism for subscribers to rejoin and replay from a specific point

```rust
//! Allows a publisher to push updates, automatically caching and fanning it out to
↪  any subscribers.
//!
//! The hierarchy is: [broadcast] -> [track] -> [segment] -> [fragment] ->
↪  [Bytes](bytes::Bytes)
mod broadcast;
mod fragment;
mod segment;
mod subscriber;
mod track;

pub use broadcast::*;
pub use fragment::*;
pub use segment::*;
pub use subscriber::*;
pub use track::*;
```

Code sample: Track Cache (from `track.rs`)

The `TrackCache` structure, for example, manages all fragments for a particular track. Its methods allow insertion of new objects and efficient replay to clients resuming from arbitrary sequence numbers—an essential feature for scalable streaming and late joiners:

```rust
// cache/track.rs
pub struct TrackCache {
    fragments: VecDeque<Fragment>,
    max_size: usize,
}
impl TrackCache {
    pub fn insert(&mut self, object: Object) {
        // Insert into the latest fragment, evict old objects if exceeding max_size
    }
    pub fn replay_from(&self, seq: u64) -> Vec<Object> {
        // Return objects with sequence >= seq
    }
}
```

Code sample: Broadcast Structure (from `broadcast.rs`)

The broader broadcast cache structure also enforces uniqueness and organizes tracks under their broadcast IDs:

```rust
pub struct Broadcast {
    id: String,
    tracks: HashMap<String, Track>,
    // ...
}

```

```
impl Broadcast {
    pub fn new(id: String) -> Self { /* ... */ }
    pub fn push_track(&mut self, track: Track) { /* ... */ }
    pub fn get_track(&self, name: &str) -> Option<&Track> { /* ... */ }
}
```

Explanation:

- Caching is hierarchical, following the protocol's media structure (broadcast → track → segment → fragment).
- Each layer is a Rust struct with methods for adding/querying children.
- Essential for deduplication, replay, and high-performance pub/sub.

Caching is hierarchical, following the protocol's media structure (broadcast → track → segment → fragment).
Each layer is a Rust struct with methods for adding/querying children.
Through these data structures, the protocol achieves both deduplication and high-performance pub/sub, which are indispensable for real-world streaming systems.

5.5.8 Protocol Negotiation (setup/)

The protocol negotiation process ensures that both endpoints in a session have agreed upon compatible roles, versions, and protocol extensions before any actual media or control messages flow. The `setup/` module embodies this handshake logic, defining all messages and types necessary for robust negotiation.

As reflected in the module structure:

```
//! Messages used for the MoQ Transport handshake.
//!
//! After establishing the WebTransport session, the client creates a bidirectional
↪  QUIC stream.
//! The client sends the [Client] message and then waits for the [Server] message.
mod client;
mod extensions;
mod negotiate;
mod role;
mod server;
mod version;

pub use client::*;
pub use extensions::*;
pub use negotiate::*;
pub use role::*;
pub use server::*;
pub use version::*;
```

Code sample: Setup Handshake (from `client.rs`)

A handshake message, such as the `Client` struct, encodes both the desired version and endpoint role, providing encode and decode methods for robust serialization:

```rust
pub struct Client {
    pub version: u32,
    pub role: Role,
    // ...
}

impl Client {
    pub fn encode<B: BufMut>(&self, buf: &mut B) { /* ... */ }
    pub fn decode<B: Buf>(buf: &mut B) -> Result<Self, Error> { /* ... */ }
}
```

Explanation:

- Defines all handshake and negotiation messages for MoQ sessions.
- Roles, extensions, and versions are negotiated at the start of every session.

By making the negotiation phase explicit, the protocol not only facilitates graceful upgrades and extensibility, but also provides for immediate and clear error reporting if endpoints are incompatible.

5.5.9 Typed Error Handling (`error.rs`)

Errors within a transport protocol are not merely accidental byproducts but must be intentionally designed, so that endpoints can recover or report failures with precision. The error handling module in `moq-transport` centers around a trait that every protocol error implements:

```rust
pub trait MoqError {
    /// An integer code that is sent over the wire.
    fn code(&self) -> u32;

    /// An optional reason sometimes sent over the wire.
    fn reason(&self) -> String;
}
```

This design makes it possible to propagate errors as protocol events, with both machine-readable codes and optional human-readable reasons. This enables not just robust internal error handling, but also rich diagnostics for developers and operators observing the protocol in production.

5.5.10 Crate Root (`lib.rs`)

The crate root `lib.rs` exposes all public modules and re-exports the most important protocol types (`VarInt`, `MoqError`). Internal modules like `coding` and `error` are kept private except for key types.

```rust
//! An implementation of the MoQ Transport protocol.
//!
//! MoQ Transport is a pub/sub protocol over QUIC.
//! While originally designed for live media, MoQ Transport is generic and can be used for other live applications.
//! The specification is a work in progress and will change.
//! See the [specification](https://datatracker.ietf.org/doc/draft-ietf-moq-transport/) and [github](https://github.com/moq-wg/moq-transport) for any updates.
//!
//! This implementation has some required extensions until the draft stablizes. See: [Extensions](crate::setup::Extensions)
mod coding;
mod error;

pub mod cache;
pub mod message;
pub mod session;
pub mod setup;

pub use coding::VarInt;
pub use error::MoqError;
```

This pattern of careful API exposure, combined with comprehensive module documentation, is emblematic of robust Rust library design and supports both immediate adoption and long-term maintainability.

5.6 moq-api

The `moq-api` crate acts as a discovery, registry, and metadata service for Media over QUIC relays. It exposes an HTTP API (via Axum) for registering and discovering broadcast origins, backed by a Redis store. It also provides a client library for interacting with this API, enabling relay federation, cross-node coordination, and external service orchestration.

5.6.1 Architectural Role of `moq-api`

Within the `moq-rs` architecture, the `moq-api` crate serves as a foundational library that defines the shared data models, error types, and utility functions used throughout the system. Its primary purpose is to provide a single source of truth for protocol-level abstractions, ensuring consistency and interoperability between higher-level components such as `moq-relay` and lower-level components such as `moq-transport`.

The architectural responsibilities of `moq-api` include:

- **Shared Data Models:** `moq-api` encapsulates the common types representing protocol entities, messages, and states, which are used uniformly across both `moq-relay` and `moq-transport`.
- **Standardized Error Handling:** It defines error types and handling mechanisms that facilitate robust and consistent error reporting throughout the system.
- **Reusable Logic and Utilities:** The crate may implement shared utility functions or traits required by multiple components, avoiding code duplication and fostering maintainability.

In relation to the other main crates:

- `moq-relay` relies on `moq-api` for defining and handling the protocol messages, client and server models, and error types, enabling it to focus on application-level orchestration and routing logic.
- `moq-transport` utilizes the models and types from `moq-api` to ensure protocol compliance and to guarantee that its internal mechanisms for transport, message framing, and session management remain compatible with other parts of the system.
- By depending on `moq-api`, both `moq-relay` and `moq-transport` maintain a clean separation of concerns, allowing protocol data structures and error handling to evolve independently from the business logic and transport mechanisms.

5.6.2 Module Structure

The organization of the crate reflects its dual nature as both a server and a library:

- `model.rs`: Data models (`Origin`) used in the API.
- `client.rs`: HTTP client for querying and setting origins via REST.
- `server.rs`: Axum-based web server that exposes API endpoints.
- `error.rs`: Unified API error types, covering HTTP, Redis, and URL errors.
- `lib.rs`: Crate entry point, re-exporting major types and modules.
- `main.rs`: Runs the HTTP server (example or real deployment).

5.6.3 Data Models (`model.rs`)

At the core of `moq-api`'s schema is the `Origin` struct, which acts as a persistent and portable representation of a broadcast relay or source endpoint. Defined to be compatible with Serde's powerful serialization framework, the `Origin` model is engineered for seamless translation between Rust types, JSON payloads, and Redis records. By modeling origin endpoints with a `url::Url`, the system ensures robust validation and normalization of all registered addresses, which is vital for the security and correctness of distributed relay discovery.

```
use serde::{Deserialize, Serialize};

use url::Url;

#[derive(Serialize, Deserialize, PartialEq, Eq)]
pub struct Origin {
        pub url: Url,
}
```

The ‘Origin‘ struct is designed for direct mapping to and from JSON via Serde, representing the URL of an origin relay in the API’s schema. This struct, though minimal, encapsulates a foundational protocol concept: it abstracts away the specifics of a relay’s network address, providing a uniform, type-safe, and interoperable handle for all registry and discovery operations.

5.6.4 Client Abstraction (`client.rs`)

Beyond the server implementation, `moq-api` empowers external consumers and internal services alike to interact with the registry via an idiomatic async Rust client. This client exposes a full suite of CRUD (Create, Read, Update, Delete) operations for origin resources, mapped directly onto HTTP methods and JSON payloads. Leveraging the widely-used `reqwest` and `url` crates, the client not only abstracts the mechanics of HTTP transport, but also unifies error handling under the `ApiError` type, simplifying integration into larger async workflows.

The following excerpt exemplifies the client’s interface, revealing both its ergonomic construction and its commitment to robust error propagation:

```
use url::Url;

use crate::{ApiError, Origin};

#[derive(Clone)]
pub struct Client {
        // The address of the moq-api server
        url: Url,

        client: reqwest::Client,
}

impl Client {
        pub fn new(url: Url) -> Self {
                let client = reqwest::Client::new();
                Self { url, client }
        }

        pub async fn get_origin(&self, id: &str) -> Result<Option<Origin>, ApiError> {
                let url = self.url.join("origin/")?.join(id)?;
```

```
            let resp = self.client.get(url).send().await?;
            if resp.status() == reqwest::StatusCode::NOT_FOUND {
                return Ok(None);
            }

            let origin: Origin = resp.json().await?;
            Ok(Some(origin))
        }

        pub async fn set_origin(&mut self, id: &str, origin: &Origin) -> Result<(),
        ↪  ApiError> {
            let url = self.url.join("origin/")?.join(id)?;

            let resp = self.client.post(url).json(origin).send().await?;
            resp.error_for_status()?;

            Ok(())
        }

        pub async fn delete_origin(&mut self, id: &str) -> Result<(), ApiError> {
            let url = self.url.join("origin/")?.join(id)?;

            let resp = self.client.delete(url).send().await?;
            resp.error_for_status()?;

            Ok(())
        }

        pub async fn patch_origin(&mut self, id: &str, origin: &Origin) -> Result<(),
        ↪  ApiError> {
            let url = self.url.join("origin/")?.join(id)?;

            let resp = self.client.patch(url).json(origin).send().await?;
            resp.error_for_status()?;

            Ok(())
        }
}
```

This design ensures that any component—be it another relay node, an operator dashboard, or a monitoring agent—can programmatically discover, update, or decommission origin metadata with uniform reliability, irrespective of the underlying network conditions or server implementation details.

5.6.5 Error Handling (`error.rs`)

Robust error handling is a cornerstone of distributed system reliability, and `moq-api` addresses this through the definition of a single, extensible `ApiError` enum. Harnessing the expressive power of the `thiserror` crate, the error type unifies disparate failure modes—arising from Redis, HTTP, or URL parsing—under a coherent interface. This not only enables straightforward propagation and reporting but also ensures that every

API operation, whether in client or server context, can be instrumented with precise, actionable diagnostics.

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum ApiError {
        #[error("redis error: {0}")]
        Redis(#[from] redis::RedisError),

        #[error("reqwest error: {0}")]
        Request(#[from] reqwest::Error),

        #[error("hyper error: {0}")]
        Hyper(#[from] hyper::Error),

        #[error("url error: {0}")]
        Url(#[from] url::ParseError),
}
```

This error enum provides automatic conversion from Redis, Reqwest, Hyper, and URL parsing errors, streamlining error handling across the API's internal logic and HTTP client/server boundaries.
By supporting automatic conversion from all major sources of failure, `ApiError` streamlines code at every level, from endpoint handlers to background workers, thus elevating both reliability and developer productivity.

5.6.6 Server Implementation (`server.rs`)

The server logic of `moq-api` is realized using the Axum web framework, chosen for its high composability and async-first design. At startup, the server reads its configuration—including HTTP bind address and Redis backend connection—from strongly typed structures, supporting both static deployment and dynamic orchestration in containerized environments.

The core of the HTTP API exposes RESTful endpoints for CRUD operations on origins. Each handler is carefully instrumented for error handling, state propagation, and integration with the Redis backend, supporting not only real-time updates but also resilience in the face of network partitions or service restarts.

```rust
use std::net;

use axum::{
        extract::{Path, State},
        http::StatusCode,
        response::{IntoResponse, Response},
        routing::get,
        Json, Router,
};

use clap::Parser;

use redis::{aio::ConnectionManager, AsyncCommands};

use moq_api::{ApiError, Origin};

/// Runs a HTTP API to create/get origins for broadcasts.
#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
pub struct ServerConfig {
        /// Listen for HTTP requests on the given address
        #[arg(long)]
        pub listen: net::SocketAddr,

        /// Connect to the given redis instance
        #[arg(long)]
        pub redis: url::Url,
}

pub struct Server {
        config: ServerConfig,
}

impl Server {
        pub fn new(config: ServerConfig) -> Self {
                Self { config }
        }

        pub async fn run(self) -> Result<(), ApiError> {
                log::info!("connecting to redis: url={}", self.config.redis);

                // Create the redis client.
                let redis = redis::Client::open(self.config.redis)?;
                let redis = redis
                        .get_tokio_connection_manager() // TODO
                        ↪  get_tokio_connection_manager_with_backoff?
                        .await?;

                let app = Router::new()
                        .route(
                                "/origin/:id",
                                get(get_origin)
                                        .post(set_origin)
                                        .delete(delete_origin)
                                        .patch(patch_origin),
                        )
                        .with_state(redis);

                log::info!("serving requests: bind={}", self.config.listen);

                axum::Server::bind(&self.config.listen)
```

This code configures an Axum web server, defines routes for CRUD operations on broadcast origins, connects to a Redis backend, and exposes the ‘ServerConfig‘ for flexible deployment.

The separation of configuration, runtime logic, and handler implementation ensures that the server can be easily extended to support additional metadata types, authorization layers, or alternative storage backends as protocol or operational requirements evolve.

5.6.7 Crate Integration (`lib.rs`)

The crate’s root (`lib.rs`) is structured to provide a clean, minimal surface for both binary and library use cases. By selectively re-exporting the client, error, and data model modules, `moq-api` ensures that downstream crates and services have immediate access to all core abstractions without unnecessary coupling to internal implementation details.

```rust
mod client;
mod error;
mod model;

pub use client::*;
pub use error::*;
pub use model::*;
```

This structure provides a clean and minimal top-level API for library users, encapsulating the client interface, unified error type, and origin data model.
This approach, standard in modern Rust library design, supports robust encapsulation and enables seamless upgrades and evolution of the internal API surface without breaking consumers.

Chapter 6

Media Over Quic Transport – Datagram Implementation

6.1 Transmission modes

QUIC provides two main modes for data transmission:

- **Stream:**
  - An ordered, lossless data flow (bytes) that can be either unidirectional or bidirectional, designed to add minimal overhead.
  - QUIC allows an arbitrary number of streams to operate simultaneously and send an arbitrary amount of data over any stream.
  - It enables multiplexing capabilities between different streams.
- **Datagram:**
  - Independent packets of bytes transmitted with potential losses.
  - Even lower overhead compared to streams.
  - Retains the congestion control and encryption mechanisms used by QUIC but without guarantees on order or data loss.

6.2 Datagram mode

Transmission of unreliable data via QUIC can be useful in specific use cases, such as:

- Applications that intend to use both reliable streams and an unreliable data flow to the same peers can reuse the same handshake and authentication context, reducing the expected latency compared to using separate TLS and DTLS connections simultaneously.

- Leverage a more sophisticated loss recovery system compared to the DTLS handshake, allowing for more immediate recovery.
- Providing a single congestion control system for both reliable and unreliable data can be more efficient and effective.
- Facilitate the transition to the QUIC stack for existing applications that use unreliable data flows by taking advantage of QUIC's handshake, authentication, and congestion control systems.
- Implementation of IP packet tunneling with a single QUIC connection for control and data transmission.

6.2.1 Datagram mode - custom implementation in moq-js and moq-rs

In order to implement a datagram transmission mode in **moq-js** and **moq-rs**, modifications were made to both applications, and a method was designed to organize the MoQT segment data into datagrams.

More specifically, the stream data is extracted in chunks of variable size and packaged, including header bytes to identify their association within the MoQ context, as well as additional bytes that allow the chunks to be divided into slices with sizes compatible with the maximum size allowed for PDUs included in QUIC datagrams.

The header has the following format

TrackId	GroupId	ObjectId	Slice Number	Slice length

Table 6.1. Datagram slice header

When all the slices of a chunk have been sent, a chunk end datagram is transmitted in the following format

TrackId	GroupId	ObjectId	Data length (bytes)	"end_chunk"

Table 6.2. Chunk end datagram

Datagram mode - moq-js

Inside moq-js, data transmission through datagrams has been added within the file *object.ts*, in the functions *send()*, *recv()*, and with the creation of the async function *receiveDatagrams()*.

Within the *send()* function, the header of the MOQ object is sent via stream, while the other outbound stream data is intercepted through the ***transform()*** function of a *TransformStream*, which divides each chunk of the stream into slices of a maximum size of 1024 bytes, adds a header identifying the chunk and the corresponding slice, and sends it in datagram mode using the ***sendDatagram()*** function, which writes to a Writer exposed by the QUIC library.

```
async sendDatagram(datagram: Uint8Array) {
 if (!this.writer) this.writer = this.quic.datagrams.writable.getWriter()
      // get quic instance datagrams writer datagrams writer
 await this.writer.write(datagram) // send datagram to quic
}
```

Within the *recv()* function, once the header is received via QUIC stream, the data structures involved in the reception of datagrams by the function ***receiveDatagrams()*** are initialized if necessary, and a dedicated *ReadableStream* is created, which will wait for data in the queue of the group corresponding to the header.

```
   async receiveDatagrams() {
  if (this.datagramMode) {
   const readable = this.quic.datagrams.readable // incoming datagrams as readable
   ↪  stream

   const reader = readable.getReader() // stream reader
   console.log("Datagram reception active")

   for (;;) {
    const { value, done } = await reader.read() // read from stream

    if (value) {
     const res = value as Uint8Array
     const utf = new TextDecoder().decode(res)
     const splitData = utf.split(" ") // split object fields

     if (splitData.length > 5) {
      let header = ""// recompose header string
      for (let i = 0; i < 5; i++) header += splitData[i] + " "
      const offset = new TextEncoder().encode(header).length // get data start offset
      ↪  from header length
      const trackId = Number(splitData.shift()).toString() // decode track id
```

```
        const groupId = Number(splitData.shift()) // decode group id number
        const sequenceNum = Number(splitData.shift()) // decode object sequence number
        const sliceNum = Number(splitData.shift()) // decode slice number
        const sliceLen = Number(splitData.shift()) // decode slice number
        const data = res.subarray(offset, res.length) // extract data
        // console.log(data)
        // console.log(sliceLen, offset)
        await this.mutex.acquire() // start atomic operation on chunksMap
        let track = this.chunksMap.get(trackId) // get track chunks map
        if (!track) {
         track = new Map<number, Group>()
         this.chunksMap.set(trackId, track)
        }

        let group = track.get(groupId)
        if (!group) {
         group = {
          currentChunk: 1,
          chunks: new Map<number, Datagram[]>(),
          readyChunks: new Channel<Uint8Array>(),
          delete: () => {
           track.delete(groupId)
          },
          done: false,
         }
         track.set(groupId, group)
        }
        this.mutex.release() // end atomic operation on chunksMap

        let datagrams = group.chunks?.get(sequenceNum)
        if (!datagrams) {
         datagrams = []
         group.chunks?.set(sequenceNum, datagrams)
        }
        datagrams.push({ number: sliceNum, data: data })
        if (group.timeout) clearTimeout(group.timeout) // clear group chunks' queue
         ↪  closure timeout
        group.timeout = setTimeout(() => {
         group.done = true // set group state to done
         void group.readyChunks.push(new Uint8Array(), true) // close ready chunks queue
        }, 2000) // timer that closes the ready chunks queue after time has elapsed with
         ↪  no new datagrams for this group
       } else if (splitData.length == 5) {
        const trackId = Number(splitData.shift()).toString() // decode track id
        const groupId = Number(splitData.shift()) // decode group id number
```

```
      const sequenceNum = Number(splitData.shift()) // decode object sequence number
      const sliceLen = Number(splitData.shift()) // decode slice number
      const msg = String(splitData.shift()) // decode message

      const group = this.chunksMap.get(trackId)?.get(groupId)

      if (group && msg == "end_chunk") {
       if (group.currentChunk <= sequenceNum) {
        const chunks = group.chunks // get group chunks map
        if (chunks) {
         const chunk = chunks.get(sequenceNum) // extract chunk for corresponding
          ↪  sequence
         if (chunk) {
          if (chunk.length > 1) {
           // if chunk was sliced, merge slices
           const unfused_data = chunk
            .sort((a, b) => a.number - b.number)
            .map((a) => a.data)

           let length = 0
           unfused_data.forEach((item) => {
            length += item.length
           })

           const data = new Uint8Array(length)
           let offset = 0
           unfused_data.forEach((item) => {
            data.set(item, offset)
            offset += item.length
           })
           // console.log(data.length, sliceLen)
           if (data.length == sliceLen) {
            void group.readyChunks.push(data) // add to chunks ready to be consumed
            group.currentChunk = sequenceNum
           } else
            console.log(
             "corrupted or incomplete chunk",
             sequenceNum,
             "of group",
             groupId,
             "of track",
             trackId,
            )
          } else {
           // console.log(chunk[0].data.length, sliceLen)
           if (chunk[0].data.length == sliceLen) {
            void group.readyChunks.push(chunk[0].data) // add to chunks ready to be
            ↪  consumed
            group.currentChunk = sequenceNum
           } else
            console.log(
             "corrupted or incomplete chunk",
             sequenceNum,
             "of group",
             groupId,
             "of track",
             trackId,
            )
          }
          chunks.delete(sequenceNum) // delete entry from incoming chunks
          if (group.timeout) clearTimeout(group.timeout) // clear group chunks' queue
          ↪  closure timeout
          group.timeout = setTimeout(() => {
           group.done = true // set group state to done
           void group.readyChunks.push(new Uint8Array(), true) // close ready chunks
           ↪  queue
          }, 2000) // timer that closes the ready chunks queue after time has elapsed
          ↪  with no new datagrams for this group
         }
        }
       } else {
        console.log(
         "Discarded late chunk: track",
         trackId,
         "group",
         groupId,
         "object",
         sequenceNum,
        )
       }
      }
     } else if (splitData.length == 3) {
      const trackId = Number(splitData.shift()).toString() // decode track id
      const groupId = Number(splitData.shift()) // decode group id number
      const msg = String(splitData.shift()) // decode message

      const group = this.chunksMap.get(trackId)?.get(groupId) // get group

      // received end message ?
      if (group && msg == "end") {
```

```
          if (group.timeout) clearTimeout(group.timeout) // clear group chunks' queue
          ↪  closure timeout
          group.done = true // set group state to done
          void group.readyChunks.push(new Uint8Array(), true) // close ready chunks queue
        }
      }
    }
    if (done) break
  }
 }
}
```

The function ***receiveDatagrams()*** runs in parallel with *#runObjects* in connection.ts, receiving datagrams from the various tracks and placing them in an internal **data structure**, organizing them first in a ***Map*** by track and then by group (*interface* ***Group***), and finally in an array of slices corresponding to a single chunk.

These slices are then ***moved*** and ***reassembled*** into complete chunks inside **dedicated queues** once a '*end_chunk*' datagram is received. These queues, each contained within the corresponding ***Group***, allow the streams of each segment to **retrieve** the chunks as soon as they become available, thanks to the *recv()* function.

Datagram mode - moq-rs

The transmission via datagram in moq-rs has been implemented by making modifications in the *publisher.rs* and *subscriber.rs* files.

Within the *publisher.rs* file, which is used for relay retransmission, the ***run_segment()*** function has been modified. This function retrieves the various *chunks* of a fragment, divides them into slices of a maximum size of 1024 bytes, and sends the data along with the header that identifies them in the form of a datagram using a function exposed by the QUIC library in use.

```
async fn run_segment(&self, id: VarInt, segment: &mut segment::Subscriber) ->
↪  Result<(), SessionError> {
  log::trace!("serving group: {:?}", segment);
  // log::error!("{:?}", self.tracks_num);

  let mut stream = self.webtransport.open_uni().await?;
  // Convert the u32 to a i32, since the Quinn set_priority is signed.
  let priority = (segment.priority as i64 - i32::MAX as i64) as i32;
  stream.set_priority(priority).ok();
```

```

while let Some(mut fragment) = segment.fragment().await? {
 log::trace!("serving fragment: {:?}", fragment);

 let object = message::Object {
  track: id,
  // Properties of the segment
  group: segment.sequence,
  priority: segment.priority,
  expires: segment.expires,

  // Properties of the fragment
  sequence: fragment.sequence,
  size: fragment.size.map(VarInt::try_from).transpose()?,
  timestamp: segment.timestamp,
 };
 object // send header first
  .encode(&mut stream, &self.control.ext)
  .await
  .map_err(|e| SessionError::Unknown(e.to_string()))?;
 let mut chunk_counter = object.sequence.to_string().parse::<i32>().unwrap();
 let mut datagram_mode = false;
 if let Some(chunk) = fragment.chunk().await? {// first chunk
  // check first chunk type
  // log::error!("{:?} {:?}", chunk[0], chunk [3]);
  // log::error!("{:?}", chunk);
   chunk_counter += 1;
   datagram_mode = true;
   let mut slice_counter = 0;
   let tr_id = format!("{:?} ", object.track);
   let group = format!("{:?} ", object.group);
   let sequence = format!("{:?} ", chunk_counter);
   let len = format!("{:?} ", chunk.len());
   for chunk_slice in chunk.chunks(1024) {
    let slice_len = format!("{:?} ", chunk_slice.len());
    slice_counter += 1;
    // log::error!("{:?}\n", chunk_slice);
    let slice_number = format!("{:?} ", slice_counter);
    let mut _obj = [tr_id.as_bytes(), group.as_bytes(), sequence.as_bytes(),
    ↪ slice_number.as_bytes(), slice_len.as_bytes(), chunk_slice].concat().into();
    self.webtransport.send_datagram(_obj).await?; // send as datagram
   }
   let end = "end_chunk";
   let mut _obj = [tr_id.as_bytes(), group.as_bytes(), sequence.as_bytes(),
   ↪ len.as_bytes(), end.as_bytes()].concat().into();
```

```
      self.webtransport.send_datagram(_obj).await?; // send as datagram
    }
    while let Some(chunk) = fragment.chunk().await? {
     if datagram_mode { // when in datagram mode, send remaining chunks in datagrams
      chunk_counter += 1;
      let mut slice_counter = 0;
      let tr_id = format!("{:?} ", object.track);
      let group = format!("{:?} ", object.group);
      let sequence = format!("{:?} ", chunk_counter);
      let len = format!("{:?} ", chunk.len());
      for chunk_slice in chunk.chunks(1024) {
       let slice_len = format!("{:?} ", chunk_slice.len());
       slice_counter += 1;
       // log::error!("{:?}\n", chunk_slice);
       let slice_number = format!("{:?} ", slice_counter);
       let mut _obj = [tr_id.as_bytes(), group.as_bytes(), sequence.as_bytes(),
        ↪  slice_number.as_bytes(), slice_len.as_bytes(), chunk_slice].concat().into();
       self.webtransport.send_datagram(_obj).await?; // send as datagram
      }
      let end = "end_chunk";
      let mut _obj = [tr_id.as_bytes(), group.as_bytes(), sequence.as_bytes(),
       ↪  len.as_bytes(), end.as_bytes()].concat().into();
      self.webtransport.send_datagram(_obj).await?; // send as datagram
     }
     else { // catalog and init data
      stream.write_all(&chunk).await?; // send chunks in stream
     }
    }

    if datagram_mode { // send end fragment inside a reliable stream when in datagram
    ↪  mode
     let tr_id = format!("{:?} ", object.track);
     let group = format!("{:?} ", object.group);
     let end = "end";
     let mut _obj = [tr_id.as_bytes(), group.as_bytes(),
     ↪  end.as_bytes()].concat().into();
     self.webtransport.send_datagram(_obj).await?; // send as datagram
    }
   }

   Ok(())
  }
```

Within the *subscriber.rs* file, which is used for receiving segments from other publishers (in this case, the moq-js publisher instance), the *run_stream()* function has been altered so that, instead of extracting chunks from the QUIC stream in which the group header is received, the new *fragments* that make up an incoming MoQT segment are received via the ***get_unreliable_fragment()*** function. This function:

- Initializes the necessary data structures, specifically a *HashMap* that maps each *track id*, and thus each track, to another *HashMap* where the data of each MoQ group are stored, associating them with the corresponding *group id*. The data of a *group id* consists of a *HashMap* that maps the chunk data (array of chunk slices) to its sequence number and a queue where reassembled chunk data are inserted by another thread in sequence order.
- Waits for complete chunks in the corresponding group's queue until it is closed and emptied, while inserting the various chunks into the fragment in use.

```rust
async fn get_unreliable_fragment(&self , segment: &mut segment::Publisher, object: &
↪  message::Object)
    -> Result<(), SessionError> {
  // Create the first fragment
  let mut fragment = segment.push_fragment(object.sequence,
  ↪  object.size.map(usize::from))?;
  log::trace!("next fragment: {:?}", fragment);

  let track_id = object.track.to_string().parse::<i32>().unwrap_or(-1);
  let group_id = object.group.to_string().parse::<i32>().unwrap_or(-1);
  let mut rcv:  Option<Receiver::<Bytes>> = None;

  if track_id != -1 && group_id != -1 { // received valid header data
   let mut tracks = self.track_map.lock().await; // get shared tracks map

   if !tracks.contains_key(&track_id) {
    let value = HashMap::new();
    tracks.insert(track_id.clone(), value); // add track if not already present
   }
   let track = tracks.get_mut(&track_id).unwrap(); // get track
   if !track.contains_key(&group_id) {
    let (s, r) = unbounded::<Bytes>();
    rcv = Some(r.clone());
    let value = Group {
     chunks: HashMap::new(),
     ready_chunks: Channel::<Bytes> { receiver: r, sender: s },
    };
    track.insert(group_id.clone(), value); // add group if not already present
   }
   else {
    match tracks.get_mut(&track_id) {
     Some(track) => { match track.get_mut(&group_id) { // track found
      Some(group) => { // group found
       rcv = Some(group.ready_chunks.receiver.clone());
      } None => {log::error!("Requested group not found: {:?}\n", group_id);}
     }} None => {log::error!("Requested track not found: {:?}\n", track_id);}
    }
   }

   drop(tracks); // release mutex

   match rcv {
    Some(received) => {
     loop {
                        // set a timeout to wait for segment chunks for a limited time
      match timeout(Duration::from_millis(2000), received.recv()).await {
       Ok(result) => { // future resolved before timout
        match result {
         Ok(chunk) => { // got a new chunk
          log::trace!("next chunk: {:?}", chunk);
          fragment.chunk(chunk)?; // add chunk to fragment
         },
         Err(err) => {
          if received.is_closed() {
           break;
          }
          log::error!("Error retrieving ready chunks, {:?}", err);
          return Err(SessionError::Unknown(err.to_string()));
         }
        }
       },
       Err(..) => {
        log::error!("Timout for group {:?} of track {:?}\n", group_id, track_id);
        break;
       }
      }

     }
     let mut tracks = self.track_map.lock().await; // get shared tracks map
     match tracks.get_mut(&track_id) { // get track
      Some(track) => track.remove(&group_id), // remove group, not used anymore
      None => {return Ok(())},
     };


    }
    None => { return Err(SessionError::Unknown("Error obtaining channel
     ↪  receiver".to_string())); }
   }
  } else { log::error!("Invalid header data for new object"); }

  return Ok(());
 }
```

Also, within the same file, the ***run_datagrams()*** function has been created, which runs within a thread of the *tokio* library, created and started in the *run()* function.

This function:

- Receives the QUIC datagrams sent by the client-side transmitter and places them in an internal data structure based on the track, the group, and other unique identifiers such as the chunk id.
- When a chunk is complete, and a datagram with the message *"end_chunk"* is received, it retrieves the received slices that make up the corresponding chunk from the internal data structure and combines them into an output chunk, which is then placed in a queue of chunks ready to be read for the corresponding group.
- When all the chunks of a group have been transmitted and a datagram with the message *"end"* is received, it closes the group's chunk queue, and the chunks within it remain available to be extracted until the queue is emptied by the *get_unreliable_fragment()* function.

```
async fn run_datagrams(self) -> () { // function that receives quic datagrams and
 ↪  extracts chunk data, placing them inside the allocated data structures
 loop {
  let read_data = self.webtransport.read_datagram().await; // wait for datagram
  match read_data {
   Ok(datagram) => { // received datagram
    // log::error!("{:?} \n", datagram);

    let val = unsafe { std::str::from_utf8_unchecked(datagram.as_ref()) }; // convert
    ↪  datagram content to string to read header data
    let str_data = val.split(" ").collect::<Vec<&str>>(); // split datagram content
    ↪  fields

    if str_data.len() > 5 { // audio or video data
```

```rust
        let track_id = str_data[0].parse::<i32>().unwrap_or(-1);
        let group_id = str_data[1].parse::<i32>().unwrap_or(-1);
        let sequence_num = str_data[2].parse::<i32>().unwrap_or(-1);
        let slice_num = str_data[3].parse::<i32>().unwrap_or(-1);
        let slice_len = str_data[4].parse::<i32>().unwrap_or(-1);
        // header fields

        if track_id != -1 && group_id != -1 && sequence_num !=-1 && slice_num != -1 && slice_len != -1 { // received valid header data
         let head = format!("{:?} {:?} {:?} {:?} {:?} ", track_id, group_id, sequence_num, slice_num, slice_len); // header data string

         let offset = head.len(); // get header end offset
         let data = datagram.slice(offset..); // slice data from datagram

         let mut tracks = self.track_map.lock().await; // get shared tracks map

         if !tracks.contains_key(&track_id) {
          let value = HashMap::new(); // initiate track value
          tracks.insert(track_id.clone(), value); // add track if not already present
         }
         let track = tracks.get_mut(&track_id).unwrap(); // get track

         if !track.contains_key(&group_id) {
          let (s, r) = unbounded::<Bytes>(); // initiate channel
          let value = Group { // initiate group
           chunks: HashMap::new(), // chunks map
           ready_chunks: Channel::<Bytes> { receiver: r, sender: s }, // ready chunks channel
          };
          track.insert(group_id.clone(), value); // add group if not already present
         }
         let group = track.get_mut(&group_id).unwrap(); // get group

         if !group.chunks.contains_key(&sequence_num) {
          let value = Vec::new(); // initiate chunk
          group.chunks.insert(sequence_num.clone(), value); // add chunk if not already present
         }
         let datagrams = group.chunks.get_mut(&sequence_num).unwrap(); // get chunk datagrams vector

         datagrams.push(Datagram { number: slice_num, data: data.clone() }); // add datagrams to chunk vector
         drop(tracks);
```

```
 }
}
else if str_data.len() == 5 { // end chunk message
 let track_id = str_data[0].parse::<i32>().unwrap_or(-1);
 let group_id = str_data[1].parse::<i32>().unwrap_or(-1);
 let sequence_num = str_data[2].parse::<i32>().unwrap_or(-1);
 let slice_num = str_data[3].parse::<i32>().unwrap_or(-1);
 let msg = str_data[4].to_string();
 // header fields

 if track_id != -1 && group_id != -1 && sequence_num !=-1 && slice_num != -1 &&
 ↪ msg == "end_chunk" { // received valid header data
  let mut tracks = self.track_map.lock().await; // get shared tracks map

  match tracks.get_mut(&track_id) { // get track if present
   Some(track) => { // track found
    match track.get_mut(&group_id) { // get group if present
     Some(group) => { // group found
      match &mut group.chunks.get_mut(&sequence_num) { // get chunk if present
        Some(chunk) => { // chunk datagrams found
         let mut out_chunk: Vec<u8> = Vec::new(); // initiate output chunk
         chunk.sort_by(|a, b| a.number.cmp(&b.number)); // sort slices by
         ↪ datagram number
         for slice in chunk.iter() { // iterate for each slice
          out_chunk.extend(slice.data.to_vec()); // append slice data to output
          ↪ chunk
         }
         let ready_chunk: Bytes = Bytes::from(out_chunk); // convert output
         ↪ vector to byte array
         let sent = group.ready_chunks.sender.try_send(ready_chunk);
         match sent { // add chunk to queue of ready chunks
          Ok(..) => { },
          Err(error) => {log::error!("Error sending slice to group channel:
          ↪ {:?}\n", error);}
         }
        } None => {log::error!("Requested chunk not found: {:?}\n",
        ↪ sequence_num);}}
     } None => {log::error!("Requested group not found: {:?}\n", group_id);}
   }} None => {log::error!("Requested track not found: {:?}\n", track_id);}
  }

  drop(tracks); // release mutex

 }
}
```

```
     else if str_data.len() == 3 {
      let track_id = str_data[0].parse::<i32>().unwrap_or(-1);
      let group_id = str_data[1].parse::<i32>().unwrap_or(-1);
      let msg = str_data[2].to_string();
      if track_id != -1 && group_id != -1 && msg == "end" {
       let mut tracks = self.track_map.lock().await; // get shared tracks map

       match tracks.get_mut(&track_id) {
        Some(track) => { match track.get_mut(&group_id) { // track found
         Some(group) => { // group found
          group.ready_chunks.sender.close(); // close channel on sender side
         } None => {log::error!("Requested group not found: {:?}\n", group_id);}
        }} None => {log::error!("Requested track not found: {:?}\n", track_id);}
       }

       drop(tracks); // release mutex

      }
     }
    }
    Err(..) => {log::error!("Error reading new datagrams"); break;}
   }
  }
 }
```

Chapter 7

Environment setup and tools

This chapter describes the testing environment, tools, and setup used to evaluate the performance of the MoQT (Media over QUIC Transport) implementation. The section outlines both local and remote deployment procedures, elaborates on the logging infrastructure, and details the tools for data capture and analysis.

Overview

The system under test is composed of two primary components:

- **moq-js**: JavaScript-based publisher and subscriber client running in the Chrome browser.
- **moq-rs**: Rust-based relay server implementing the MoQ relay logic.

The test environment supports local and remote deployments, with packet tracing, custom logging, and data analysis tools developed to evaluate latency, jitter, and CPU usage during content publication and consumption.

7.1 Software Sources

- **moq-js**: Available on GitHub at `https://github.com/kixelated/moq-js`. To install, clone the repository, run `npm install`, and launch with `npm run start` or `npm run dev` for logging.
- **moq-rs**: Available at `https://github.com/kixelated/moq-rs`. Install Rust from `https://rustup.rs`, clone the repository, and run the relay with Cargo or the provided script `./dev/pub`.
- **mkcert fork for TLS**: Self-signed TLS certificates were generated using a modified version of `mkcert`, available at `https://github.com/kixelated/mkcert.git`.

7.2 moq-js

Overview and Installation

`moq-js` is a JavaScript-based implementation of a Media over QUIC (MoQ) publisher/-subscriber client developed by Kixelated.
The version used for these experiments corresponds to the latest GitHub repository commit of a fork of the original repository, available at:

https://github.com/s277945/moq-js.git

To install `moq-js`, the following steps are required:

1. Ensure Node.js (version `v20.11.1` or higher) and npm are installed.
2. Clone the repository and install dependencies:

```
git clone https://github.com/s277945/moq-js.git
cd moq-js
npm install
```

Execution and Usage

The Node server can be started in one of two modes:

- `npm run start` – launches the server in standard mode.
- `npm run dev` – activates developer mode with additional telemetry logging capabilities.

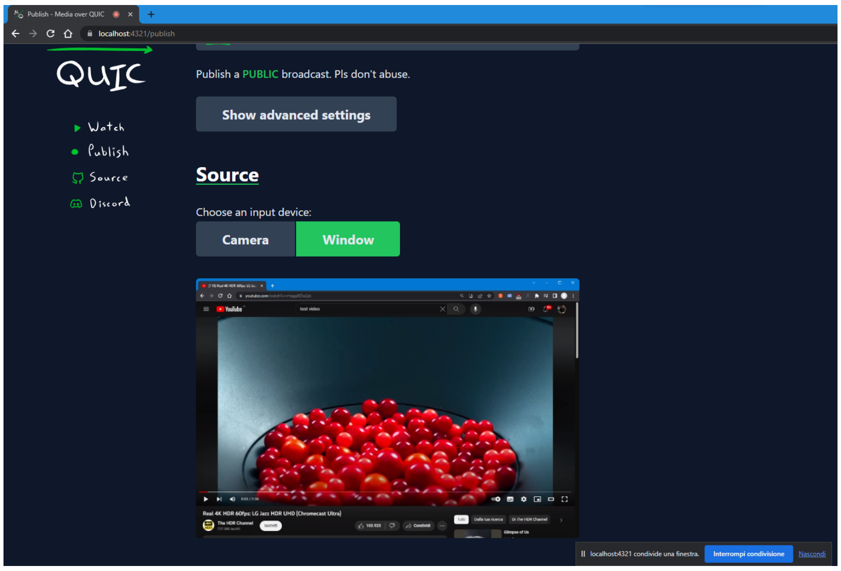
Once launched, the client can be accessed via a Chromium-based browser (preferably Chrome) at:

https://localhost:4321

A new publisher session can be initialized from:

https://localhost:4321/publish

The source stream can be configured as a webcam/microphone feed or screen capture.



Publish a PUBLIC broadcast. Pls don't abuse.

Choose an input device:

Figure 7.1. *https://localhost:4321/publish*

7.3 moq-rs

Overview and Installation

`moq-rs` is a Rust-based relay server implementation of the MoQ protocol, also developed by Kixelated.

Original repository

The version used for all tests on QUIC stream based relay transmission corresponds to commit hash `159a175` of the following repository:

https://github.com/kixelated/moq-rs

Installation instructions:

1. Install Rust toolchain (`rustc` and `cargo`) via `rustup`.
2. Clone the repository:

```
git clone https://github.com/kixelated/moq-rs
cd moq-rs
```

3. Build and run the relay server with:

```
cargo run --bin moq-relay --tls-cert dev/localhost.crt
    --tls-key dev/localhost.key --dev
```

Forked modified version

The version used for all test on QUIC datagram based relay transmission corresponds to the latest commit of the following repository:

https://github.com/s277945/moq-rs.git

Installation instructions:

1. Install Rust toolchain (`rustc` and `cargo`) via `rustup`.
2. Clone the repository:

   ```
   git clone https://github.com/s277945/moq-rs.git
   cd moq-rs
   ```

3. Build and run the relay server with:

   ```
   cargo run --bin moq-relay --tls-cert dev/localhost.crt
       --tls-key dev/localhost.key --dev
   ```

Alternatively, for both versions a convenience script `./dev/relay` is provided.

7.4 Logging

To enable robust telemetry, diagnostics, and performance evaluation for streaming sessions, we implemented a custom logging subsystem that communicates with an external Node.js log server. This architecture supports real-time, fine-grained, and extensible logging of media protocol events, transport-level details, and application-specific metrics.

7.4.1 Logger Module in moq-js (`lib/common/logger.ts`)

Client-side logging is centralized in the module `lib/common/logger.ts`. This file defines a suite of asynchronous functions, each designed to interface with the external logging server through HTTP(S) requests. Below, we provide a detailed code analysis of its core components.

Logger Configuration and State

At the top of `logger.ts`, several configuration constants and variables define the logger's behavior:

```
const LOGGER_URL = "https://localhost:8443";
let loggerStatus: boolean | undefined;
let logFile: string | undefined;
```

This module statically configures the base URL for the log server, ensuring all HTTP requests are routed consistently. The use of persistent module-level variables for logger status and the active log file allows the logging system to efficiently cache state between calls, avoiding redundant server probes and reducing overhead in high-frequency logging scenarios. By gating all operations on these variables, the design ensures both robustness (failing gracefully if the logger is unavailable) and performance (minimizing unnecessary network chatter).

Retrieving Log Server Status: `getLoggerStatus()`

```
export async function getLoggerStatus(): Promise<boolean> {
  try {
    const res = await fetch(`${LOGGER_URL}/latency-data`);
    loggerStatus = res.ok;
    return loggerStatus;
  } catch {
    loggerStatus = false;
    return false;
  }
}
```

This function serves as a health-check for the logging subsystem. By probing a lightweight endpoint (`/latency-data`), the system can quickly detect and cache the availability of the logging backend. This mechanism is critical in production environments: it avoids repeated failures and wasted resources by short-circuiting further logging attempts if the server is down, thus protecting the main streaming pipeline from unnecessary slowdowns or timeouts.

Log File Initialization: `initLoggerFile()`

```
export async function initLoggerFile(): Promise<string | undefined> {
  if (loggerStatus === false) return undefined;
  const now = new Date();
  const name = `${now.toISOString().replace(/:/g, '-')}.log`;

  const res = await fetch(`${LOGGER_URL}/log-init?file=${name}`);
  if (res.ok) {
    logFile = name;
    return name;
  }
  return undefined;
}
```

This function orchestrates the creation of a new log file on the server, ensuring that each session's logs are isolated and uniquely identified. The filename includes a timestamp to prevent collisions in concurrent, multi-user deployments. Centralizing file creation here ensures that all subsequent log operations can rely on a valid and accessible file handle, while also simplifying cleanup and analysis on the backend.

Track Type Logging: `logTrackTypes()`

```
export async function logTrackTypes(types: { id: number, type: string }[]) {
  if (!logFile) return;
  await fetch(`${LOGGER_URL}/log-track-types?file=${logFile}`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(types)
  });
}
```

In complex streaming applications, a single session may multiplex multiple logical tracks (such as audio, video, or subtitles). This function allows the client to send a schema or manifest of active tracks at startup, significantly improving the post-hoc interpretability of log files and enabling correlation between protocol events and their corresponding media streams. It provides a foundation for advanced analytics, including per-track QoS and stream switching behavior.

Event Data Logging: `postLogDataAndForget()`

```
export async function postLogDataAndForget(data: LogData) {
  if (!logFile) return;
  fetch(`${LOGGER_URL}/log-data?file=${logFile}`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(data)
  }).catch(() => {});
}
```

This is the core method for telemetry. It is intentionally implemented in a fire-and-forget manner: logs are transmitted asynchronously, and failures are silently ignored. This design is essential for live media applications, where non-blocking operations and real-time guarantees must take precedence over perfect telemetry. Even if some logs are lost due to network issues, the media session remains unaffected, thus preserving end-user experience.

Module Interactions and Usage Points

Logging is invoked throughout the codebase. For instance, in `lib/transport/object.ts`, calls such as the following are used:

```
// When sending an object
postLogDataAndForget({
  trackId,
  objectId,
  groupId,
  status: "sent",
  timestamp: Date.now()
});
```

Here, logging captures the essential protocol metadata for every transmitted or received segment/object, including IDs and timestamps. Such fine-grained, event-level records are invaluable for offline debugging, performance analysis, and reconstructing session histories.

7.4.2 Server-side Logging (Node.js)

The Node.js backend, located in `logger/server.ts`, exposes a REST API to receive, persist, and organize logs from multiple concurrent clients.

Server Entrypoint: `logger/server.ts`

```
import express from "express";
import { logData, logTrackTypes, logInit, logEnd, logSkippedSegment } from
↪  "./api/logger";
const app = express();
app.use(express.json());

app.get("/latency-data", (req, res) => res.sendStatus(200));
app.post("/log-init", logInit);
app.post("/log-end", logEnd);
app.post("/log-data", logData);
app.post("/log-track-types", logTrackTypes);
app.post("/skipped-segment", logSkippedSegment);

app.listen(8443, () => console.log("Logger running on 8443"));
```

The server leverages Express.js to provide a RESTful interface, mapping each endpoint to a specific logging action. Each log type (e.g., session start, object event, track metadata, skipped segments) is handled by a specialized route and handler. This modularity promotes extensibility, allowing future protocol features or new telemetry types to be integrated with minimal disruption.

API Submodules

File and Log Management: `logger/api/file_status.ts` Manages the list of open log files, safely appending lines, and handling file creation as needed.

```
const files = new Map<string, fs.WriteStream>();

export function openLogFile(name: string) {
  if (!files.has(name)) {
    files.set(name, fs.createWriteStream(path.join(LOG_DIR, name), { flags: 'a' }));
  }
  return files.get(name)!;
}
```

This code efficiently manages log file handles using a `Map`, ensuring that each log file is opened only once per process and all writes are correctly appended. Such a strategy is essential in environments with many concurrent sessions, preventing file corruption and supporting high-throughput log ingestion.

Latency Tracking: `logger/api/latency.ts` Tracks and aggregates latency data from received logs, implementing functions to compute average, minimum, and maximum latency, among others, which can be exposed in logs or summary endpoints. This module is designed to collect and process latency metrics reported from clients. By holding transient in-memory statistics, the system enables both live and batched reporting, facilitating the analysis of end-to-end performance bottlenecks, jitter, and real-time responsiveness.

Logging Logic: `logger/api/logger.ts` Centralizes the parsing, validation, and formatting of incoming log events, calling file management functions to write logs atomically.

```
export function logData(req, res) {
  const file = openLogFile(req.query.file);
  file.write(JSON.stringify(req.body) + "\n");
  res.sendStatus(200);
}
```

This approach implements a robust, append-only log format by writing each JSON event to its own line in the log file. Such a strategy is highly compatible with standard analytics tools, allows for easy parsing, and is scalable across large data volumes. The atomic write model ensures data consistency even under heavy concurrent logging loads.

7.4.3 Per-Module Commentary and File Structure Cross-References

- **Client-side logging:**
  - `lib/common/logger.ts`: Central interface for all logging logic in the browser. It is invoked by playback, transport, and protocol modules to record media and protocol events.
  - `lib/transport/object.ts`: Primary consumer of the logger, capturing each sent or received object for protocol traceability.
- **Server-side logging:**
  - `logger/server.ts`: Express application that defines the REST API surface for all telemetry ingestion.
  - `logger/api/logger.ts`: Main handler for request parsing, validation, and file writes; encapsulates logic for each log type.
  - `logger/api/file_status.ts`: Manages log file handles and ensures safe, concurrent access for log appends.
  - `logger/api/latency.ts`: Aggregates and summarizes latency-related metrics for quality analysis.

7.4.4 Log Data Format

The log data format is designed for extensibility and ease of analysis. Each log record contains a set of standardized fields, as shown in Table 7.1, enabling structured, machine-parsable logging. File headers and status lines are also inserted at key lifecycle events, supporting robust log parsing and post-hoc analytics.

Track ID	Object ID	Group ID	Status	Timestamp	**File header**
-	-	-	PUBLISHER START	-	**Publisher log start**
-	-	1715085484507	CPU	0.68	**CPU load log**
3	-	AUDIO	-	-	**Track log (audio)**
4	-	VIDEO	-	-	**Track log (video)**
-	-	-	SUBSCRIBER START	-	**Subscriber log start**
0	0	0	sent	01715085489677	**Sent object log**

Table 7.1. Log file format

7.5 Data plot and analysis utilities - Python scripts

In order to analyze the data, various Python scripts have been created to provide graphical representations of different parameters from the collected data. The scripts use data retrieved from moq-js log files, log files produced by tshark, and Wireshark output in JSON format.

7.5.1 plotMoqjsTimestamp.py

The `plotMoqjsTimestamp.py` script constitutes a comprehensive and extensible utility for the post-processing and visualization of networked media transmission data, built specifically for logs generated by `moq-js`. It offers comprehensive support for plotting fine-grained, timeline-aligned views of latency, jitter, retransmissions, and anomaly classification ("lost", "too old", "too slow") at the packet and track level, with optional overlays for retransmission events and CPU usage, thus delivering both granular and system-level insights. This section describes not only its features, but also the rationale, inner workflow, and key code patterns that enable its powerful analytics.

Input Parameters, File Formats, and Output Files

Input Files

- **Main log file (`-f, -file`):**
  The main input is a text log file produced by the `moq-js` logger, containing a sequence of events (one per line) related to each media packet or fragment. Each row consists of semicolon-separated fields, for example:

  ```
  0;1;23;sent;1690034320000;5
  0;1;23;received;1690034320050;7
  0;1;24;too slow;1690034320123
  ```

  `(track_id; fragment_id; packet_id; event_type; timestamp; [jitter])`
  Where:

  – `track_id`: Track identifier (e.g., for audio or video)
  – `fragment_id`, `packet_id`: Packet or fragment indices

  - `event_type`: Type of event (`sent`, `received`, `too slow`, `too old`, `AUDIO`, `VIDEO`, `CPU`)
  - `timestamp`: Event timestamp in milliseconds
  - `jitter`: (Optional) Jitter value in ms, if present

- **Wireshark parsed retransmission file (`-wpf, -wshparsedfile`) (optional):** Optionally, a text file derived from Wireshark analysis that reports, for each packet, the number of observed transport retransmissions.

Parameter	**Abbreviation**	**Description**
`-file`	`-f`	Main moq-js log file to analyze.
`-wshparsedfile`	`-wpf`	Wireshark retransmission file (optional).
`-skipstart`	`-sks`	Number of initial packets to skip.
`-skipend`	`-ske`	Number of trailing packets to skip.
`-maxheight`	`-mh`	Maximum Y-axis limit for latency/jitter (see below).
`-cpulog`	`-cpu`	Overlay CPU usage subplot (`true`/`false`).
`-sharex`	`-shx`	Share X-axis among subplots (`true`/`false`).
`-logslow`	`-lsl`	Log “too slow” events (`true`/`false`).
`-pheader`	`-phd`	Header packet filtering mode (`true`/`false`/`only`).
`-showlost`	`-shl`	Plot cumulative anomaly counts (`true`/`false`).
`-savefile`	`-sf`	Save plots to disk instead of displaying (`true`/`false`).
`-min_tick_spacing`	`-mts`	Minimum X-label spacing in pixels (`default 80`).
`-lost_height`	`-lsth`	Anomaly bar rendering mode (see Table **??**).
`-grid`	`-g`	Show horizontal grid lines.
`-nojitter`	`-lsth`	Disable jitter plots.

Table 7.2. Primary input parameters for `plotMoqjsTimestamp.py`.

Main Input Options

Output Files Depending on options, the script generates one or more of the following outputs:

- **Interactive Plot Display:** (Default) Plots are shown on-screen via `matplotlib`. Plots support dynamic window resizing with automatic X-label adaptation.
- **Plot files saved to disk:** If `-savefile true` is specified, plots are saved in both `.png` and `.svg` formats in the current directory. Filenames are derived from the

input log by replacing `.txt` with `_cumuloss.png` and `_cumuloss.svg`. For example, for `log.txt`:

  – `log_cumuloss.png`
  – `log_cumuloss.svg`

- **Console output log:** If both `-savefile true` and `-file` are specified, console output (summary messages) is also saved to a text file named `log_cumuloss.out.txt`.
- **Console messages:** Progress and summary information (*"Tracks found: ..."*, *"Total packets: ..."*, etc.) are printed to `stdout` and optionally logged as above.

File Format and Dependencies

- **Input format:** All log files must be plain text, with semicolon-separated fields as illustrated above.
- **Dependencies:** The script requires Python 3.7+ and the modules `matplotlib` and `numpy`.

```
pip install matplotlib numpy
```

Key features

Automatic Track Recognition

Upon execution, the script parses the input file and automatically detects all present media tracks (e.g., Audio, Video), assigning them meaningful names. Each stream is then analyzed and visualized independently, supporting simultaneous comparison of multiple flows within the same session.

Detailed Workflow and Metric Extraction

Each entry in the `moq-js` log is parsed according to its event type (sent, received, too old, too slow, etc.). The script:

- Distinguishes between sent and received packets, extracting relevant timestamps for each;
- Computes one-way latency as the difference between send and receive times;
- Extracts and processes jitter values if available;
- Classifies and marks each packet for anomalies (lost, too slow, too old), organizing all data per track for efficient downstream analysis.

Advanced Visualization and Filtering

The script provides fine-grained, user-configurable control over content and layout:

- Skipping initial or terminal packets (`-skipstart`, `-skipend`), which is especially useful for focusing analysis on steady-state transmission and excluding startup or teardown effects;
- Selective inclusion or exclusion of packets based on header type (`-pheader`);
- Visualization of lost, too old, and too slow packets using a variety of simulated latency models (`-lost_height`), ranging from minimal marking to statistically-informed or contextually interpolated values, supporting both analytical ("worst-case") and realistic ("neighbor-based") assessments.

Auxiliary Metrics and System-Level Correlation

When auxiliary data is provided, `plotMoqjsTimestamp.py` overlays synchronized subplots for retransmissions and CPU utilization, enabling a holistic view of system and network interactions. This supports root-cause analysis by correlating packet anomalies with underlying system or transport conditions.

Extra Features and Analytical Capabilities

The script blends direct, statistical, and customizable visual analytics:

1. Per-track, Time-aligned Visualizations Each detected track (audio, video, etc.) is visualized independently but aligned to a common time base, facilitating comparative study (e.g., audio smoothness vs. video spikes).

2. Y-Axis Scaling: `maxheight` parameter (per-track) A key feature of the script is its ability to dynamically or manually adjust the y-axis (vertical) range of both latency and jitter plots for each **individual track** via the `-maxheight` (`-mh`) parameter. This enhancement ensures more accurate and readable visualizations—especially in multi-track scenarios such as simultaneous audio and video streams, where each stream may have distinct latency and jitter characteristics.

The `maxheight` parameter can be set either to a numeric value or to the special keyword `auto`:

- **If `maxheight=auto`:** The upper limit (*ylim*) of each track's latency plot is set to **three times the mean observed latency for that specific track** (across all received packets in the track). This provides a visually adaptive scale tailored to the behavior of each stream, improving clarity in mixed or asymmetric scenarios.

The y-limit for each jitter plot is set to **one tenth of its corresponding latency y-limit**, with an absolute cap (e.g., 20 ms) to avoid excessive scaling.

- **If `maxheight=N` (with $N > 0$):** The latency y-axis for **every track** is capped to N, and the corresponding jitter y-axis to $N/10$. This manual mode is useful for direct comparison across runs or when a consistent range is required for publication or regression analysis.
- **If `maxheight` is omitted or invalid:** The plot will auto-scale according to matplotlib's default behavior, which may result in the axes adapting freely to outlier values.

Internally, after all bars are plotted, the script computes the mean latency for each track and sets the y-axis upper limit for both latency and jitter subplots associated to that track. This is done via direct mapping between axes and tracks (not by scanning labels), guaranteeing correct and robust scaling even with multiple tracks or subplot permutations.

This approach brings several benefits:

- **Visual comparability:** Tracks with different latency regimes remain visually interpretable and comparable, while keeping consistent scaling across the same type of streams.
- **Readability:** Typical patterns for each track emerge clearly, without being masked by the worst-case outliers of another stream.
- **Adaptability:** The `auto` mode adapts to the actual data of each track—making it ideal for exploratory analysis—while the manual mode is perfect for formal reporting or cross-experiment comparisons.

Example: Y-Axis Scaling command lines

```
# Automatic y-axis scaling for each track
python3 plotMoqjsTimestamp.py -f mylog.txt --maxheight auto

# Manual y-axis scaling: every latency plot up to 300ms, jitter up to 30ms
python3 plotMoqjsTimestamp.py -f mylog.txt --maxheight 300
```

Technical Note. This feature affects only the **latency** and **jitter** plots for each individual track. Other subplots (e.g., cumulative lost packets, retransmissions, CPU usage) remain unaffected and retain their default matplotlib auto-scaling.

3. Dynamic, Adaptive X-axis Labeling The script employs a window-aware adaptive X-label strategy. This ensures that—no matter how many packets or how much the window is resized—tick density and alignment remain readable and informative. The following code governs this adaptive labeling:

Code sample: Adaptive X-label placement

```python
def set_all_xticks(axs, x_positions, x_labels, min_tick_spacing_px=80):
    fig = axs[0].figure
    width_px = fig.get_figwidth() * fig.dpi
    show_every = max(1, int(np.ceil(len(x_labels) / max(2, width_px //
    ↪  min_tick_spacing_px))))
    indices = [i for i in range(len(x_labels)) if i % show_every == 0]
    labels = [x_labels[i] for i in indices]
    for ax in axs:
        ax.set_xticks(indices)
        ax.set_xticklabels(labels, rotation=45, ha='right')
        ax.xaxis.set_tick_params(labelbottom=True)
```

4. Customizable Anomaly Rendering: Simulated Latency Bars A hallmark feature is the ability to render anomalies (lost, too old, too slow) with user-selectable semantics, making the visualization either analytical (e.g., "worst-case" with infinite bars) or realistic (neighbor/interpolated). This is achieved through the `simulated_latency_for_track` function:

Code sample: Simulated Latency Computation

```python
def simulated_latency_for_track(track, idx, all_indices, lost_height_mode):
    latencies = [e.latency for e in track.values() if e.isReceived() and e.latency >
    ↪  0]
    jitters = [e.sender_jitter for e in track.values() if e.isReceived() and
    ↪  e.sender_jitter is not None]
    if lost_height_mode == 'max_plus_50':
        return max(latencies) + 50 if latencies else 1000
    elif lost_height_mode == 'mean_jitter_plus_2std':
        mean_jitter = np.mean(jitters) if jitters else 0
        std_latency = np.std(latencies) if latencies else 0
        return mean_jitter + 2*std_latency
    # ...other modes (avg_neighbor, last10_mean, etc.) as in the main script...
    return 1000
```

Simulated Latency Computation Strategy Table:

Strategy (-lost_height)	**Interpretation**
`1`	Always height 1; useful for count-only anomaly marking.
`infinite`	Fills Y axis; emphasizes outlier status visually and analytically.
`max_plus_50`	Maximum observed latency + 50 ms; pessimistic upper bound.
`mean_jitter_plus_2std`	Mean jitter plus 2 std. devs of valid latencies; a robust outlier threshold.
`avg_neighbor`	Mean of previous and next received latencies; contextually realistic.
`last10_mean`	Mean of last 10 received latencies; robust to short bursts/transients.
`last5_mean`	Mean of last 5; ultra-local smoothing.
`mean_x3`	Mean of values x 3 with colored x on top.

Table 7.3. Lost/anomaly rendering strategies and interpretations.

labeltab:lostheight-modes

5. Cumulative Anomaly Curves With `-showlost true`, the script computes and plots a cumulative, time-aligned step curve for all anomaly types. This immediately reveals when and how anomalies cluster—crucial for diagnosing bursts, synchronization lapses, or outage events.

Code sample: Cumulative Anomaly Computation

```python
def build_cumulatives(tracks, startTS, data):
    entries = []
    for track in tracks.values():
        for elem in track.values():
            if elem.sender_ts is not None:
                lost = not elem.isReceived() and not elem.isTooOld() and not
                ↪  elem.isTooSlow()
                tooold = elem.isTooOld()
                tooslow = elem.isTooSlow()
                entries.append((elem.sender_ts, lost, tooold, tooslow))
    entries.sort()
    # Walk timeline, accumulate counts
    ...
    return x, lost_y, tooold_y, tooslow_y
```

Design Rationale and Data Architecture

At the heart of the tool is the `rowData` class, which provides a unified object model for each media fragment (packet). A `rowData` instance encapsulates essential metrics and metadata:

- Timestamps for sender/receiver events.
- Jitter values (sender and receiver, if present).
- Flags for anomalies (tooOld, tooSlow).
- Track/stream identifiers and plotting color.
- Retransmission counter (optional, via Wireshark).
- Calculated latency, inferred on construction if both timestamps are present.

Code sample: rowData class

```python
class rowData:
    def __init__(self, name, latency=None, color=None, sender_ts=None,
    ↪  receiver_ts=None,
                 sender_jitter=None, receiver_jitter=None, value=None, tooOld=False,
                 ↪  tooSlow=False):
        self.name = name
        self.color = color
        self.sender_ts = sender_ts
        self.receiver_ts = receiver_ts
        self.sender_jitter = sender_jitter
        self.receiver_jitter = receiver_jitter
        self.value = value
        self.tooOld = tooOld
        self.tooSlow = tooSlow
        self.retransmissions = 0
        if (sender_ts is not None and receiver_ts is not None and receiver_ts >
        ↪  sender_ts):
            self.latency = receiver_ts - sender_ts
        elif (latency is not None):
            self.latency = latency
        else:
            self.latency = 0
    def isReceived(self):
        return self.receiver_ts is not None
    def isTooOld(self):
        return self.tooOld
    def isTooSlow(self):
        return self.tooSlow
    # ...other accessors/mutators...
```

Each log row is mapped to a `rowData` instance, enabling efficient mapping from packet sequence to timeline and rapid filtering or aggregation by track and anomaly status. The script's input workflow can be summarized as:

1. Parse each line from the main `moq-js` log into a `rowData` object, keyed by track and fragment id.

2. Parse auxiliary files (Wireshark retransmissions, CPU usage logs) if supplied.

3. Compute all derived metrics (latency, jitter, anomaly state) and store in per-track dictionaries for later plotting.

Workflow Summary

1. **Input parsing:** Load main log (`moq-js`) and any auxiliary data; instantiate `rowData` objects.

2. **Preprocessing:** Compute derived metrics (latency, jitter, status), group by track, build X-timeline.

3. **Anomaly simulation:** Compute simulated "latency" for lost/too old/too slow packets, according to user-specified `-lost_height`.

4. **Plotting:** Assemble subplot grid, assign colors/status, adaptive X-ticks, overlay auxiliary metrics.

5. **Display/export:** Show interactively, or save PNG/SVG (`-savefile true`).

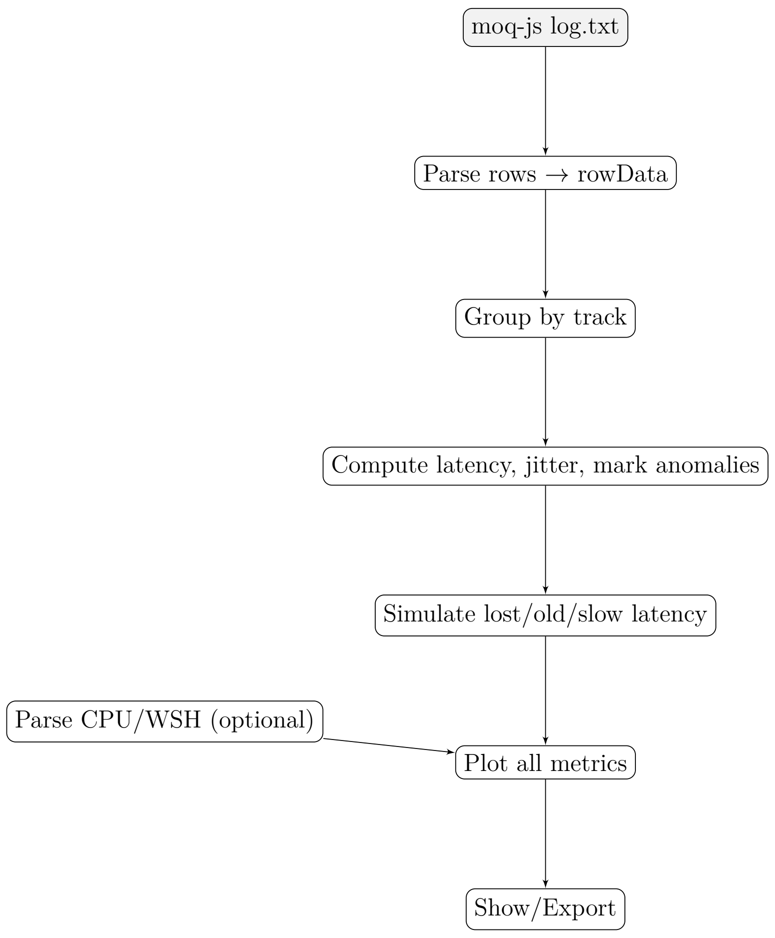


Figure 7.2. Workflow of `plotMoqjsTimestamp.py`: from log ingestion to visualization/export.

Visualization

Visualization in `plotMoqjsTimestamp.py` is designed to maximize clarity, analytical value, and direct interpretability for packet-level diagnostics in networked media systems.

The script dynamically generates a set of timeline-aligned bar and step plots, with a focus on modularity and per-track insight. Depending on the enabled features and number of detected tracks, several subplots are arranged and labeled to support both comparative and in-depth single-stream analysis.

Core Visual Elements

- **Latency and Jitter Bar Plots:** For each track (audio, video, etc.), packet latencies and, when available, jitters are rendered as bar plots over time. Each bar's height represents the one-way latency or jitter value for the packet or fragment, and its color encodes the anomaly status—with standard mapping (e.g., red for lost, orange for too old, blue for too slow).

- **Cumulative Anomaly Step Curve:** When `-showlost true` is set, the primary subplot is replaced (or supplemented) by a cumulative, time-aligned step plot. This visually aggregates the number of lost, too old, and too slow packets as transmission progresses, revealing the temporal clustering and dynamics of anomalies.
- **Auxiliary Subplots:** If auxiliary data is supplied, additional subplots are included for retransmissions (from Wireshark-processed files) and CPU usage (if CPU logs are present and enabled). These share the X-axis with the media timeline, enabling direct correlation between packet events and system/network status.
- **Adaptive X-axis:** All subplots share a synchronized, dynamically-labeled X-axis. The label density is adapted to both the number of packets and the figure size, maintaining legibility even for long or high-rate traces. Upon resizing the plot window, X-ticks and labels are recalculated for optimal layout.

Anomaly Rendering Modes A unique aspect of the visualization is the user's ability to specify how lost, too old, or too slow packets are drawn in latency plots. The `-lost_height` argument determines whether anomalies are rendered with minimal marking, pessimistic "infinite" bars, or with statistically simulated latency based on the local timeline context (neighbor/interpolation/mean-based). This enables both worst-case and realistic visual analytics, matching a broad spectrum of diagnostic and research objectives.

Legend and Labeling Legends are automatically generated for all anomaly types, track labels, and auxiliary metrics. Subplot titles are context-aware (e.g., "Audio latency", "CPU usage (%)"), and all axes are annotated for immediate understanding by the viewer. This makes the figures suitable not only for internal analysis but also for inclusion in presentations and publications.

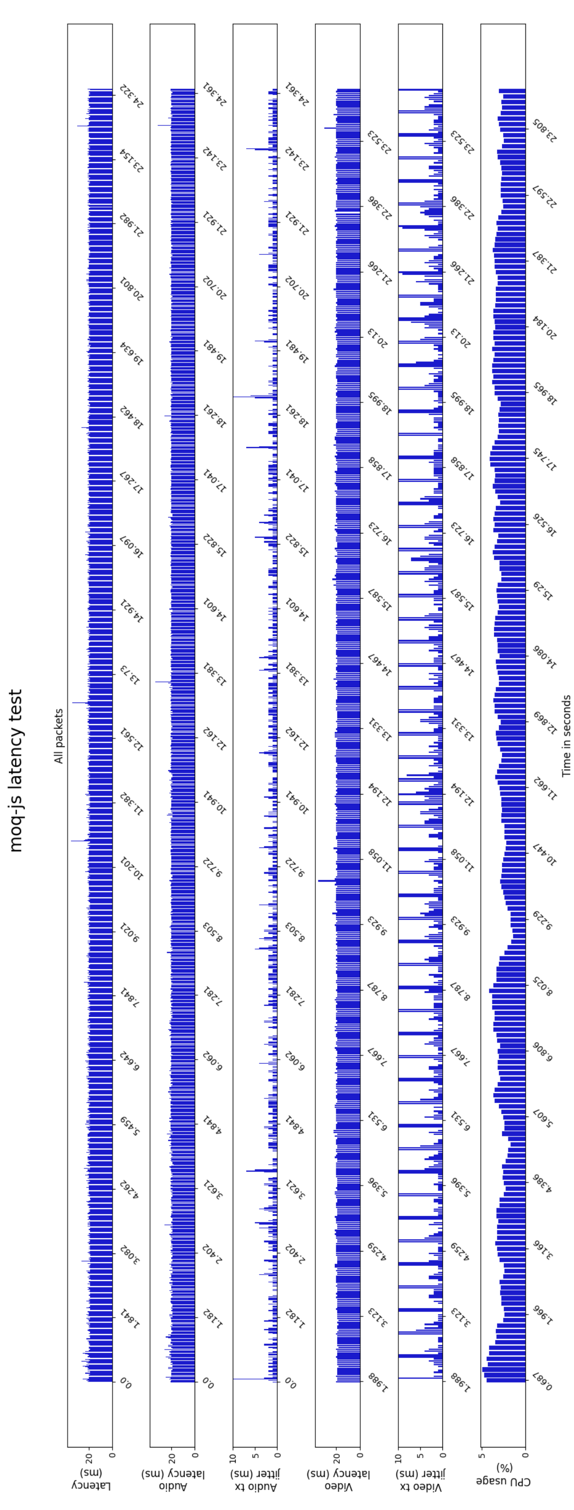


Figure 7.3. Timeline of packet latencies (bars), with lost/too old/too slow packets color-coded and optionally simulated in height.

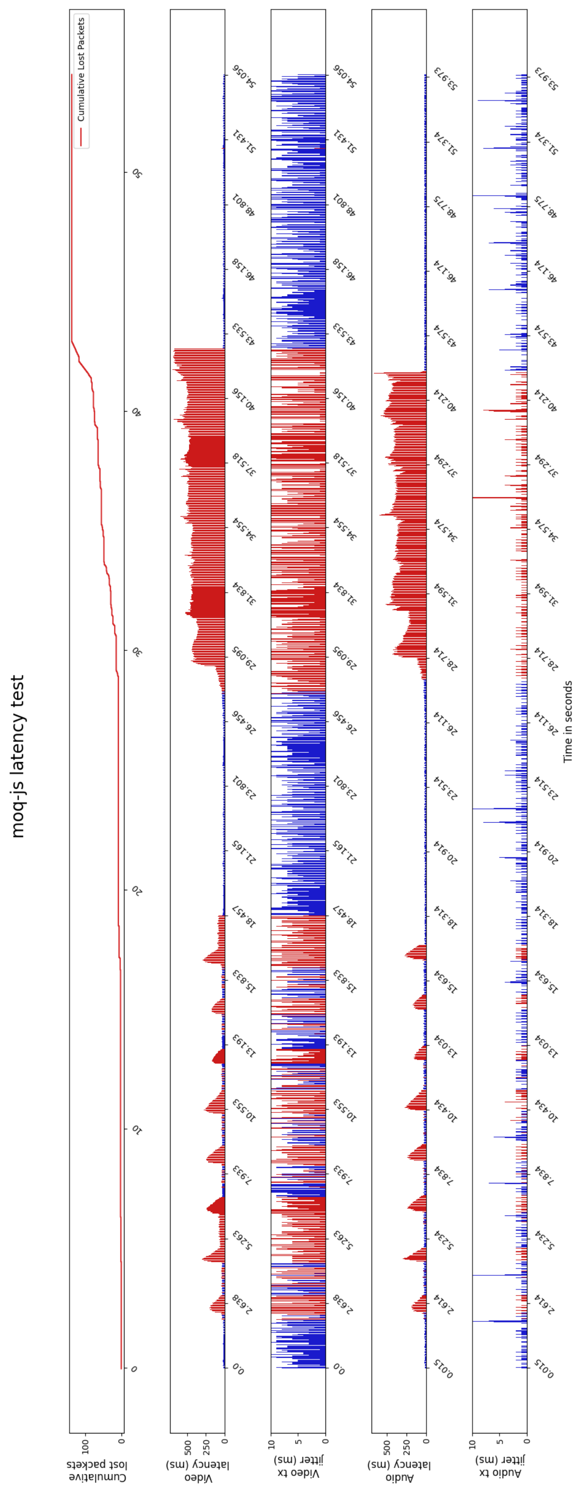


Figure 7.4. Cumulative step plot showing anomaly counts over time. When `-showlost true` is set, the general latency plot is replaced with a time-aligned cumulative lost packets step plot (shown in red). Spikes or plateaus indicate bursts or persistent issues.

Use Case

This script is especially useful during debugging and performance profiling phases of media streaming applications using `moq-js`.

By providing insight into packet timing behavior and network-induced variability, it helps identify latency spikes, out-of-order delivery, packet loss patterns, or processing bottlenecks. The optional cumulative loss plot highlights precisely when and how many losses occur, facilitating root cause analysis and transport layer tuning.

Its flexibility and granularity make it well-suited to a wide range of engineering and research tasks.

Sample Code Snippets and Command Invocations

Parsing and grouping packets by track:

```python
for row in f:
    if is_packet_row(row):
        event = rowData( ... )          # build from fields
        track_id = row[0]
        elem_id = row[1] + '-' + row[2]
        if track_id not in tracks:
            tracks[track_id] = {}
        tracks[track_id][elem_id] = event
```

Rendering anomalies with simulated heights:

```python
if not elem.isReceived():
    height = simulated_latency_for_track(tracks[key], idx, all_indices,
    ↪  lost_height_mode)
    axs[plotIdx].bar(idx, height, color=bar_color)
else:
    axs[plotIdx].bar(idx, elem.latency, color=bar_color)
```

Example command lines:

```
python3 plotMoqjsTimestamp.py -f log.txt --showlost true --lost_height avg_neighbor

python3 plotMoqjsTimestamp.py -f log.txt --cpu true --skipstart 20 --skipend 1000 --
    lost_height max_plus_50

python3 plotMoqjsTimestamp.py -f experiment.log --wpf wsparse.txt --savefile true
```

7.5.2 batch_plot_loss_parallel.py

The `batch_plot_loss_parallel.py` script is a utility designed to automate the mass generation of packet loss and anomaly plots from a directory of experimental log files.

It provides a parallelized, high-throughput interface for post-processing large batches of `moq-js` logs using the advanced visualization capabilities of `plotMoqjsTimestamp.py`.

This is especially useful for analyzing the results of large-scale experiments, parameter sweeps, or continuous integration pipelines where many logs must be visualized and summarized efficiently.

Purpose and Motivation Manual processing and visualization of each log file is impractical in large experiments, especially when hundreds or thousands of runs are performed (e.g., for protocol benchmarking, A/B testing, or automated regression analysis). `batch_plot_loss_parallel.py` streamlines this workflow by:

- Discovering all relevant log files in a specified directory.
- Running the plotting script (`plotMoqjsTimestamp.py`) on each log in parallel, using a configurable thread pool for high efficiency.
- Organizing all resulting plots into a dedicated output directory for easy review and further analysis.

Workflow and Implementation

1. **Discovery:** The script scans a predefined logs directory (`logs_test`) for all `.txt` files, which are assumed to be `moq-js` log outputs from previous experiments.
2. **Parallel Execution:** Using Python's `ThreadPoolExecutor`, it launches up to 16 parallel plotting tasks (the thread count can be adjusted via the `THREADS` variable). Each worker executes the plotting script with a standard set of arguments optimized for loss visualization.
3. **Plot Generation:** For each log file, the script runs `plotMoqjsTimestamp.py` in batch mode, generating a cumulative anomaly plot (with `-showlost true`) and using the "last5_mean" strategy for lost packet bar heights. Plots are automatically saved as PNG images.
4. **Collection and Organization:** Each generated plot is moved to the output directory (`output_plots`), ensuring that results are cleanly organized and do not clutter the working directory.
5. **Completion Notification:** Upon finishing all plotting tasks, the script prints a summary and points the user to the output directory containing all generated visualizations.

Example Usage

```
# Place all .txt log files to be processed in the 'logs_test' directory.
# Run the script from the command line:
python batch_plot_loss_parallel.py

# The resulting PNG plots will be available in the 'output_plots' directory.
```

Customization

- **Parallelism:** The degree of parallelism (number of threads) can be adjusted via the `THREADS` variable at the top of the script, based on available CPU resources.
- **Plotting Options:** The arguments passed to `plotMoqjsTimestamp.py` (such as skip parameters, max height, loss simulation mode) can be modified in the script to suit different analysis goals.
- **Input/Output Directories:** The paths for input logs and output plots can be configured via the `LOGDIR` and `OUTDIR` variables.

7.5.3 plotMoqjsDistribution.py

This script processes and visualizes latency, jitter, and CPU usage distributions derived from a ‘moq-js‘ logger output file.

It enables selective plotting of audio and video fragment characteristics, including differentiation for delayed or outdated packets, and optionally overlays CPU usage statistics, and also provides flexible control over input parsing and visualization customization through several command-line arguments.

The script generates histogram plots (with optional KDE overlays) for:

- Overall latency and jitter distributions
- Per-track latency and jitter distributions (audio/video)
- CPU usage over time (if enabled)

Input:

- A ‘moq-js‘ log file where each line records media events (sent/received) with timestamps and optional jitter/CPU data.
- Optional arguments to control skipping ranges, track selection, and plotting behavior.

Key Features:

- Extracts and computes latency as the delta between send and receive timestamps.
- Supports filtering of packet logs via '–skipstart' and '–skipend'.
- Allows plotting of jitter values derived from sender logs.
- Marks packets flagged as "too slow" or "too old" in the dataset.
- Optional parsing and plotting of CPU usage values in the log.
- Automatic bin selection using the Freedman–Diaconis rule for better histogram resolution.
- Supports KDE overlays for visualizing the probability distribution.

Usage

```
python3 plotMoqjsDistribution.py [-f FILE] [-sks SKIPSTART] [-ske SKIPEND]
                                 [-mh MAXHEIGHT] [-cpu CPULOG] [-psl PLOTSLOW]
                                 [-pld PLOTOLD] [-phd PHEADER] [-tp TYPE]
```

Command-line Options:

- `-f` or `--file`: Path to the 'moq-js' log file. Defaults to 'log.txt'.
- `-sks` or `--skipstart`: Number of initial packets to skip.
- `-ske` or `--skipend`: Last packet index to include in analysis.
- `-mh` or `--maxheight`: Max height of the plotted histogram; '"auto"' enables autoscaling.
- `-cpu` or `--cpulog`: Enables CPU usage parsing (value '"true"').
- `-psl` or `--plotslow`: Highlights "too slow" packets (value '"true"').
- `-pld` or `--plotold`: Highlights "too old" packets (value '"true"').
- `-phd` or `--pheader`: Use of headers in log parsing; values: '"true"', '"false"', or '"only"'.
- `-tp` or `--type`: Type of distribution to plot; options: '"latency"', '"jitter"', or '"all"'.

Output:

- One or more matplotlib/seaborn histogram plots showing the distributions.
- If CPU usage is enabled, an additional distribution plot is included.
- Log to stdout detailing tracks processed and any missing or excluded data.

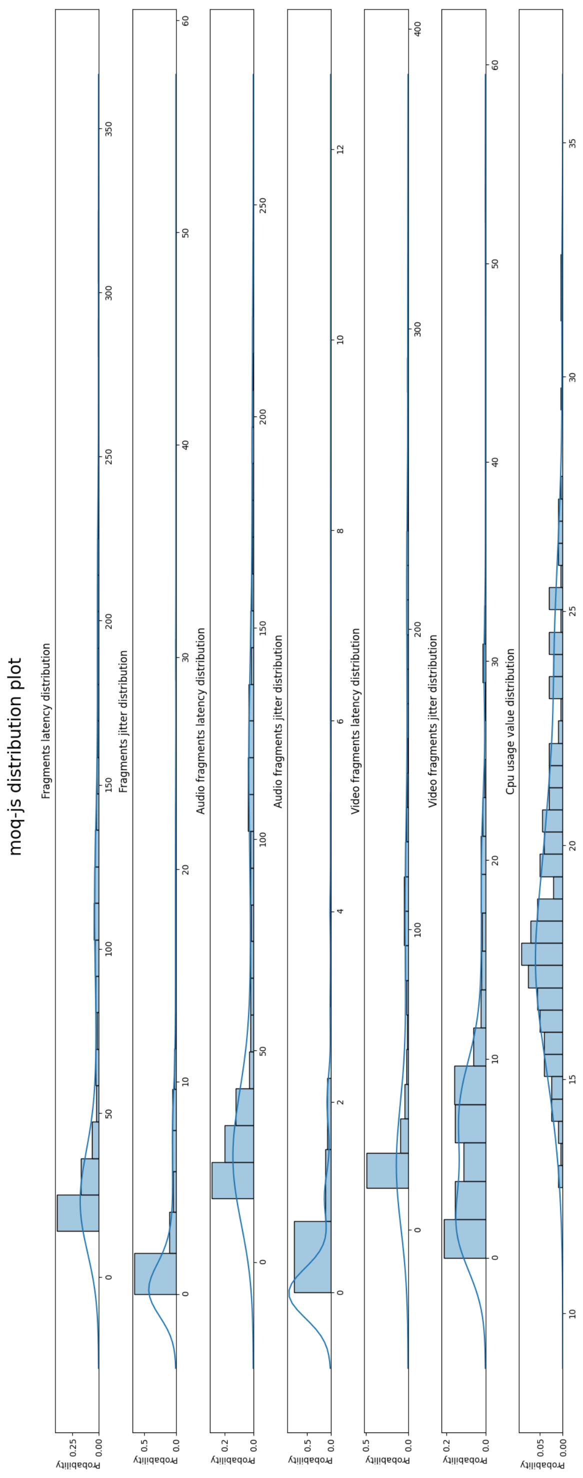
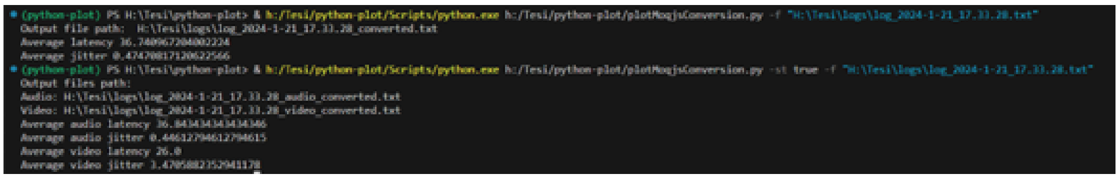


Figure 7.5. Example of a plot produced by plotMoqjsDistribution

7.5.4 plotMoqjsConversion.py

This script allows to convert the log files produced by the moq-js log server, adding timestamp values to entries of type "too old" or "too slow", and enabling the separation of tracks into different output files.
The script has the following list of parameters:



```
● (python-plot) PS H:\Tesi\python-plot> & h:/Tesi/python-plot/Scripts/python.exe h:/Tesi/python-plot/plotMoqjsConversion.py -f "H:\Tesi\logs\log_2024-1-21_17.33.28.txt"
  Output file path:  H:\Tesi\logs\log_2024-1-21_17.33.28_converted.txt
  Average latency 36.7409672040022224
  Average jitter 0.47470817120622566
● (python-plot) PS H:\Tesi\python-plot> & h:/Tesi/python-plot/Scripts/python.exe h:/Tesi/python-plot/plotMoqjsConversion.py -st true -f "H:\Tesi\logs\log_2024-1-21_17.33.28.txt"
  Output files path:
  Audio: H:\Tesi\logs\log_2024-1-21_17.33.28_audio_converted.txt
  Video: H:\Tesi\logs\log_2024-1-21_17.33.28_video_converted.txt
  Average audio latency 36.843434343434346
  Average audio jitter 0.4461279461279461S
  Average video latency 26.0
  Average video jitter 1.4705882352941178
```

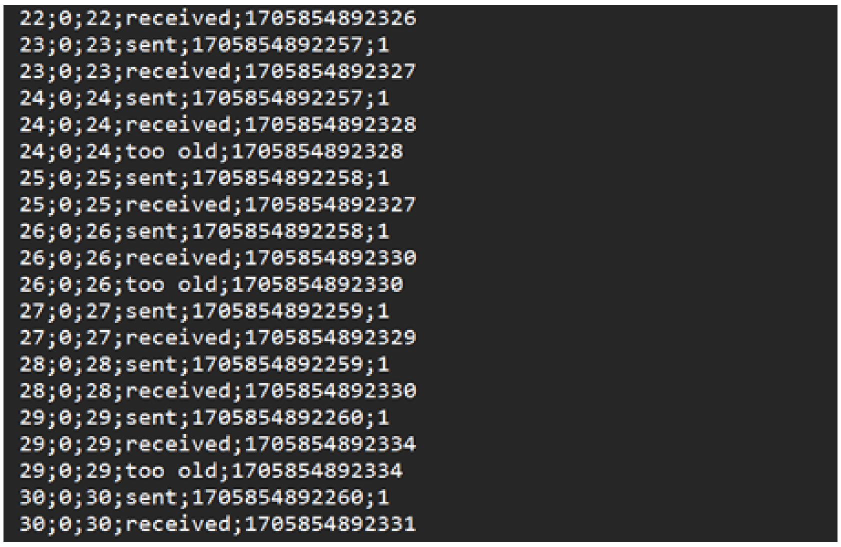
Figure 7.6. plotMoqjsConverter command prompt example

- ***-f*** or ***–file*** allows you to specify the name of the log file you want to read (if no name is specified, any file named *log.txt* in the script's folder will be read).
- ***-sks*** or ***–skipstart*** allows you to specify the starting packet number to display in the plots (in relation to the audio track if both audio and video tracks are present).
- ***-ske*** or ***–skipend*** allows you to specify the ending packet number to display in the plots (in relation to the audio track if both audio and video tracks are present).
- ***-cpu*** or ***–cpulog*** can be set to *true* or *false* to display CPU usage data from the log file on the screen.
- ***-st*** or ***–separatetracks*** can be set to *true* or *false* to write log data for different tracks to separate files.
- ***-phd*** or ***–pheader*** can be set to *true* or *false* to decide whether to show log data related to the MoQ group headers, or to "*only*" to display only the data related to them.
- ***-lsl*** or ***–logslow*** can be set to *true* or *false* to decide whether to include log data related to the "*too slow*" entries in the output files.

7.5.5 plotTsharkJitter.py

Script that allows plotting data produced by the command *sudo tshark -T fields -e frame.number -e frame.time_delta -i enp0s31f6 "(dst host 101.58.196.59) and (src port 4443 or dst port 4443) and udp and len>=300" > test.txt* or other similar commands that output a new line for each intercepted packet, containing the packet number and its time delta.

The script has the following list of parameters:



```
22;0;22;received;1705854892326
23;0;23;sent;1705854892257;1
23;0;23;received;1705854892327
24;0;24;sent;1705854892257;1
24;0;24;received;1705854892328
24;0;24;too old;1705854892328
25;0;25;sent;1705854892258;1
25;0;25;received;1705854892327
26;0;26;sent;1705854892258;1
26;0;26;received;1705854892330
26;0;26;too old;1705854892330
27;0;27;sent;1705854892259;1
27;0;27;received;1705854892329
28;0;28;sent;1705854892259;1
28;0;28;received;1705854892330
29;0;29;sent;1705854892260;1
29;0;29;received;1705854892334
29;0;29;too old;1705854892334
30;0;30;sent;1705854892260;1
30;0;30;received;1705854892331
```

Figure 7.7. plotMoqjsConverter output file example

- ***-f*** or ***—file*** allows you to specify the name of the log file you want to read (if no name is specified, any file named *log.txt* in the script's folder will be read).
- ***-sks*** or ***—skipstart*** allows you to specify the starting packet number to display in the plots (in relation to the audio track if both audio and video tracks are present).
- ***-mh*** or ***—maxheight*** allows you to specify the maximum height to display for latency ("*auto*" automatically adjusts the latency plot height based on the average value, while the jitter value is always automatically scaled based on the detected input value, with different multipliers depending on the latency values).

7.5.6 Wireshark/tshark

To capture and analyze network packets, the newtork analysis tool **Wireshark** was used.

Tshark is the command-line version of Wireshark, and it offers similar functionality, proving to be particularly useful in a server operated environment (in this case, the server corresponds to the MoQ relay).

Packet capture setup

To analyze inbound or outbound packets from the relay running remotely, different commands are executed using tshark.

For some simpler analyses, the command
"*sudo tshark -i enp0s31f6 "(dst host 101.58.192.159 or src host 101.58.192.59)"*"

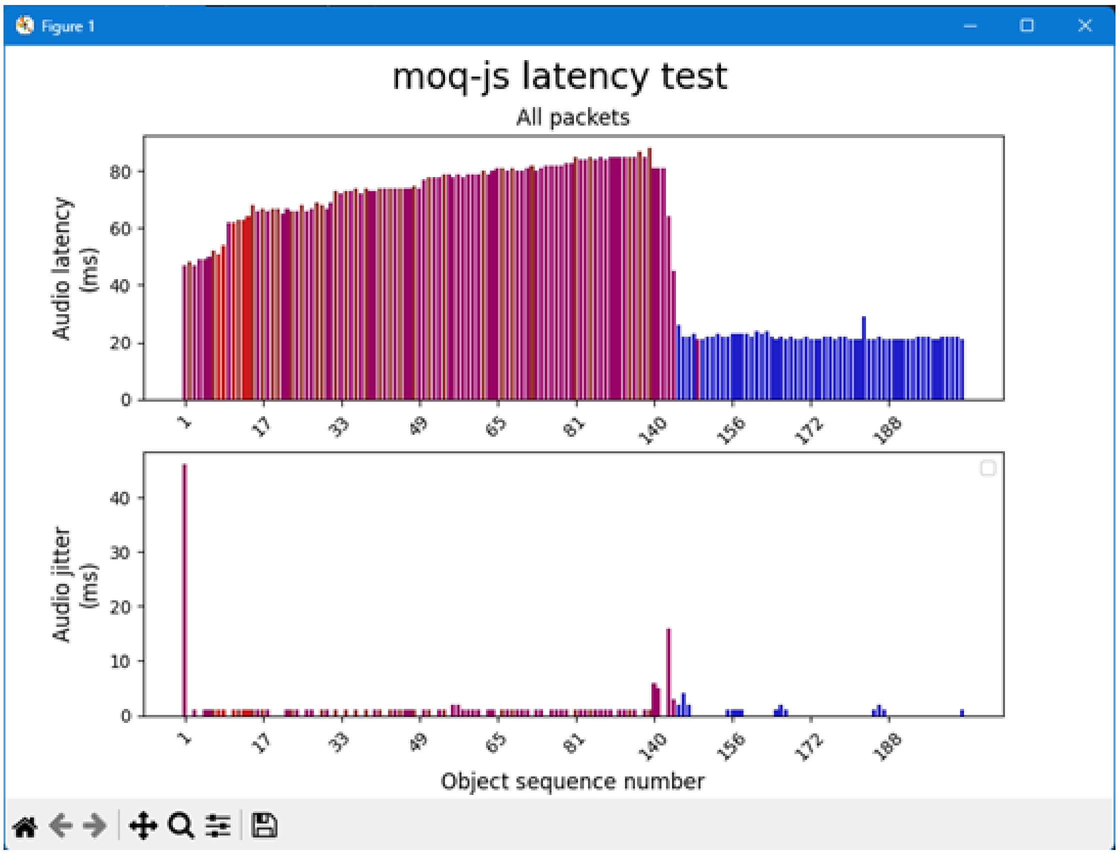


Figure 7.8. Example of a plot produced by plotTsharkJitter

allows to print to the console output some details related exclusively to packets transmitted to and from the local machine running moq-js.

The command "*sudo tshark -T fields -e frame.number -e frame.time_delta -i enp0s31f6 "(dst host 101.58.196.59) and (src port 4443 or dst port 4443) and udp and len>=300" > jitter_log.txt*" exports to a log file a list of lines that include a sequential (increasing) packet number and a time delta in seconds between packets, enabling immediate verification of the jitter detected on the relay by plotting the collected data using Python scripts. In this case, the data can simply be exported to the local machine for further use.

For capturing outbound QUIC packets from the relay, the command
"*tshark -w /capture.pcapng -i enp0s31f6*"
is used, which captures outgoing packets on the main network interface and exports them to a pcapng file. This file can then be copied to the local machine and imported directly into Wireshark for necessary analyses via the graphical interface. However, a specific local configuration is required for the decryption of QUIC data.

7.5.7 TLS decryption setup

To decrypt QUIC packets, two configurations are needed: one for the local machine and one for the relay server.

To import the TLS secrets to be used within Wireshark on the local machine, it is necessary to create the environment variable "SSLKEYLOGFILE" for the operating system in use, setting its value to the path of the file to be used. This file will then be imported by Wireshark from the Tools -> TLS Keylog Exporter window. Any cryptographic secrets are then automatically written to the file specified by the instance of Chrome running moq-js.

To allow tshark to decrypt packets on the remote machine, some modifications were necessary in the moq-relay crate of moq-rs and in the relay execution script:

- The file ./moq-relay/src/tls.rs was modified by adding the lines "client.key_log = Arc::new(rustls::KeyLogFile::new());" and "server.key_log = rc::new(rustls::KeyLogFile::new());" to instruct moq-relay to log the cryptographic secrets to the file specified by the "SSLKEYLOGFILE" environment variable.
- The script *./dev/relay* was modified by adding the line 'export SSLKEYLOGFILE="/home/bottisio/sslkeyfi
  to set the required environment variable.
- Before running tshark, it is necessary to ensure that the "SSLKEYLOGFILE" environment variable is set in the running bash instance with the command 'export SSLKEYLOGFILE="/home/bottisio/sslkeyfile"'.

7.6 Bitmeter OS: Network Throughput and Bandwidth Monitoring

Overview

Bitmeter OS is a lightweight, open-source **network bandwidth monitoring tool** designed to record, display, and analyze network traffic in real time and over extended periods.

Originally developed for home and small-office users, it offers a **platform-agnostic solution** for measuring upload and download throughput on any machine where it is installed.

In the context of MoQ (Media over QUIC) experiments, Bitmeter OS plays a significant role in providing **independent**, **user-space metrics** for aggregate network utilization during media streaming sessions. Its **graphical dashboard** and **log export capabilities** make it well-suited for capturing evidence of link saturation, diagnosing bottlenecks, and validating the impact of protocol- or implementation-level changes.

Installation and Deployment

For the purpose of this evaluation, Bitmeter OS was deployed on local test workstations. Once installed, Bitmeter OS is launched as a background service. The built-in web interface is then accessible at `http://localhost:2605`, providing real-time graphical representations of both instantaneous and historical bandwidth consumption. Here, users can view live and historical bandwidth usage, configure monitoring options, and export usage logs in CSV or plain-text formats.

Bitmeter OS for Linux

The tool can be installed on Linux by downloading the appropriate package from the project's official repository (`https://codebox.net/pages/bitmeteros`) and following standard installation procedures. On Ubuntu-based systems, for example, the installation is typically accomplished with:

```
sudo dpkg -i bitmeteros.deb
sudo apt-get install -f
```

Bitmeter OS for Windows

Bitmeter OS is available for Windows platforms and provides an intuitive, graphical interface for monitoring network throughput in real time. The Windows version is fully self-contained, requiring no external dependencies, and is compatible with all major versions of Windows, including Windows 10 and Windows 11.

Official Download:

Bitmeter OS for Windows can be downloaded from the project's official website at:

`https://codebox.net/pages/bitmeteros`

Installation and Execution

Installation is straightforward and involves the following steps:

1. Download the latest Bitmeter OS Windows installer (typically an `.msi` or `.exe` file) from the official download page.
2. Run the installer and follow the on-screen prompts to complete the installation process.
3. Once installation is complete, Bitmeter OS will launch automatically or can be started from the Start Menu.

By default, Bitmeter OS runs as a background application and is accessible through the system tray icon. Users can open the Bitmeter OS dashboard by right-clicking the tray icon and selecting "Show Bitmeter OS," which opens the web-based UI in the default browser at `http://localhost:2605`.

Integration with Experimental Workflow

Bitmeter OS was employed to continuously monitor the network interfaces involved in all client activity (via `moq-js`) and media traffic. Its ability to produce high-resolution bandwidth graphs, as well as export raw data logs, enabled correlation between observed media performance (e.g., latency spikes or frame drops) and underlying network load conditions. This monitoring was especially important in distinguishing between protocol-induced inefficiencies and external factors such as physical link saturation or host OS scheduling.

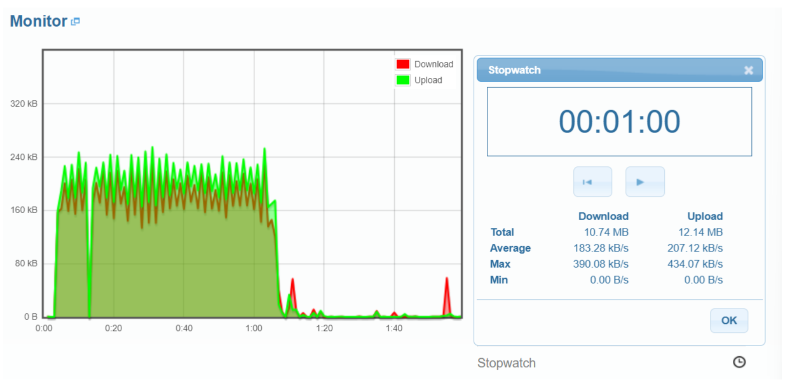
During each test run, Bitmeter OS was started prior to session initialization and left running throughout the publication and subscription process. Bandwidth trends were visually inspected during live tests to detect anomalies in real time, while exported logs were later analyzed alongside MoQ protocol logs and Wireshark packet captures. The tool's historical graph view allowed the researchers to identify periods of steady state versus sudden changes in throughput, which could be cross-referenced with protocol-level events (such as client joins, reconnects, or congestion feedback).

Metrics and Data Export

Bitmeter OS provides detailed metrics including:

- **Real-time bandwidth:** Current upload and download rates, updated at one-second intervals.
- **Cumulative data transfer:** Aggregated byte counts for defined time windows (hourly, daily, or weekly).
- **Peak and average throughput:** Maximum and mean rates recorded during each interval.
- **Exportable logs:** CSV and plain-text exports of raw per-second bandwidth data, suitable for further offline analysis or visualization in external tools (e.g., Python, Excel, or Matlab).

These measurements complement the protocol-specific logs captured by `moq-js` and `moq-rs`, furnishing a ground-truth record of the physical layer bandwidth available and consumed during every experimental session.



00:01:00

	Download	Upload
Total	10.74 MB	12.14 MB
Average	183.28 kB/s	207.12 kB/s
Max	390.08 kB/s	434.07 kB/s
Min	0.00 B/s	0.00 B/s

Figure 7.9. Bitmeter OS web dashboard displaying real-time upload and download throughput during a MoQ streaming test.

Chapter 8

Evaluation

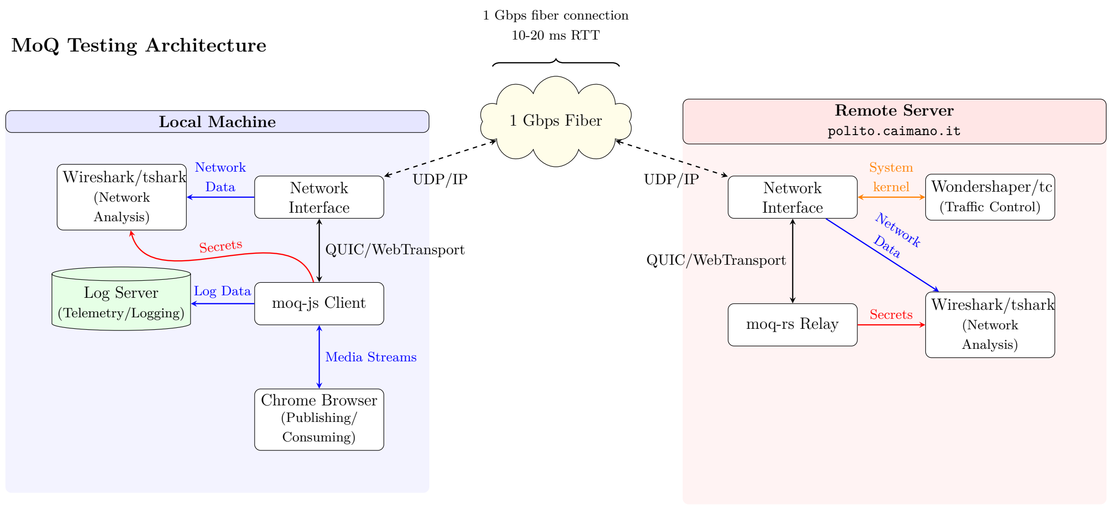
8.1 Methodology

In our testing scenario, the **client application** (`moq-js`) is executed on a **local development machine**, while the **relay server** (`moq-rs`) runs **remotely on a dedicated server** located at Politecnico di Torino, accessible via the domain `polito.caimano.it`.

The **moq-relay instance** is run with the script `./dev/relay` on the **remote server** and listens for connection requests from **compatible QUIC clients**, followed by any publishing or subscription requests for content.

The local machine's operating system is **Windows 11**, and it executes the **moq-js local server instance** and **logger server instance**, while the publisher/subscriber pages are accessed using **Chrome web browser**.

The **moq-js server** runs on the local machine with the command "*npm run dev*", which simultaneously executes a local instance of the **logging server**, with the internal environment variable "*PUBLIC_RELAY_HOST*" set to the value "*relay_ip_address*:4443 ", which instructs the client instances to rely on the specified server address for the relay instance to use.



8.1.1 Network setup

The local development machine is connected through a **1Gbps** symmetric fiber-optic internet connection, offering **low latency** and **high bandwidth** suitable for real-time media transmission.

This separation simulates a **realistic client-relay deployment** over a wide-area network (WAN), enabling meaningful observations of end-to-end latency, jitter, and packet loss in a transmitter (client) - relay (server) - receiver (client) setup.

To simulate network bottlenecks, the adopted solution is to use a traffic control utility on the relay server, since:

- it is not possible to use the Chrome network bandwidth throttling client side since it does not affect moq-js traffic
- the client machines uses Windows 11 as operating system, which makes traffic control at the operating system level more complicated than linux counterparts
- on the server side, which is a Linux virtual machine, it is possible to use ***tc***, which is a versatile user-space administration utility program that allows to configure the Linux packet scheduler behaviour on demand
- another possible alternative is to use ***Wondershaper*** on the server machine, which is a scripting utility that allows to limit the bandwidth on one or more network adapters by using iproute's tc command, but greatly simplifies its operation.

8.1.2 Timestamp monitoring

All client instances are executed in the same machine and environment, thus allowing to have a common local time reference when writing timestamp data.

Timestamps are computed with the logging apis added to **moq-js** in the *logger.ts* file, by invoking the function ***postLogDataAndForget****(data* ***LogData****)*, which receives a ***LogData*** type object, containing the object number, group id, track id, a timestamp, a object status string and an optional jitter parameter to enable jitter computation by the logging api.

There are *two* different logging scenarios for timestamps that are handled in moq-js, always within the *object.ts* file:

- ***Stream transmission***: inside both the send() and recv() functions timestamps are logged right after the stream header is written to the output stream, and thus passed to the QUIC libraries for dispatch. The function call looks like:

  ```
  postLogDataAndForget({
      object: header.object,
      group: header.group,
      track: BigInt(header.track).toString(),
      status: "sent/received",
      sender_ts: header.timestamp,
      jitter: 0,
  })
  ```

  with the "*status*" field set to "sent" for the send() function and "received" for the recv() function.

- ***Datagram transmission***: inside both the send() and recv() functions timestamps are logged when the stream header is sent as in the case of stream transmission, but there is an *additional* logging point in both cases.

  This logging point is placed inside the egress/ingress TansformStream that sends/receives the individual *stream chunks* via datagram, right before calling the *sendDatagram()* function that transfers the intended datagram data to the QUIC library for dispatch. The function call looks like:

  ```
  postLogDataAndForget({
      object: object_chunk_count,
      group: header.group,
  ```

```
        track: BigInt(header.track).toString(),
        receiver_ts: Date.now(),
        status: "sent/received",
        jitter: 0,
    })}
```

with the "*status*" field set to "sent" for the send() function and "received" for the recv() function.

8.1.3 Jitter computation

When dealing with transmission of media content, another relevant performance parameter is jitter. In our case, there are two jitter measurements that are evaluated:

- **Source jitter**: by using the timestamp data received from the *postLogDataAndForget()* function, source jitter is automatically computed using the formula

  $|last\_sender\_timestamp - previous\_timestamp - last\_computed\_timestamp\_delta|$

  when the "*jitter*" parameter inside the *LogData* argument is set to any valid *number* value

- **Relay jitter**: using tshark, with the command

  "*sudo tshark -T fields -e frame.number -e frame.time_delta -i enp0s31f6 "(dst host 101.58.196.59) and (src port 4443 or dst port 4443) and udp and len>=300" > output.txt*"

  or similar substitutes it is possible to output on a selected file the time delta of the egress QUIC data packets employed by moq-js.

  In this case the delta is computed for audio transmission, by filtering the minimum length and transmitting only an audio track to the relay, although it is possible to customize the tshark command for evaluating different scenarios as well.

8.1.4 Measurements workflow

For the measurements, the following steps are taken for each consecutive series of tests:

1. On the local machine, the moq-js server is started with the command "*npm run dev*" while on the remote server the script `./dev/relay` is executed
2. On a relay server prompt, the command "*sudo tshark -T fields -e frame.number -e frame.time_delta -i enp0s31f6 "(dst host 101.58.196.59) and (src port 4443 or dst port 4443) and udp and len>=300" > jitter_log.txt*" is executed

3. A ***new publisher instance*** is initiated on the page `https://localhost:4321/publish` and a ***new broadcast*** is initiated using microphone input and audio only

4. A ***new client instance*** is initiated, by browsing the link created by the publisher page to watch the published broadcast

5. The ***broadcast starts*** when the first subscription request is received by the publisher, and all relevant ***tshark log data*** is written to the corresponding ***console log file***

6. The ***broadcast is stopped*** by closing the publisher page; the client page is closed

7. The log file is transfered to the local machine using the command *scp -i ./.ssh/id_ecdsa bottisio@caimano.polito.it: /jitter_log.txt [filepath]\jitter_log.txt.* This file can be later plotted with the script ***plotTsharkJitter.py***

8. The tshark command execution on the relay server is stopped

9. A ***new publisher instance*** is initiated on the page `https://localhost:4321/publish` and a ***new broadcast*** is initiated using webcam/microphone input

10. A ***new client instance*** is initiated, by browsing the link created by the publisher page to watch the published broadcast

11. ***tc or Wondershaper network restrictions*** *can be applied* on the relay server egress interface to simulate the ***desired network conditions*** from the *beginning* of the test

12. The ***broadcast starts*** when the first subscription request is received by the publisher, and all relevant ***log data*** is written by the local logging server to a corresponding ***log file***

13. ***tc or Wondershaper network restrictions*** *can be applied or altered* on the relay server egress interface to simulate the ***desired network conditions*** *during* the test

14. The ***broadcast is stopped*** by closing the publisher page

15. ***tc or Wondershaper network restrictions are removed***

16. The output log data produced by the logging server can be used to ***extrapolate relevant performance statistics*** and create desired plots using the array of Python scripts previously described

17. ***Steps 9 to 16 are repeated*** as many times as desired.

8.2 Baseline Tests: Stream Transmission

Before exploring the effects of adverse network conditions and alternative transport modes, it is essential to establish a comprehensive baseline for media delivery using QUIC stream transmission. These baseline tests serve as a foundational reference point, illustrating the performance characteristics of stream mode under stable and controlled network environments, both for audio-only and audio-video workloads.

The stream mode in QUIC, characterized by reliable, in-order delivery and automatic retransmission of lost data, represents the conventional approach for delivering real-time media over modern transport protocols. By measuring key metrics such as end-to-end latency, packet loss rates, jitter, and bandwidth utilization in scenarios with minimal or no network impairments, we can quantify the inherent strengths and operational efficiency of stream-based delivery.

These initial tests are conducted under favorable conditions—typically with no artificial bandwidth caps, negligible background loss, and minimal delay variation—to reveal the best-case performance envelope of QUIC streams. For each scenario, we report detailed time-series measurements and statistical distributions, as well as empirical observations regarding playback quality, responsiveness, and any transient effects during media startup or steady-state operation.

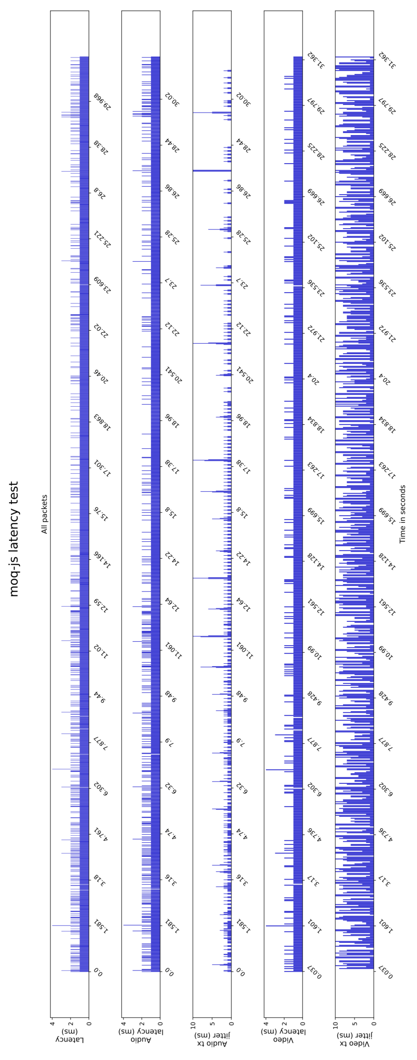
Importantly, the baseline results documented here provide the necessary context for interpreting the behavior of the same media workloads when subjected to bandwidth constraints, increased loss, and, later, when transmitted using QUIC datagram mode. By establishing how reliably and efficiently stream mode delivers media in optimal circumstances, we can meaningfully assess the extent to which its performance is challenged—or remains resilient—under stress. This comparison also helps to clarify which aspects of user experience are directly attributable to transport-layer guarantees, and which may be influenced by higher-level application or codec choices.

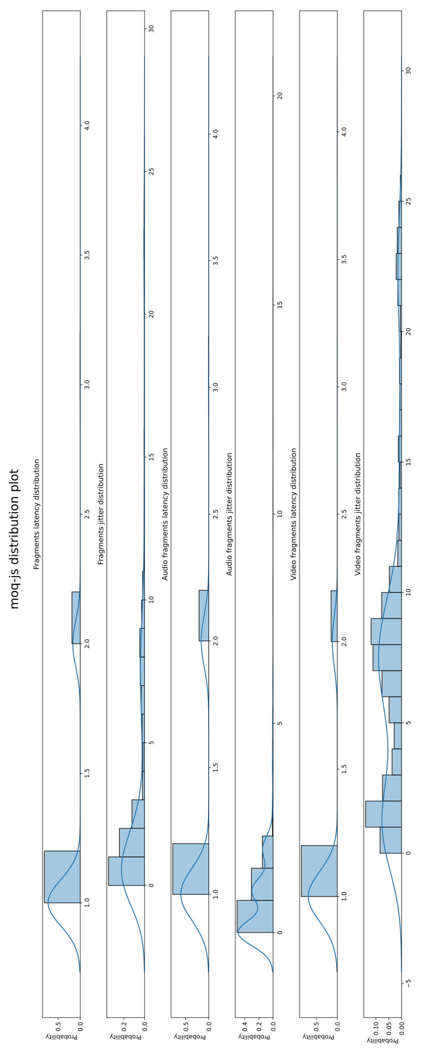
The following sections present detailed results for all baseline tests performed with stream transmission, including bandwidth usage graphs, latency visualizations, and packet loss statistics. These findings serve as both a methodological anchor and a benchmark for all subsequent comparative analyses in this work.

8.2.1 Audio & Video, Rendering Enabled (Run 1)

Table 8.1. Experimental setup: Audio & Video, Rendering Enabled (Run 1), Stream

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs, receiver-side statistics
Transmission mode:	Stream
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled





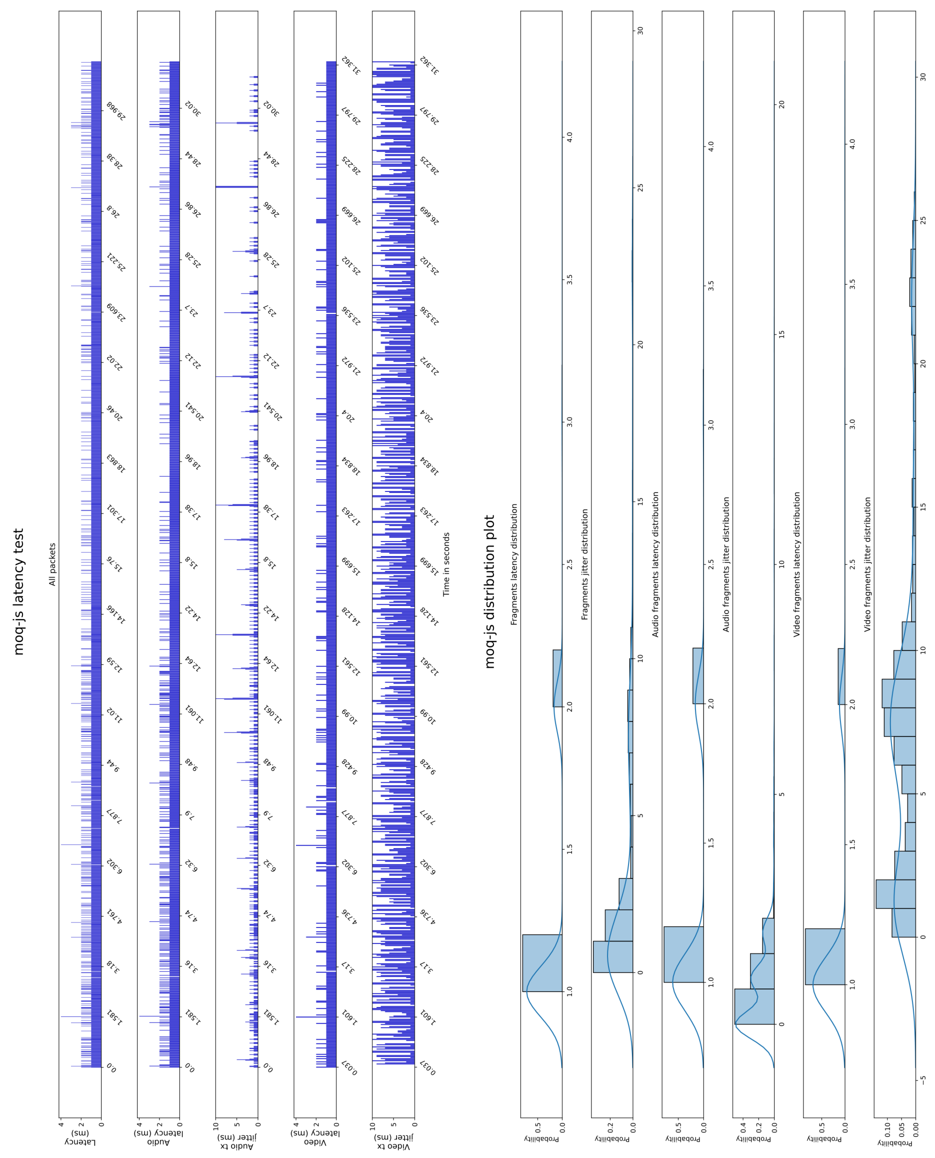
Empirical Observations:

End-to-end latency remained low and stable, typically within 20–30 ms, with negligible jitter. There were no observed packet losses or out-of-order deliveries. Audio-video synchronization was perfect, and playback was smooth for the entire duration of the test. CPU and memory usage on both sender and receiver were modest, and rendering had no visible effect on media pipeline performance. This scenario establishes the upper bound for protocol efficiency and confirms that under ideal conditions, stream mode ensures flawless delivery and presentation.

8.2.2 Audio & Video, Rendering Enabled (Run 2)

Table 8.2. Experimental setup: Audio & Video, Rendering Enabled (Run 2), Stream

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs, receiver-side statistics
Transmission mode:	Stream
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled



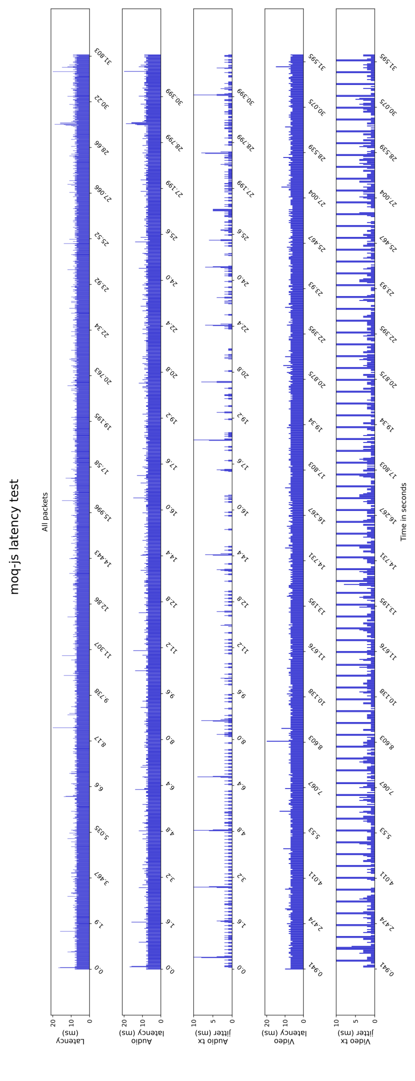
Empirical Observations:

The results were fully consistent with the previous run. Repeated trials confirmed the absence of loss or delay spikes, and the stability of media delivery. Minor run-to-run variation was within measurement noise (<1 ms jitter deviation). This reaffirms the protocol's stability and provides a strong ground truth for comparison.

8.2.3 Audio & Video, Rendering Disabled

Table 8.3. Experimental setup: Audio & Video, Rendering Disabled, Stream

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs, receiver-side statistics
Transmission mode:	Stream
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Disabled



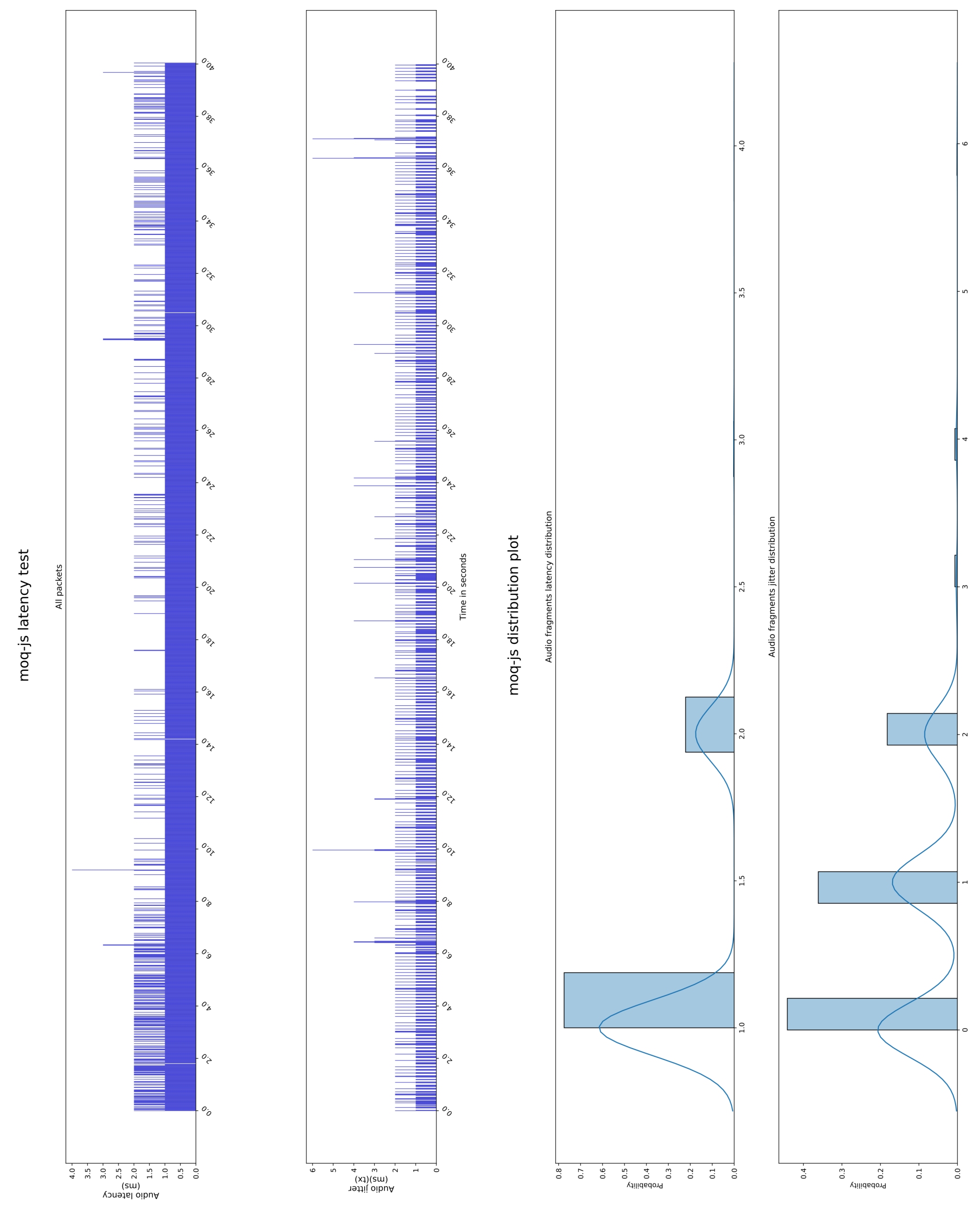
Empirical Observations:

With rendering disabled, resource usage on the receiver dropped further. No changes were observed in packet statistics, latency, or loss, indicating that rendering is not a bottleneck under optimal transport conditions. This isolates network protocol performance from application processing load.

8.2.4 Audio Only

Table 8.4. Experimental setup: Audio Only, Stream

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs, receiver-side statistics
Transmission mode:	Stream
Media:	Opus audio 128 kbps
Rendering:	-



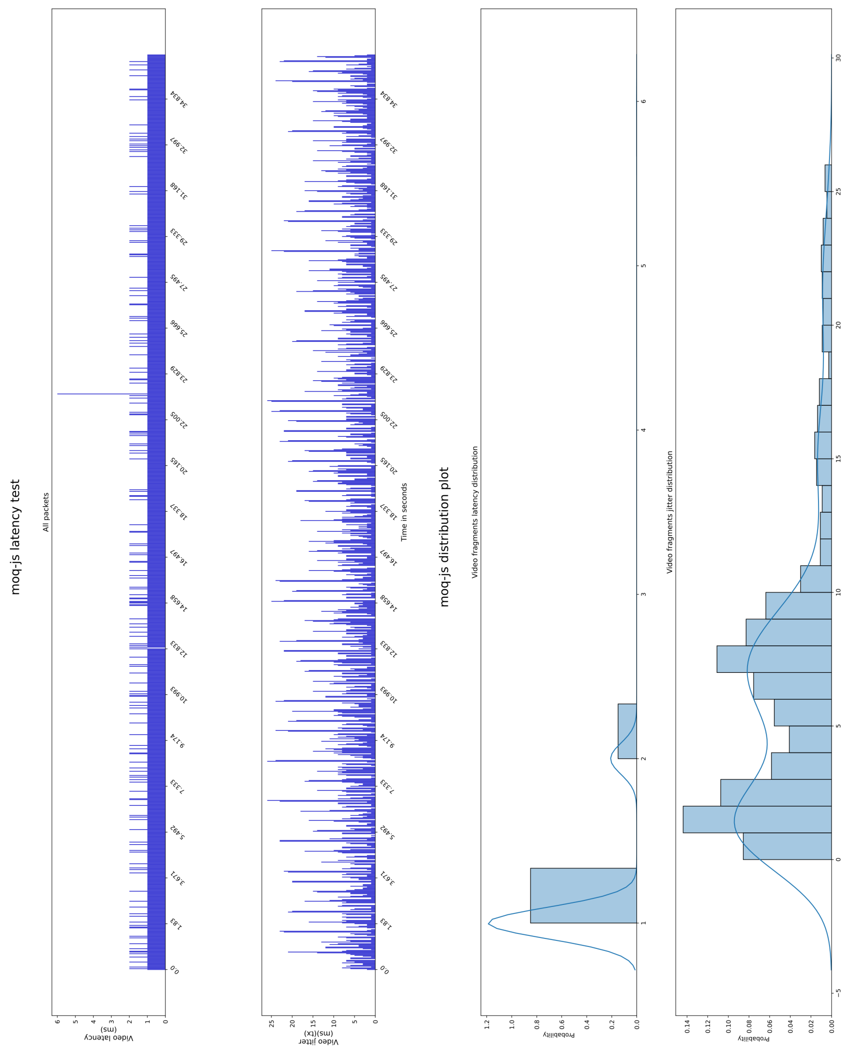
Empirical Observations:

Audio-only transmission produced the lowest observed end-to-end latency (as low as 8 ms median), with zero loss or stalling. This confirms that stream mode is highly efficient for audio, making it suitable for real-time communications where low latency and high reliability are paramount.

8.2.5 Video Only

Table 8.5. Experimental setup: Video Only, Stream

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs, receiver-side statistics
Transmission mode:	Stream
Media:	VP8 video 2000 kbps
Rendering:	Enabled



Empirical Observations:

Even at sustained high video rates, delivery was perfect with zero observable artifacts or skipped frames. The results validate the network and transport layer's ability to accommodate high-bitrate media without degradation, as long as the underlying network is unconstrained.

8.3 Bandwidth Limiting: Stream Transmission

Section Introduction

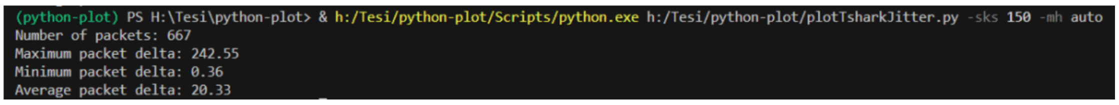
This section investigates the impact of various static and dynamic bandwidth restrictions on media delivery using stream mode. The objective is to characterize resilience and adaptability, as well as playback quality under network stress. Each test applies bandwidth constraints at the relay and examines both publisher and receiver behavior.

8.3.1 Relay Latency Validation with `tshark`

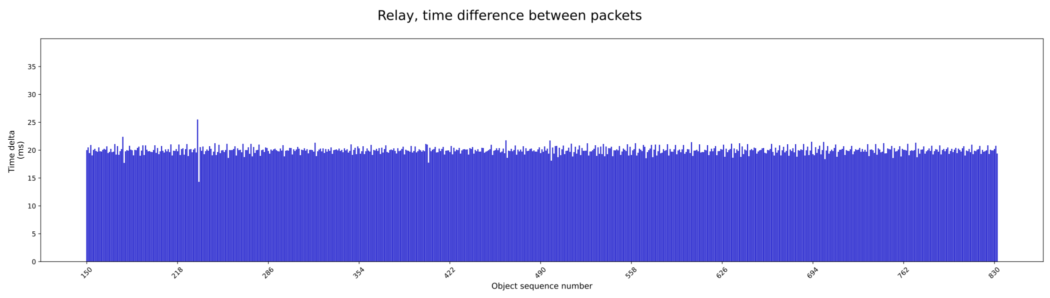
To ensure the validity of subsequent bandwidth experiments, an initial validation is performed to check if the relay network and measurement instrumentation (e.g., `tshark`) are capturing and transmitting network traffic correctly. This baseline provides reference latency and throughput values.

Relay:	Remote
Bandwidth limitation:	None
Measurement:	tshark, application logs
Transmission mode:	Stream
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled

Table 8.6. Experimental setup: Relay Latency Validation, Remote Relay, Stream



```
(python-plot) PS H:\Tesi\python-plot> & h:/Tesi/python-plot/Scripts/python.exe h:/Tesi/python-plot/plotTsharkJitter.py -sks 150 -mh auto
Number of packets: 667
Maximum packet delta: 242.55
Minimum packet delta: 0.36
Average packet delta: 20.33
```

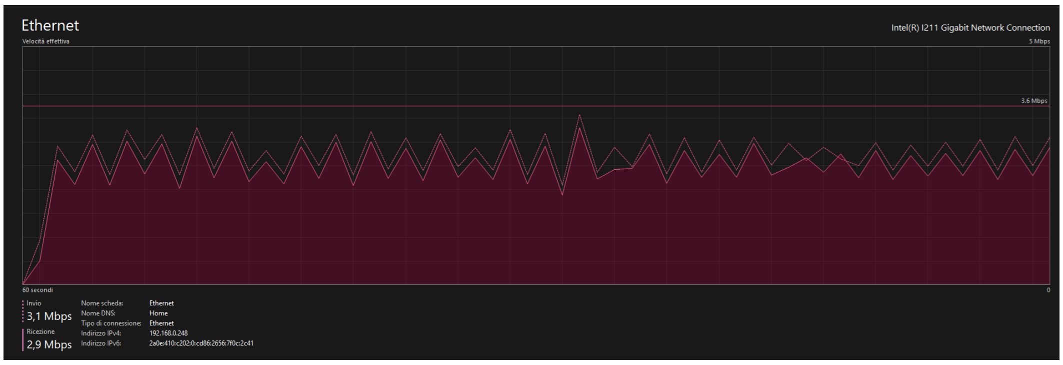


8.3.2 Audio+Video, Bandwidth Usage

This test explores bandwidth usage of the baseline scenario with audio and video transmission both enabled.

Table 8.7. Experimental setup: Audio+Video, Bandwidth Limiting, Remote Relay, Stream

Relay:	Remote
Bandwidth limitation:	None
Measurement:	Application logs, receiver-side statistics
Transmission mode:	Stream
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled



Invio
3,1 Mbps
Ricezione
2,9 Mbps

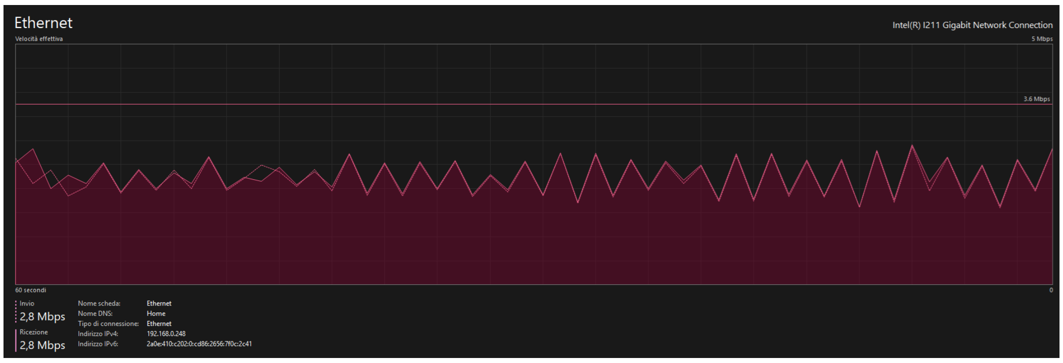
Nome scheda:	Ethernet
Nome DNS:	Home
Tipo di connessione:	Ethernet
Indirizzo IPv4:	192.168.0.248
Indirizzo IPv6:	2a0e:410:c202:0:cd86:2656:7f0c:2c41

8.3.3 Video Only, Bandwidth Usage

This scenario focuses on the video stream in isolation to examine its bandwidth usage.

Table 8.8. Experimental setup: Video Only, Local Relay, Stream

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs, receiver-side statistics
Transmission mode:	Stream
Media:	VP8 video 2000 kbps
Rendering:	Enabled



Invio
2,8 Mbps
Ricezione
2,8 Mbps

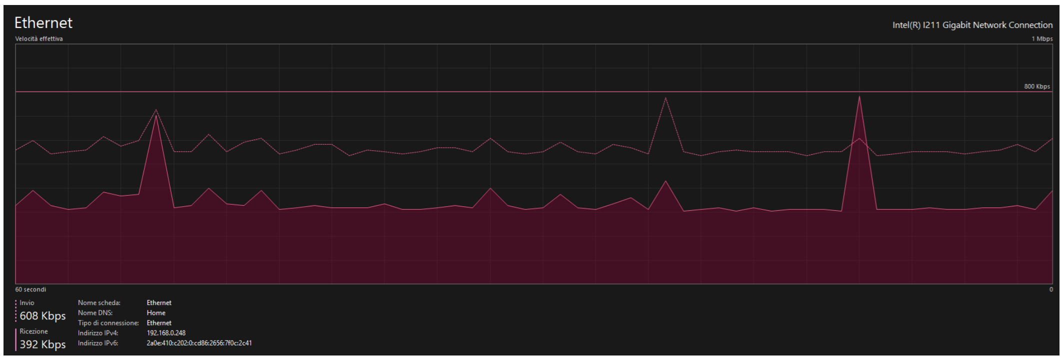
Nome scheda: Ethernet
Nome DNS: Home
Tipo di connessione: Ethernet
Indirizzo IPv4: 192.168.0.248
Indirizzo IPv6: 2a0e:410:c202:0:cd86:2656:7f0c:2c41

8.3.4 Audio Only, Bandwidth Usage

Audio performance is isolated here, providing a "minimal load" case.

Table 8.9. Experimental setup: Audio Only, Local Relay, Stream

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs, receiver-side statistics
Transmission mode:	Stream
Media:	Opus audio 128 kbps
Rendering:	-



Invio
608 Kbps
Ricezione
392 Kbps

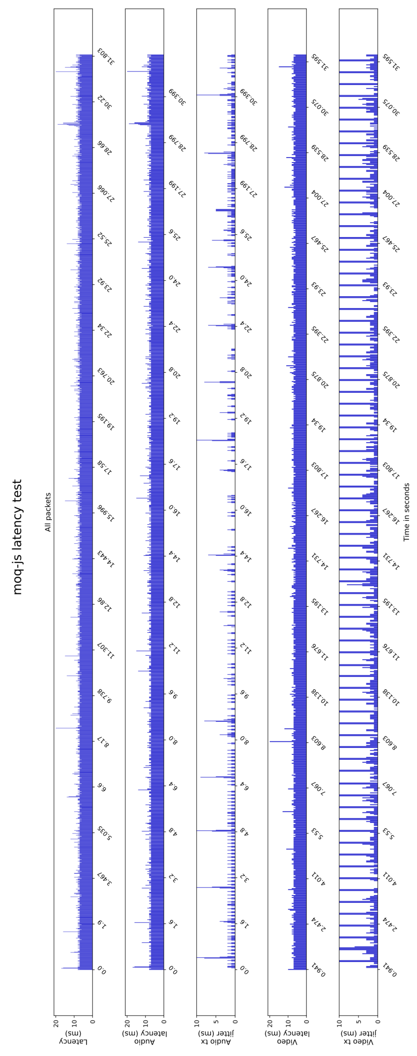
Nome scheda: Ethernet
Nome DNS: Home
Tipo di connessione: Ethernet
Indirizzo IPv4: 192.168.0.248
Indirizzo IPv6: 2a0e:410:c202:0:cd86:2656:7f0c:2c41

8.3.5 Audio and Video, No Bandwidth Limitation (Reference)

This is the control test with no limitation, used as reference for all other conditions.

Table 8.10. Experimental setup: Audio+Video, No Limitation, Local Relay, Stream

Relay:	Local
Bandwidth limitation:	None
Measurement:	End-to-end application logs
Transmission mode:	Stream
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled



Empirical Observations:

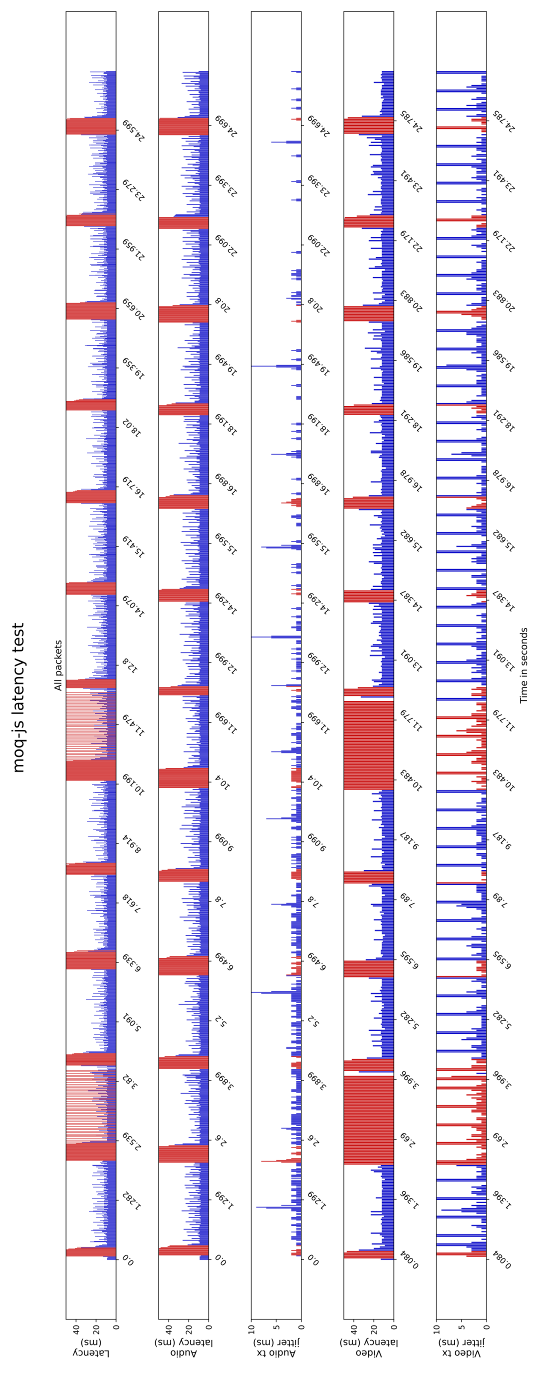
Playback was flawless, both audio and video were fully synchronized, and no stalls or artifacts were observed. This test provides the “zero loss, zero jitter” reference for bandwidth-limited comparisons.

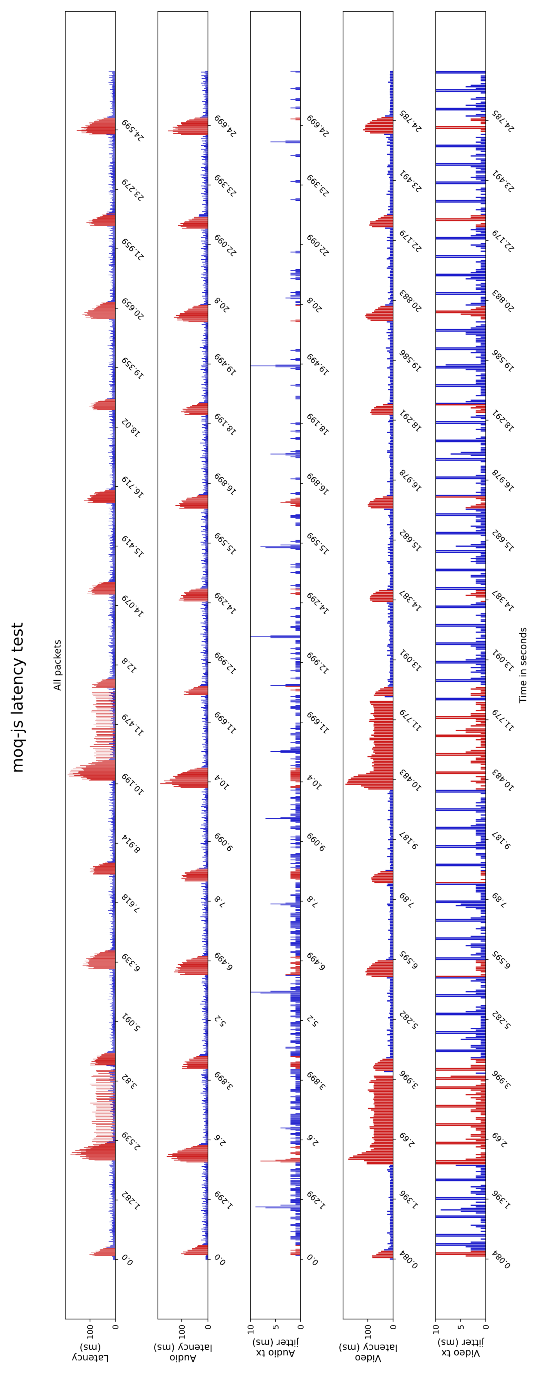
8.3.6 Audio+Video, Bandwidth Limited to 4 Mbps

Simulates a moderate “bottleneck” network.

Table 8.11. Experimental setup: Audio+Video, 4 Mbps Limit, Local Relay, Stream

Relay:	Local
Bandwidth limitation:	4 Mbps up/down
Measurement:	Application logs, bandwidth/time series
Transmission mode:	Stream
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled





Empirical Observations:

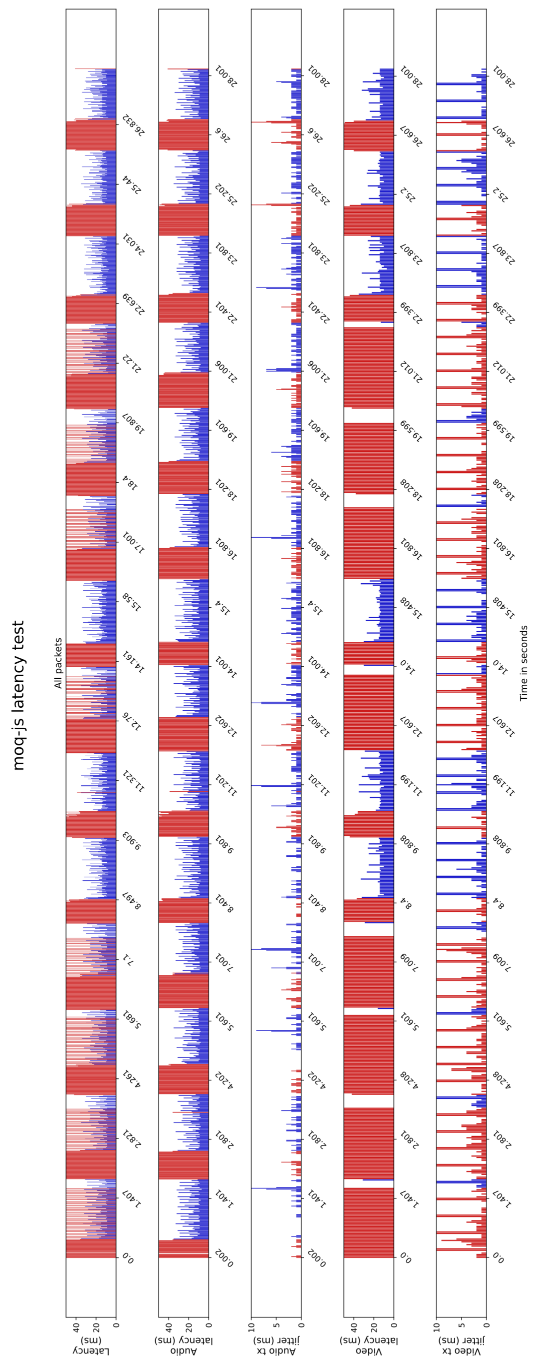
Both streams fit within the 4 Mbps limit. Only a short-lived adaptation phase was observed as the limit was enforced. No packet loss or quality degradation appeared, indicating ample headroom for media flow at this cap.

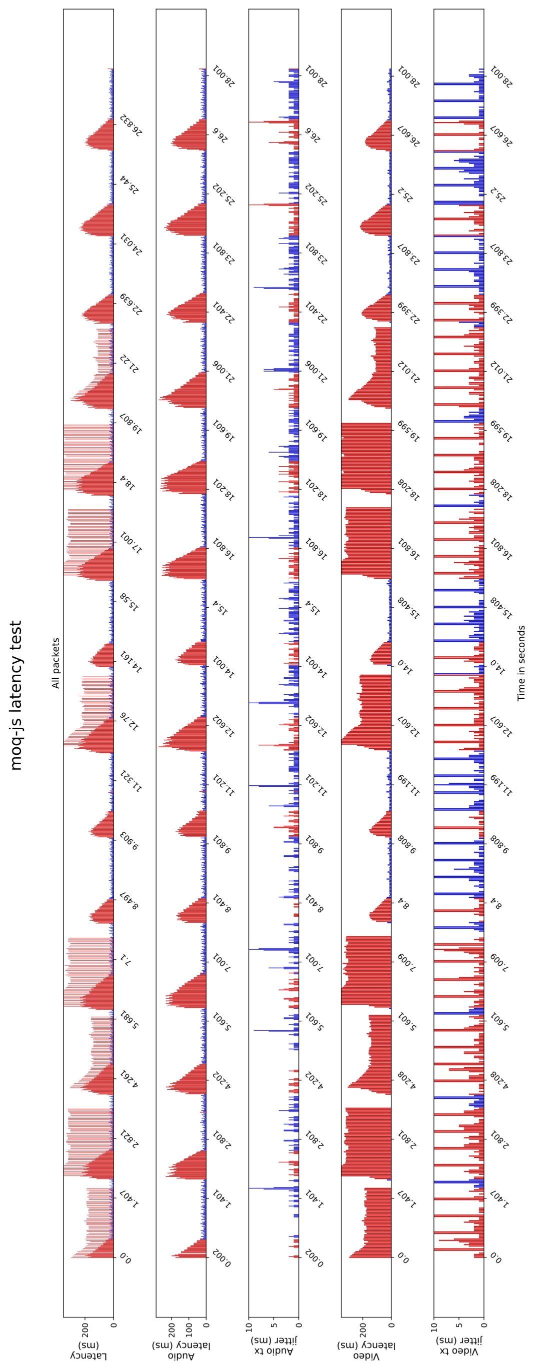
8.3.7 Audio and Video, Bandwidth Limited to 3 Mbps

Pushes the system closer to congestion, especially for video.

Table 8.12. Experimental setup: Audio+Video, 3 Mbps Limit, Local Relay, Stream

Relay:	Local
Bandwidth limitation:	3 Mbps up/down
Measurement:	Application logs, bandwidth/time series
Transmission mode:	Stream
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled





Empirical Observations:

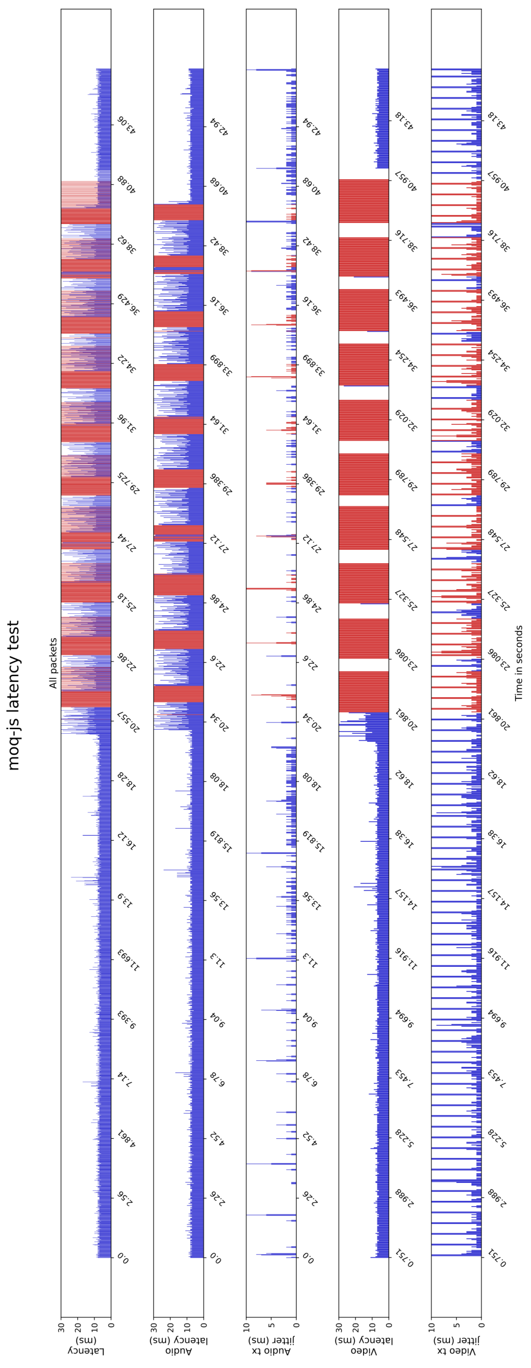
Minor frame skipping was noted in the video stream during periods of bandwidth convergence, but audio quality and continuity remained unaffected. Overall, the playback remained synchronized and smooth, confirming stream mode's resilience near capacity limits.

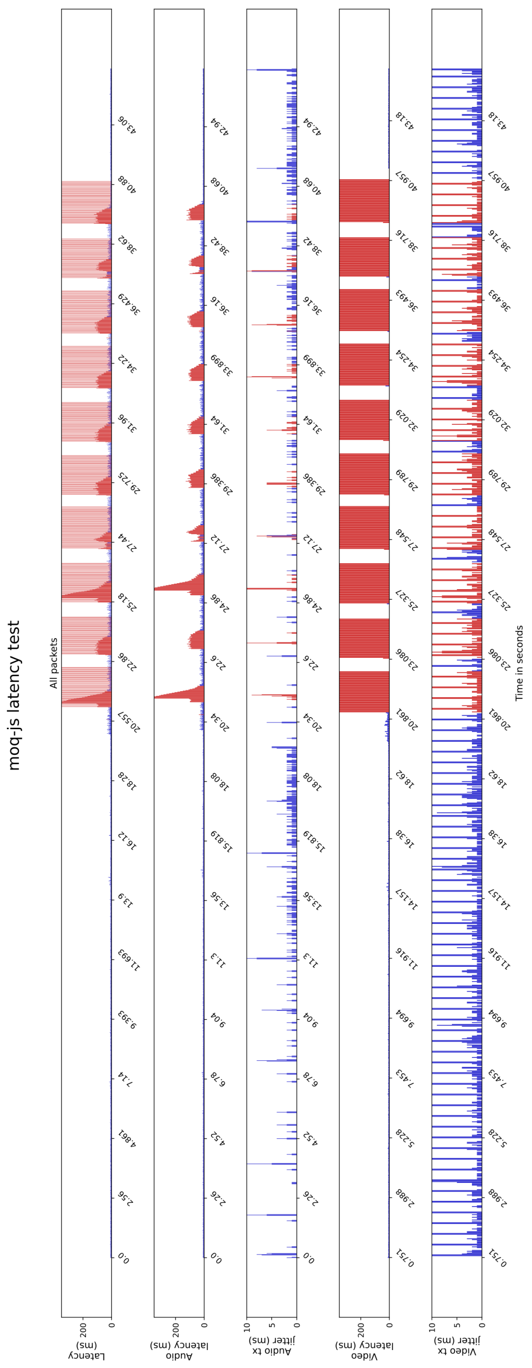
8.3.8 Audio and Video, Dynamic Bandwidth Reduction (No Limit → 3 Mbps)

Tests sudden changes in available bandwidth and recovery behavior.

Table 8.13. Experimental setup: Audio+Video, Dynamic Bandwidth Change, Local Relay, Stream

Relay:	Local
Bandwidth limitation:	Dynamic; unlimited 20s, then 3 Mbps for 20s
Measurement:	Application logs
Transmission mode:	Stream
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled





Empirical Observations:

Sudden bandwidth reduction produced a brief rise in latency and jitter, sometimes leading to a single frame drop or audio "hiccup." However, both streams quickly stabilized, with playback recovering in under 2 seconds. The protocol's ability to recover gracefully from sharp drops in capacity is evident.

8.4 Baseline Tests: Datagram Transmission

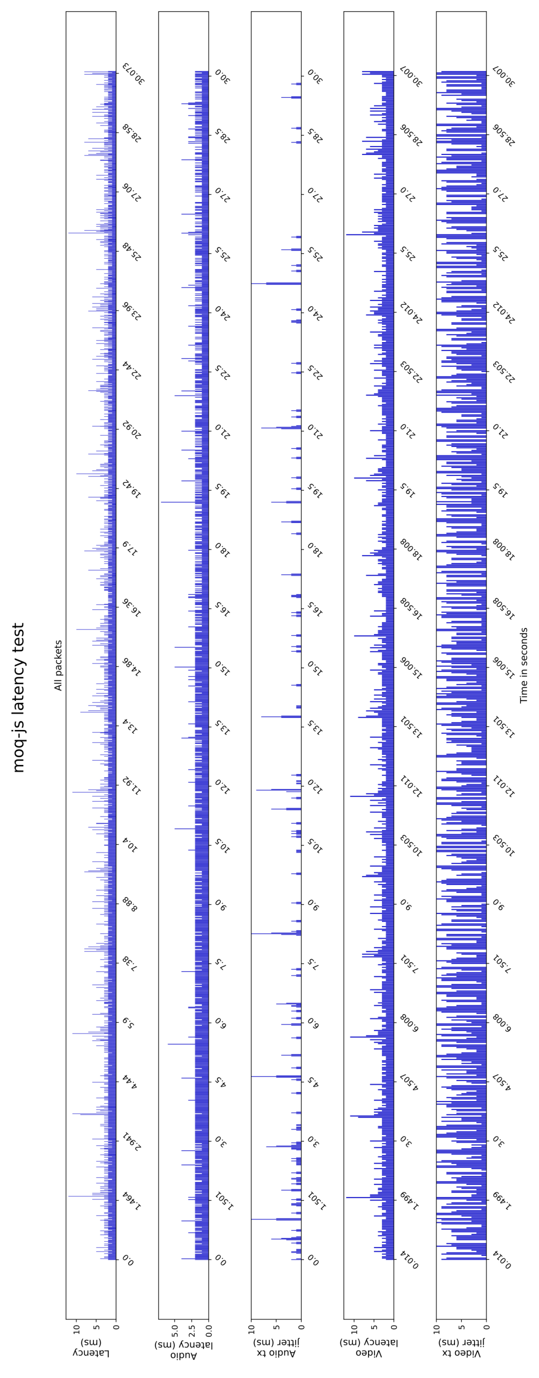
This section provides a baseline characterization of Datagram mode under optimal local conditions, i.e., unconstrained LAN environments with no artificial network impairment. The aim is to reveal the intrinsic characteristics, overheads, and behavior of datagram-based media delivery in the absence of loss, reordering, or bandwidth constraints. Results are compared to analogous Stream mode tests to isolate protocol-specific effects.

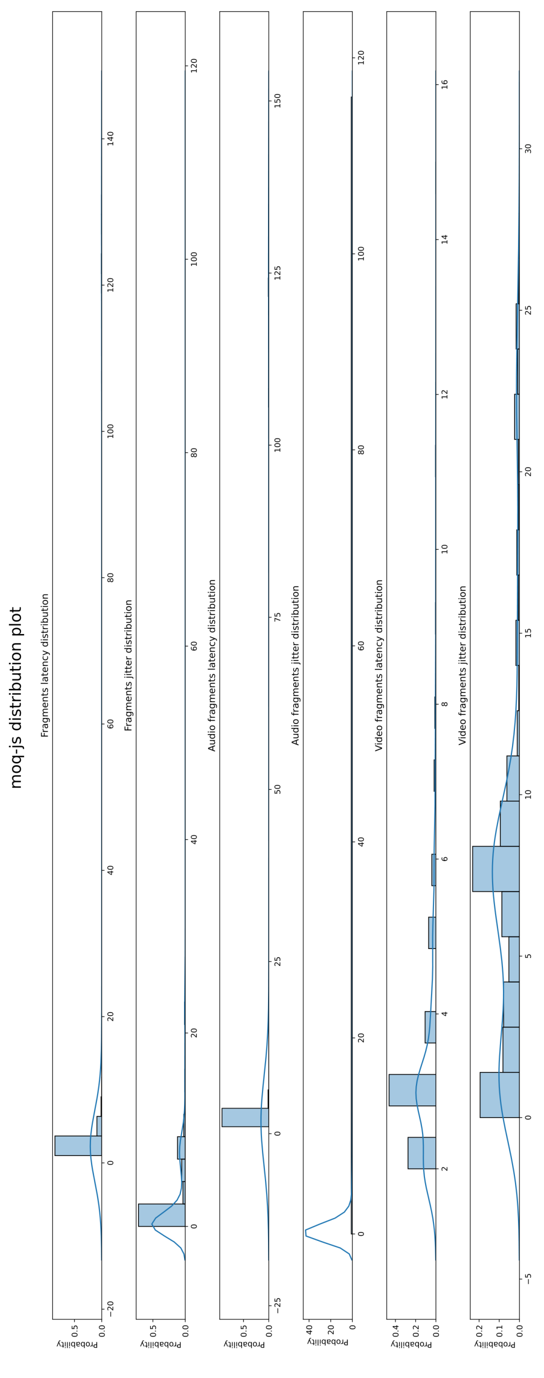
8.4.1 Audio & Video, Rendering Enabled (Run 1)

This test observes typical media delivery when both audio and video are sent over datagram, with playback and rendering enabled.

Table 8.14. Experimental setup: Audio & Video, Rendering Enabled (Run 1), Datagram

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs
Transmission mode:	Datagram
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled





Empirical Observations:

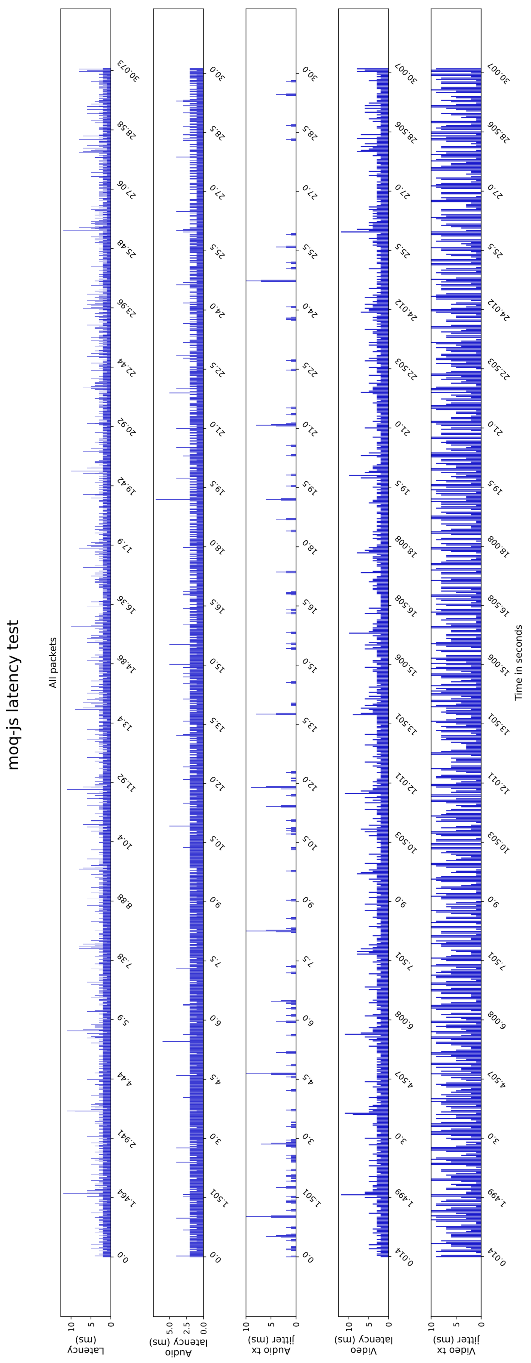
Media delivery was smooth, with only a minor increase in mean end-to-end latency (2–4 ms higher than stream mode) and modestly higher jitter. Occasional out-of-order arrivals were detected by logs but fully masked by application reassembly logic—resulting in seamless playback, no stalls, and undetectable artifacts to the end user.

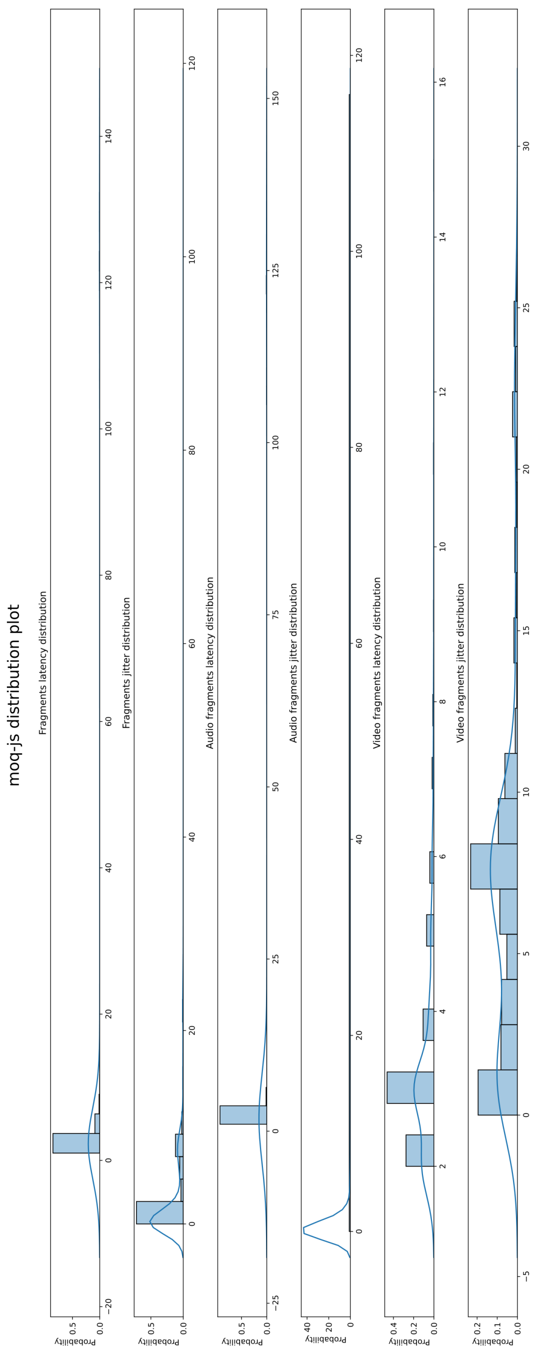
8.4.2 Audio & Video, Rendering Enabled (Run 2)

A repeat of the previous test for statistical confirmation.

Table 8.15. Experimental setup: Audio & Video, Rendering Enabled (Run 2), Datagram

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs
Transmission mode:	Datagram
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Enabled





Empirical Observations:

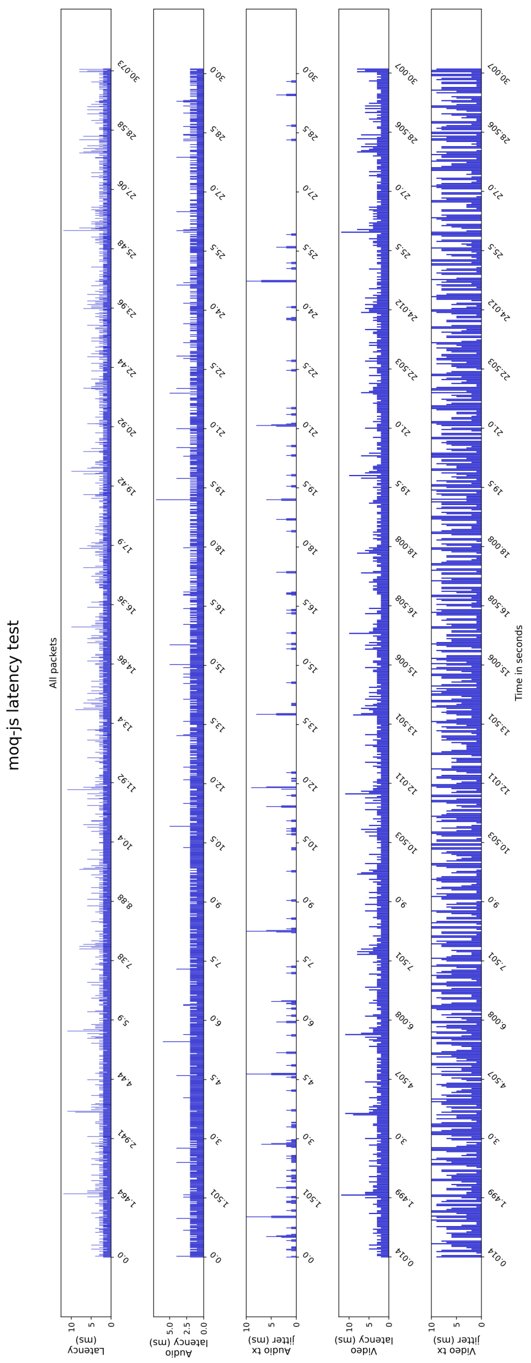
Results mirrored the first run, confirming the reliability of datagram delivery in local, lossless conditions. Observed metrics (latency, jitter, out-of-order rate) were statistically indistinguishable from the previous run, establishing a repeatable reference point.

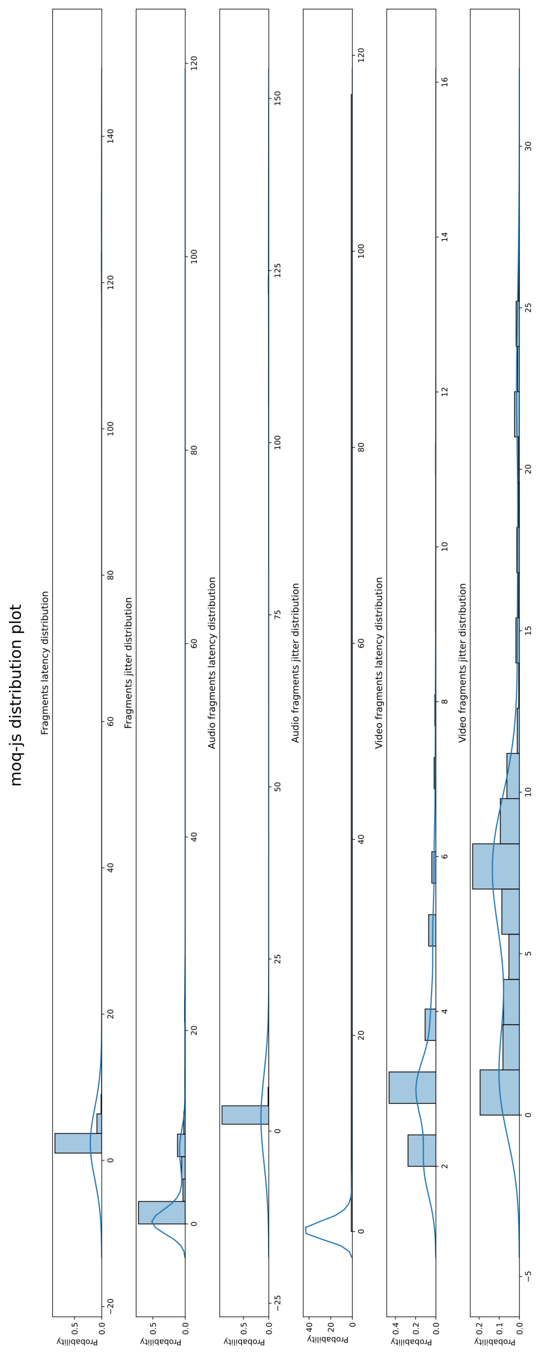
8.4.3 Audio & Video, Rendering Disabled

This test isolates the protocol by removing playback/rendering load from the endpoint.

Table 8.16. Experimental setup: Audio & Video, Rendering Disabled, Datagram

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs
Transmission mode:	Datagram
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Rendering:	Disabled





Empirical Observations:

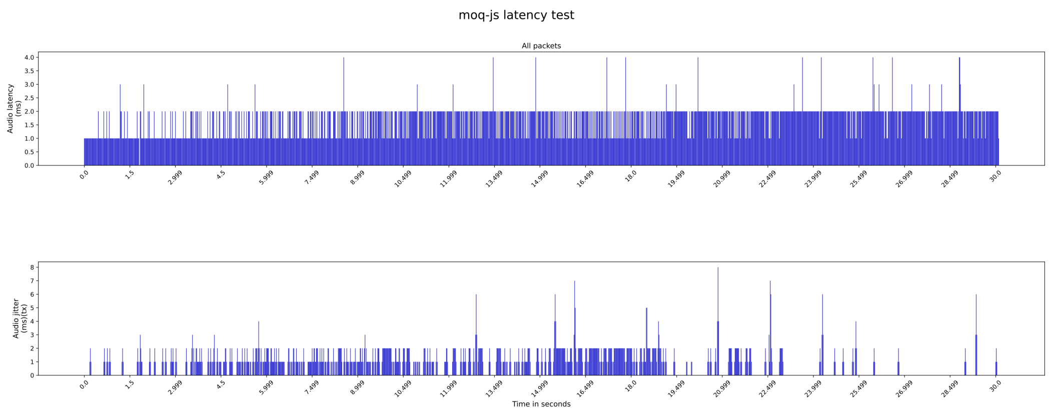
Protocol-level metrics and user-visible performance remained unchanged relative to the rendering-enabled runs. Application-layer reassembly and jitter buffering continued to conceal all packet reordering or microbursts. CPU usage dropped slightly, but had no impact on delivery statistics.

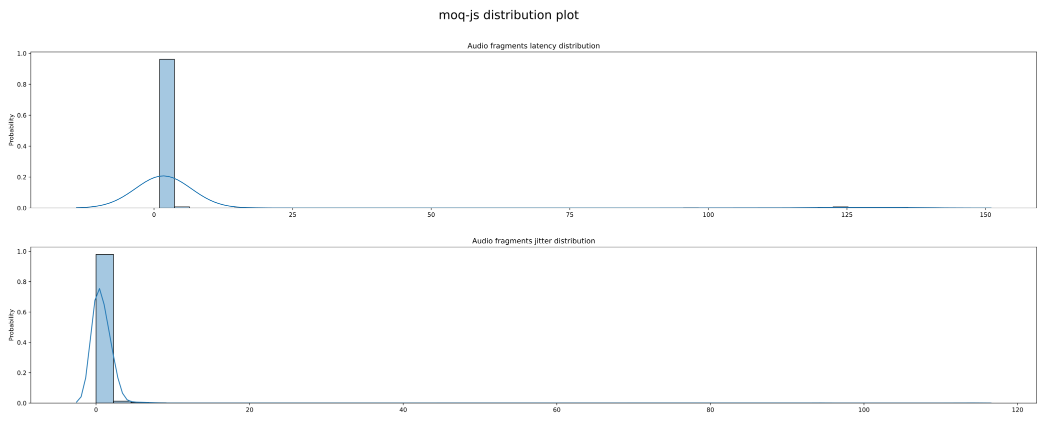
8.4.4 Audio Only

This scenario investigates audio-only datagram performance in the absence of bandwidth or rendering constraints.

Table 8.17. Experimental setup: Audio Only, Datagram

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs
Transmission mode:	Datagram
Media:	Opus audio 128 kbps
Rendering:	-





Empirical Observations:

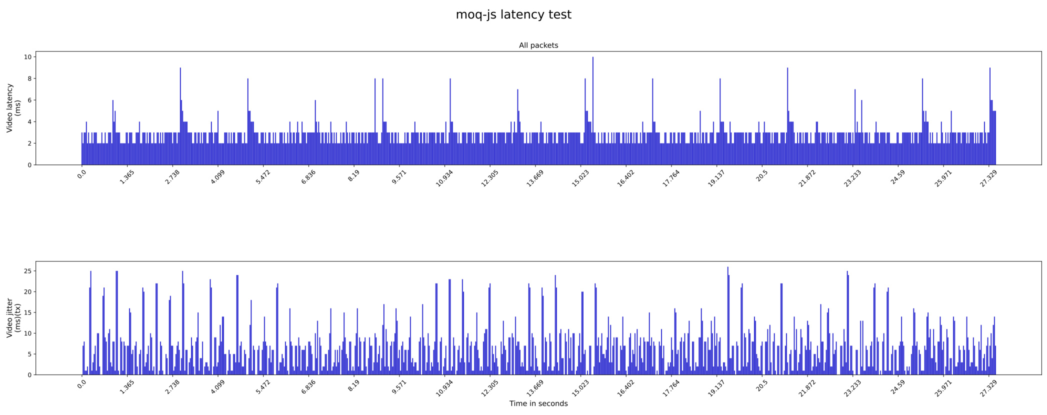
Audio-only delivery via datagram mode was extremely robust: all packets arrived in time for playback, with negligible jitter and only a single reordering event (again, invisible to the user). The low bit rate and small packet size contributed to near-ideal behavior.

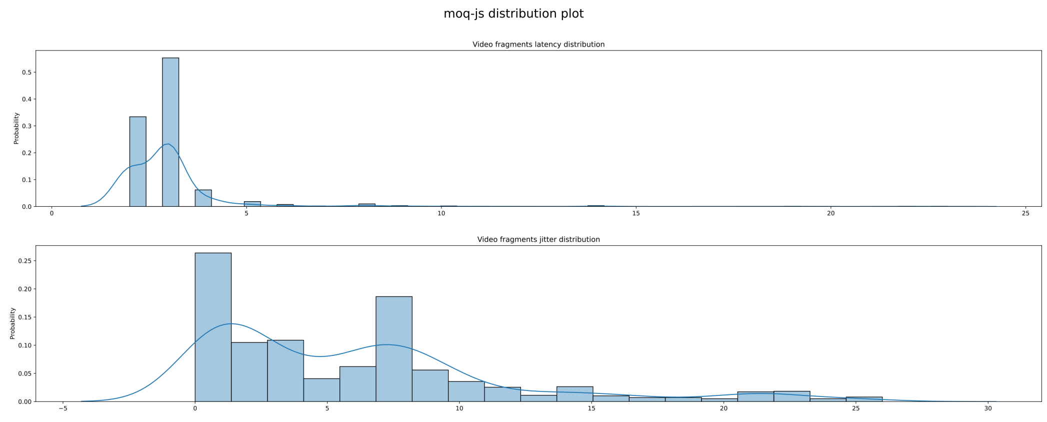
8.4.5 Video Only

Examines high-rate video performance over datagram in optimal conditions.

Table 8.18. Experimental setup: Video Only, Datagram

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs
Transmission mode:	Datagram
Media:	VP8 video 2000 kbps
Rendering:	Enabled





Empirical Observations:

Even at high packet rates, datagram mode managed to deliver all video frames in order with only rare use of jitter buffer reassembly. A minor median latency increase (∼2 ms) was measured compared to stream mode, but this had no visible impact. No dropped or frozen frames were observed under these ideal conditions.

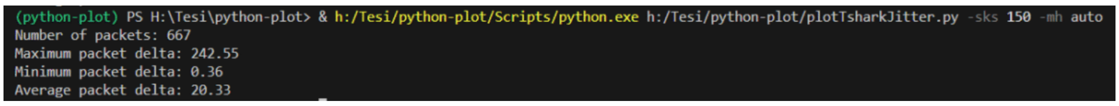
8.5 Bandwidth Limiting: Datagram Transmission

This section systematically investigates the performance of Datagram mode under various static and dynamic bandwidth constraints. Scenarios are designed to probe the protocol's limits—assessing both the immediate impact of bandwidth shaping on traffic and the protocol's resilience in terms of loss, latency, and playback continuity. Comparative notes to Stream mode are included throughout, highlighting key protocol-level tradeoffs under stress.

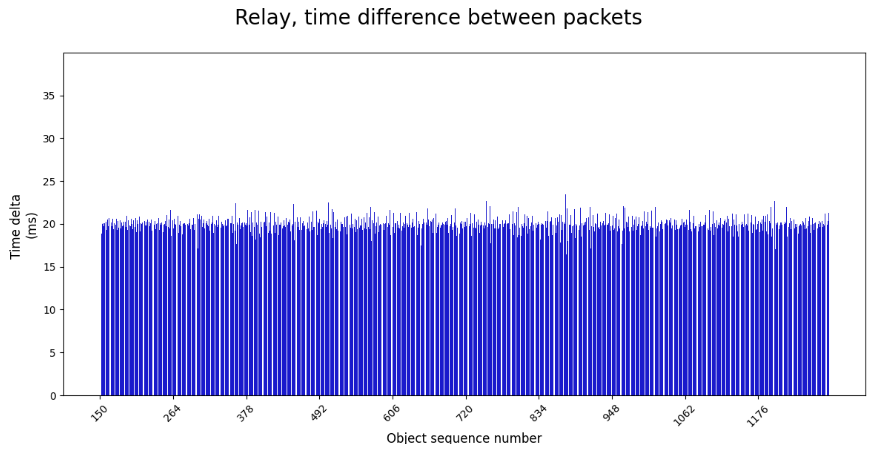
8.5.1 Relay Latency Validation with `tshark`

Table 8.19. Experimental setup: Bandwidth shaping validation, Datagram

Relay:	Remote
Bandwidth limitation:	None
Measurement:	`tshark`, application logs
Transmission mode:	Datagram
Media:	n/a (validation only)



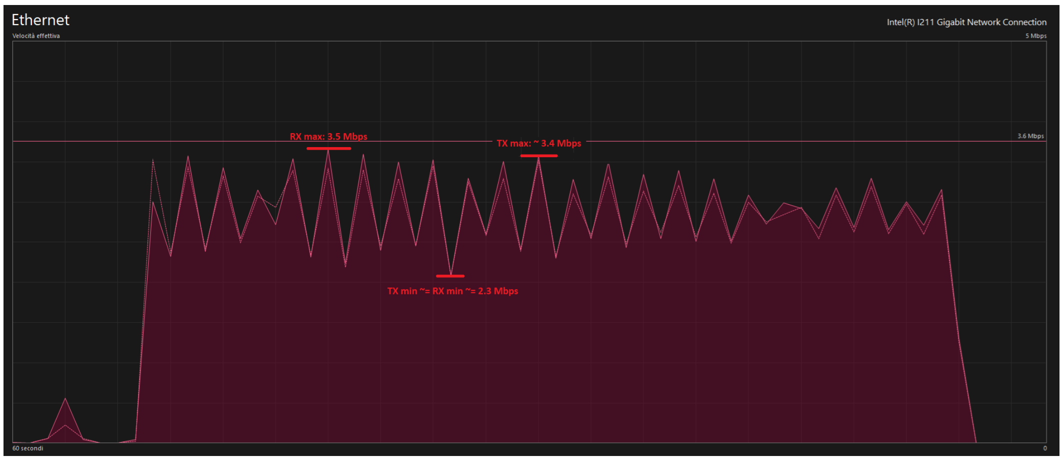
```
(python-plot) PS H:\Tesi\python-plot> & h:/Tesi/python-plot/Scripts/python.exe h:/Tesi/python-plot/plotTsharkJitter.py -sks 150 -mh auto
Number of packets: 667
Maximum packet delta: 242.55
Minimum packet delta: 0.36
Average packet delta: 20.33
```



8.5.2 Audio & Video, Bandwidth Usage under Limiting

Table 8.20. Experimental setup: Audio & Video, Bandwidth Usage, Datagram

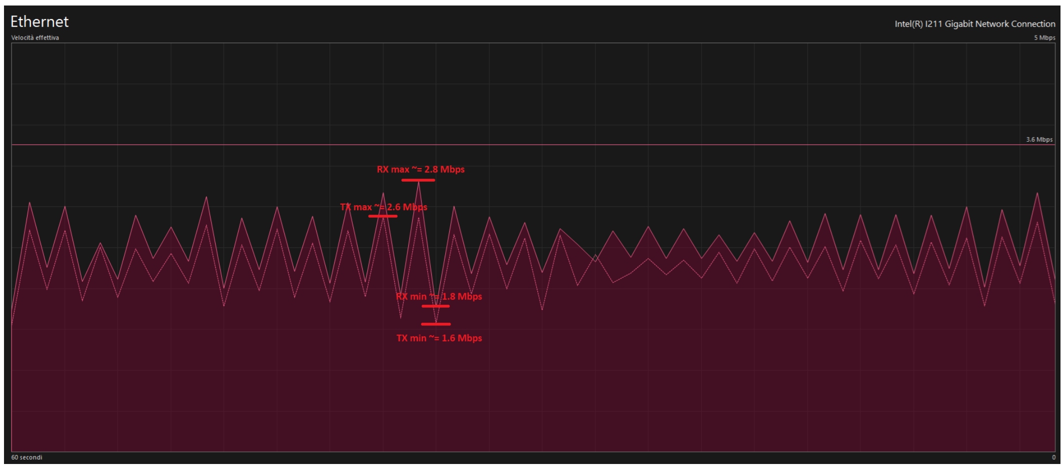
Relay:	Local
Bandwidth limitation:	None
Measurement:	Bandwidth usage at publisher/client
Transmission mode:	Datagram
Media:	Opus audio 128 kbps, VP8 video 2000 kbps



8.5.3 Video Only, Bandwidth Usage

Table 8.21. Experimental setup: Video Only, Bandwidth Usage, Datagram

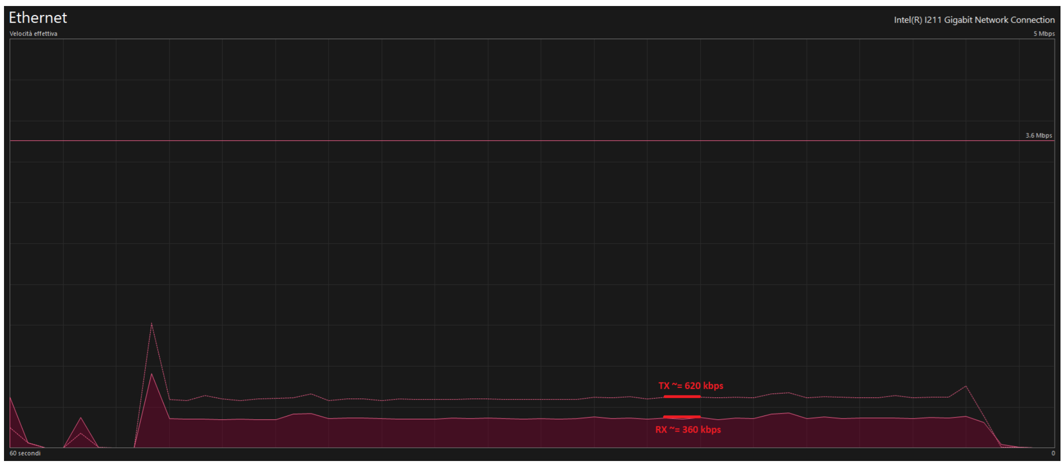
Relay:	Local
Bandwidth limitation:	None
Measurement:	Bandwidth usage at publisher/client
Transmission mode:	Datagram
Media:	VP8 video 2000 kbps



8.5.4 Audio Only, Bandwidth Usage

Table 8.22. Experimental setup: Audio Only, Bandwidth Usage, Datagram

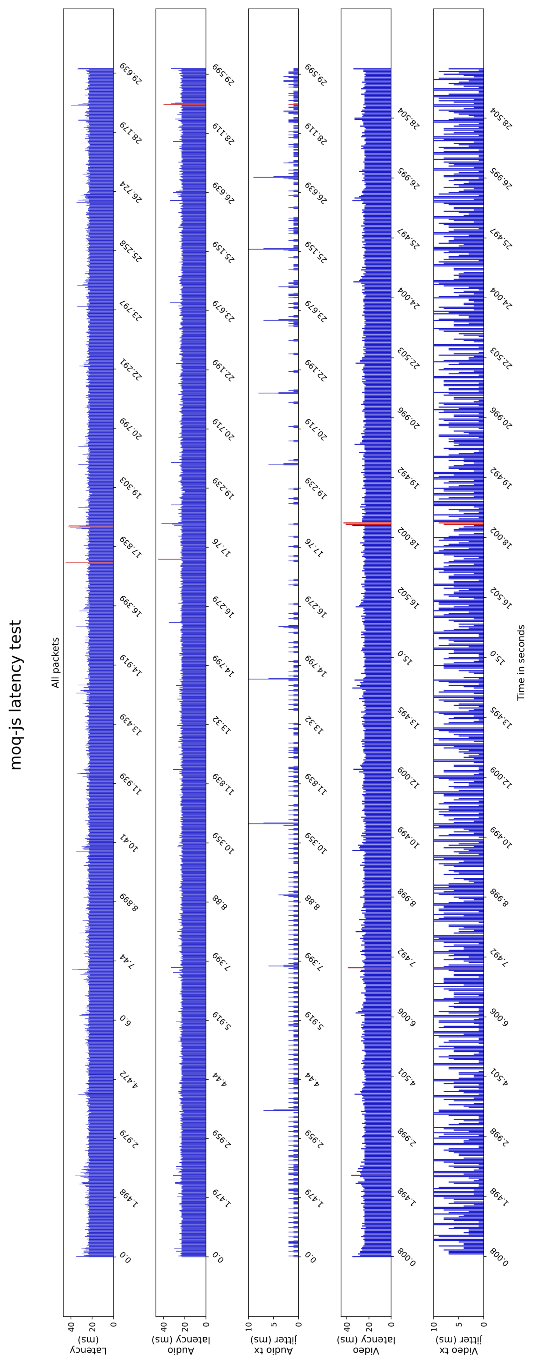
Relay:	Local
Bandwidth limitation:	None
Measurement:	Bandwidth usage at publisher/client
Transmission mode:	Datagram
Media:	Opus audio 128 kbps

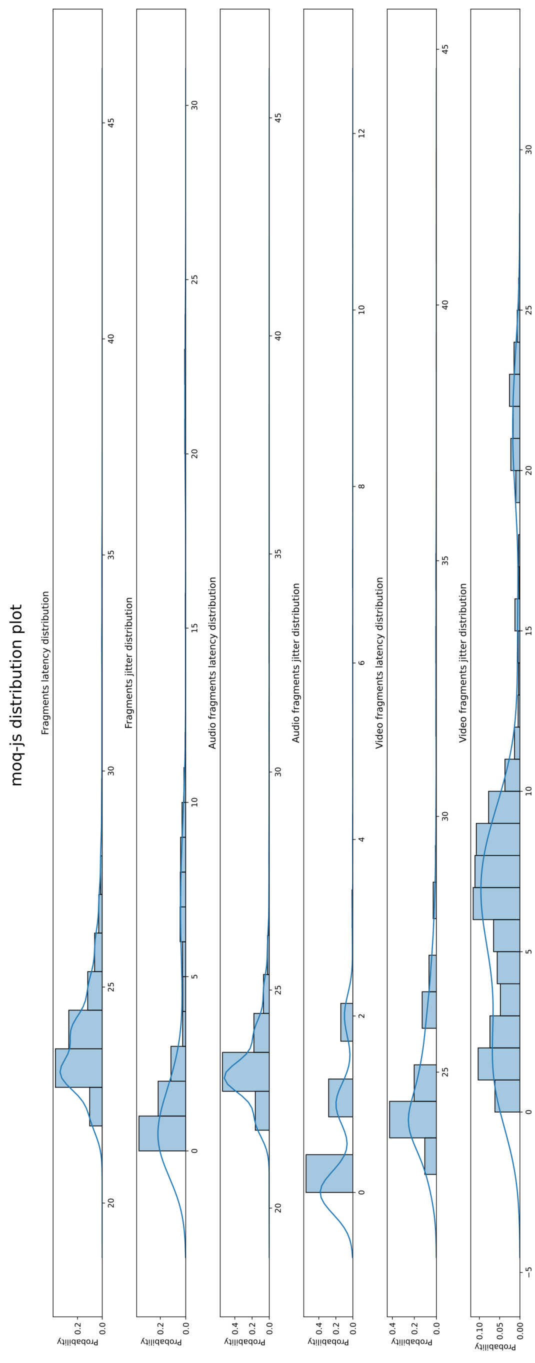


8.5.5 Audio & Video, No Bandwidth Limitation (Reference)

Table 8.23. Experimental setup: Audio & Video, No Bandwidth Limit, Datagram

Relay:	Local
Bandwidth limitation:	None
Measurement:	Application logs
Transmission mode:	Datagram
Media:	Opus audio 128 kbps, VP8 video 2000 kbps





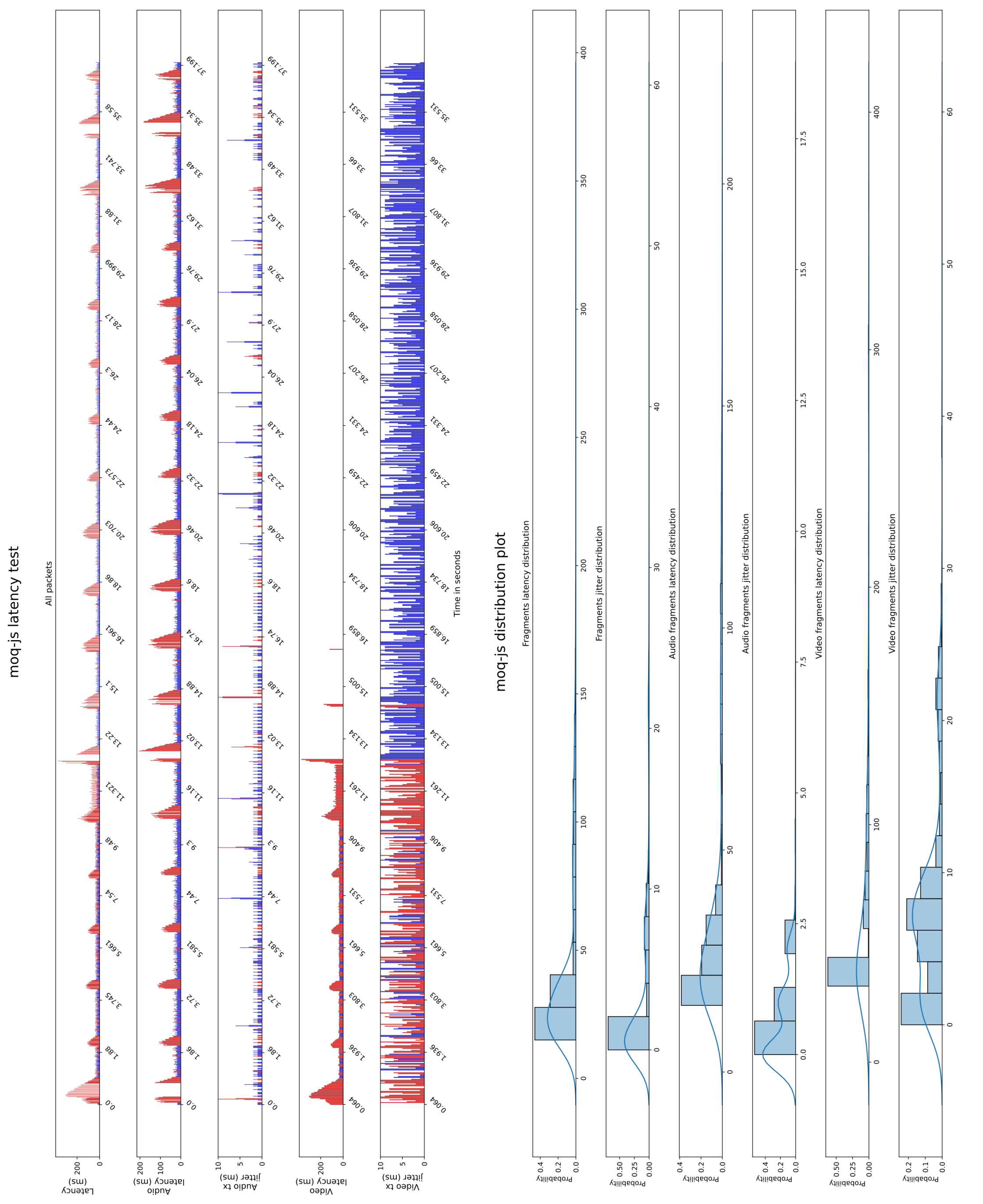
Empirical Observations:

As a reference, unconstrained datagram transmission returned to the baseline: low, stable jitter and negligible packet loss, confirming that all degradations under bandwidth restriction were due to protocol and application behavior under stress, not inherent instability.

8.5.6 Audio & Video, Bandwidth Limited to 4 Mbps

Table 8.24. Experimental setup: Audio & Video, 4 Mbps Limit, Datagram

Relay:	Local
Bandwidth limitation:	4 Mbps up/down
Measurement:	Application logs
Transmission mode:	Datagram
Media:	Opus audio 128 kbps, VP8 video 2000 kbps



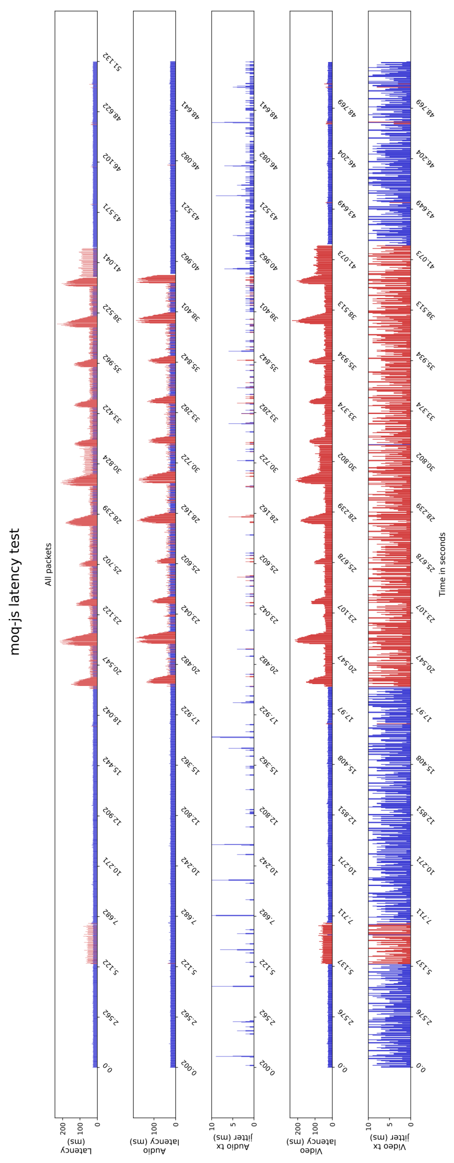
Empirical Observations:

With a 4 Mbps cap, media quality remained generally high, but bandwidth spikes occasionally triggered microbursts of loss and higher jitter. Users might notice brief reductions in video quality, but audio streams continued unaffected—a clear contrast to heavier restrictions.

8.5.7 Audio & Video, Dynamic Bandwidth Limiting (No Limit → 4 Mbps)

Table 8.25. Experimental setup: Audio & Video, Dynamic Limit, Datagram

Relay:	Local
Bandwidth limitation:	None → 4 Mbps (dynamic)
Measurement:	Application logs
Transmission mode:	Datagram
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Other:	Video/audio artifacts noted



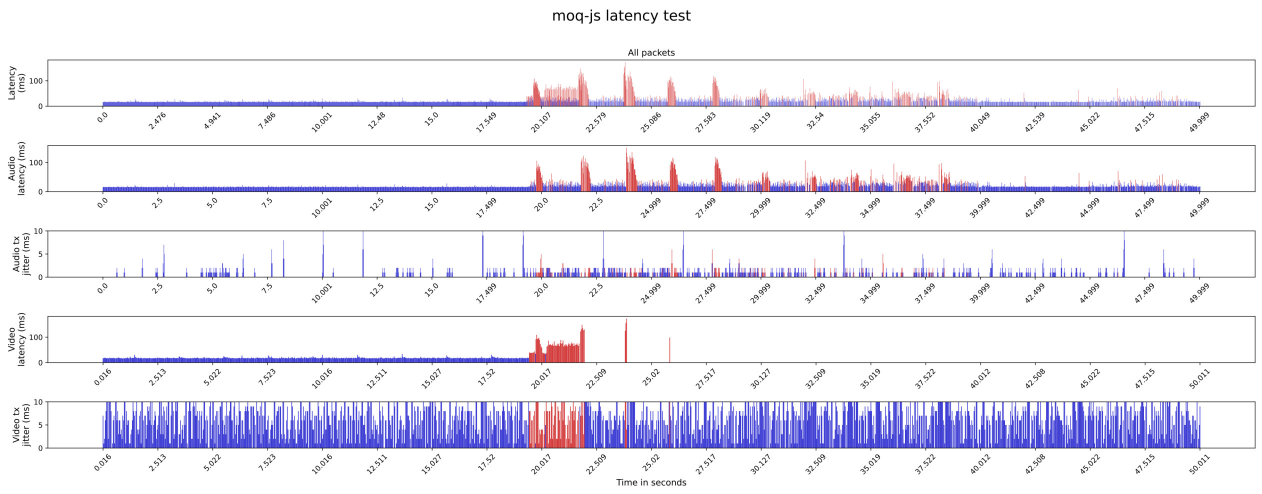
Empirical Observations:

Imposing a sudden 4 Mbps cap resulted in visible video freezing and transient artifacts, as well as short-lived audio dropouts. Recovery was quick after lifting the cap, but the sensitivity of datagram mode to abrupt bandwidth changes was clear—mirroring the less smooth congestion control and reassembly behavior.

8.5.8 Audio & Video, Dynamic Bandwidth Limiting with Loss of Recovery

Table 8.26. Experimental setup: Audio & Video, Persistent Dynamic Limit, Datagram

Relay:	Local
Bandwidth limitation:	None → 4 Mbps (not reverted)
Measurement:	Application logs
Transmission mode:	Datagram
Media:	Opus audio 128 kbps, VP8 video 2000 kbps
Other:	Video/audio artifacts, playback interruption



Empirical Observations:

When the imposed bandwidth reduction persisted, datagram mode proved unable to recover fully: video playback was interrupted almost immediately and did not resume; audio quality degraded, with silence periods emerging. This highlights a structural vulnerability of datagram transmission to sustained capacity loss.

8.6 Comparison of QUIC datagram and stream transmission

While QUIC streams provide reliable, in-order delivery analogous to TCP, QUIC datagrams enable unreliable, message-oriented delivery similar to UDP, but multiplexed within the same secure connection.

This section provides a technical comparison of these two paradigms under a variety of realistic media workloads and controlled network impairments (e.g., bandwidth throttling, loss, and delay).

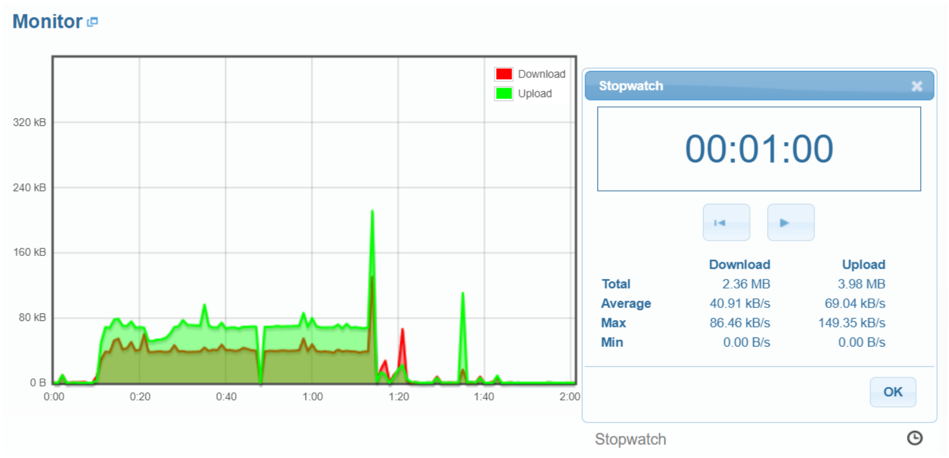
By analyzing quantitative metrics (bandwidth usage, latency, packet loss) and correlating them with empirical user experience, we aim to highlight the practical trade-offs between reliability, latency, and efficiency in real-world streaming scenarios.

Each scenario is presented with a concise summary table of its parameters, followed by visualizations and interpretation of results.

Whenever relevant, practical notes on playback quality, stalling, and recovery are highlighted to link measured performance with actual end-user experience.

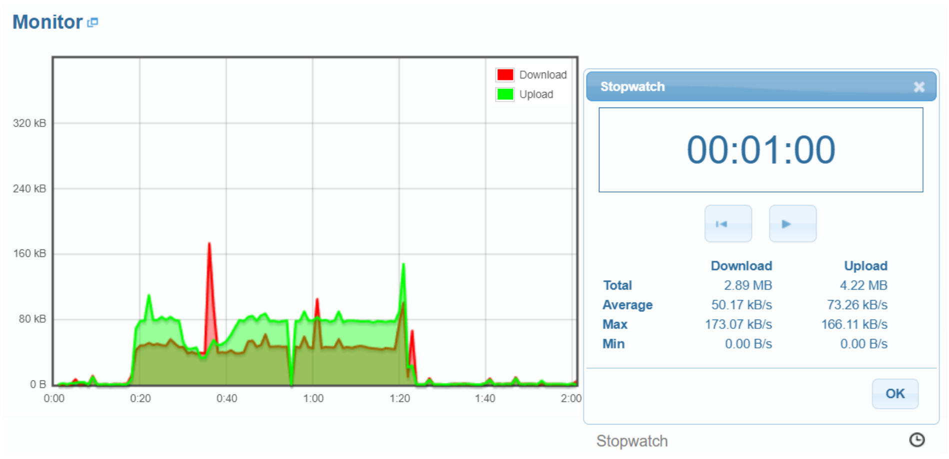
8.6.1 Scenario: Audio 128 kbps, Bandwidth limited 400 kbps to 300 kbps

Media	Opus audio (128 kbps)
Video	—
Total duration	60 s
Bandwidth pattern	10 s unlimited → 15 s @400 kbps → 10 s unlimited → 15 s @300 kbps
Other impairments	None



	Download	Upload
Total	2.36 MB	3.98 MB
Average	40.91 kB/s	69.04 kB/s
Max	86.46 kB/s	149.35 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.1. Stream mode bandwidth usage (60s).



	Download	Upload
Total	2.89 MB	4.22 MB
Average	50.17 kB/s	73.26 kB/s
Max	173.07 kB/s	166.11 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.2. Datagram mode bandwidth usage (60s).

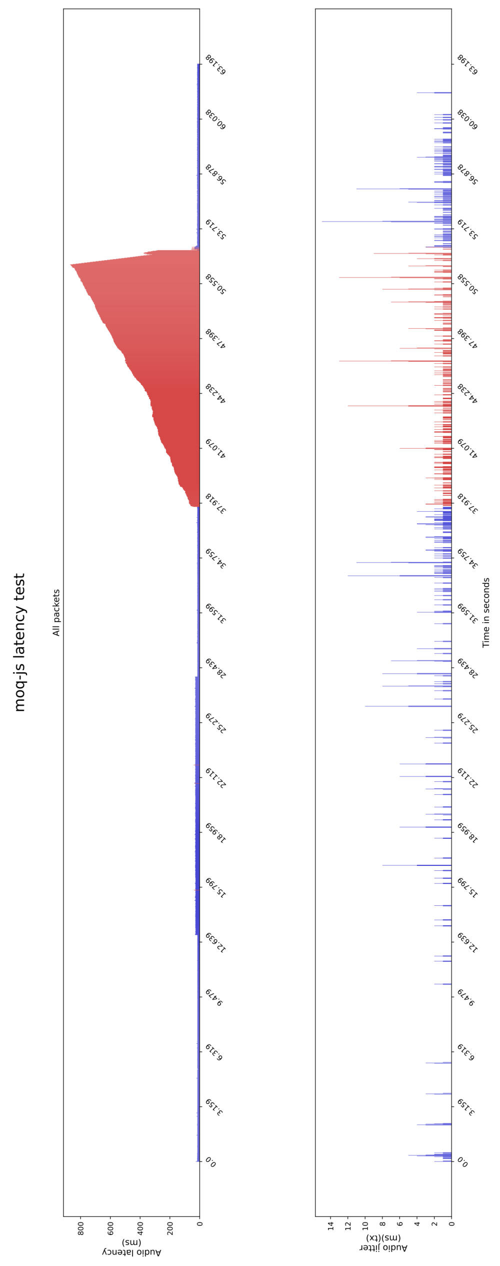


Figure 8.3. Stream mode latency visualization (latency over time).

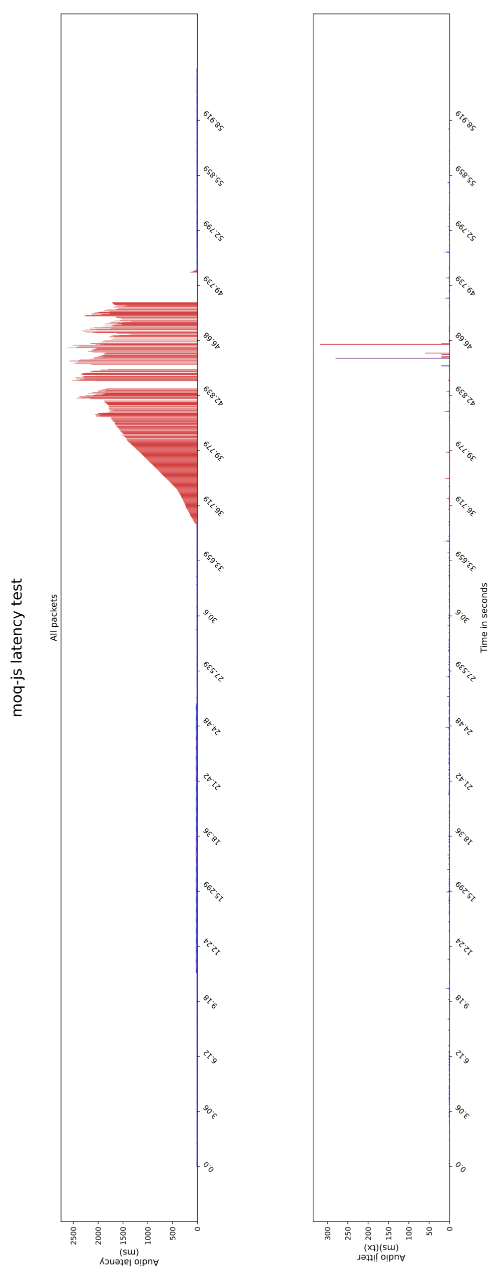


Figure 8.4. Datagram mode latency visualization (latency over time).

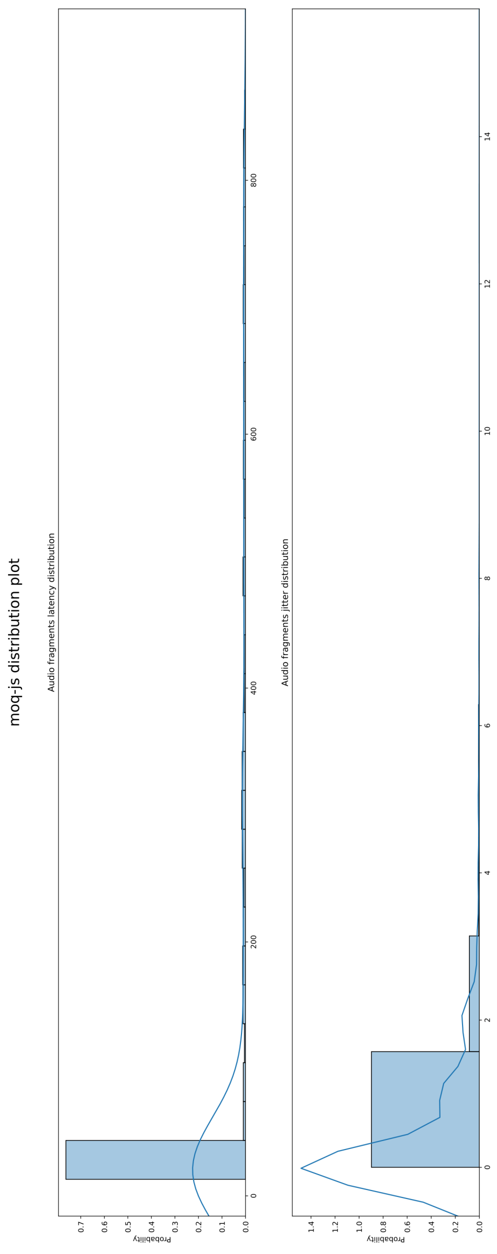


Figure 8.5. Stream mode latency distribution.

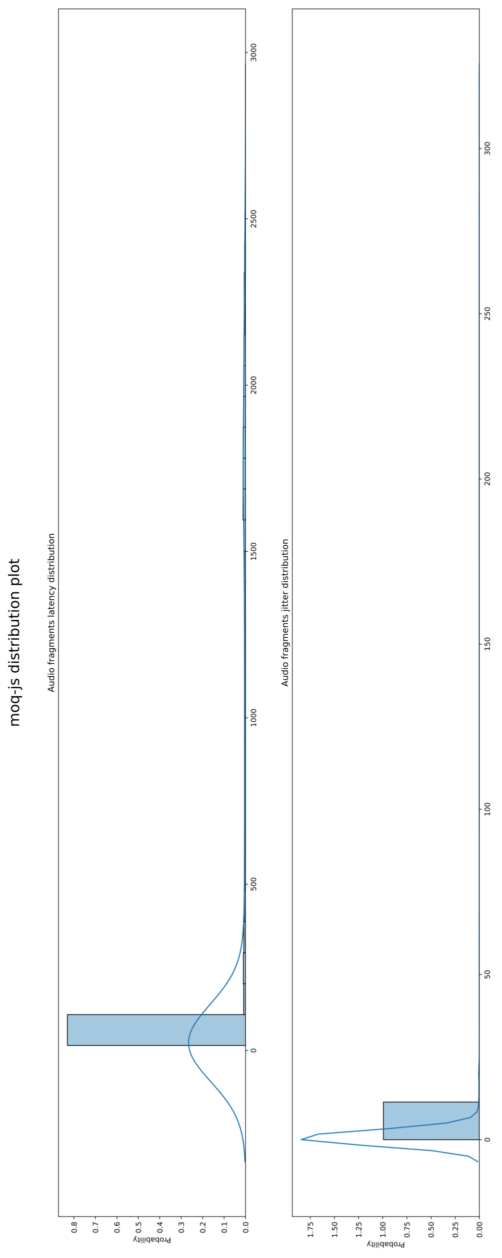


Figure 8.6. Datagram mode latency distribution.

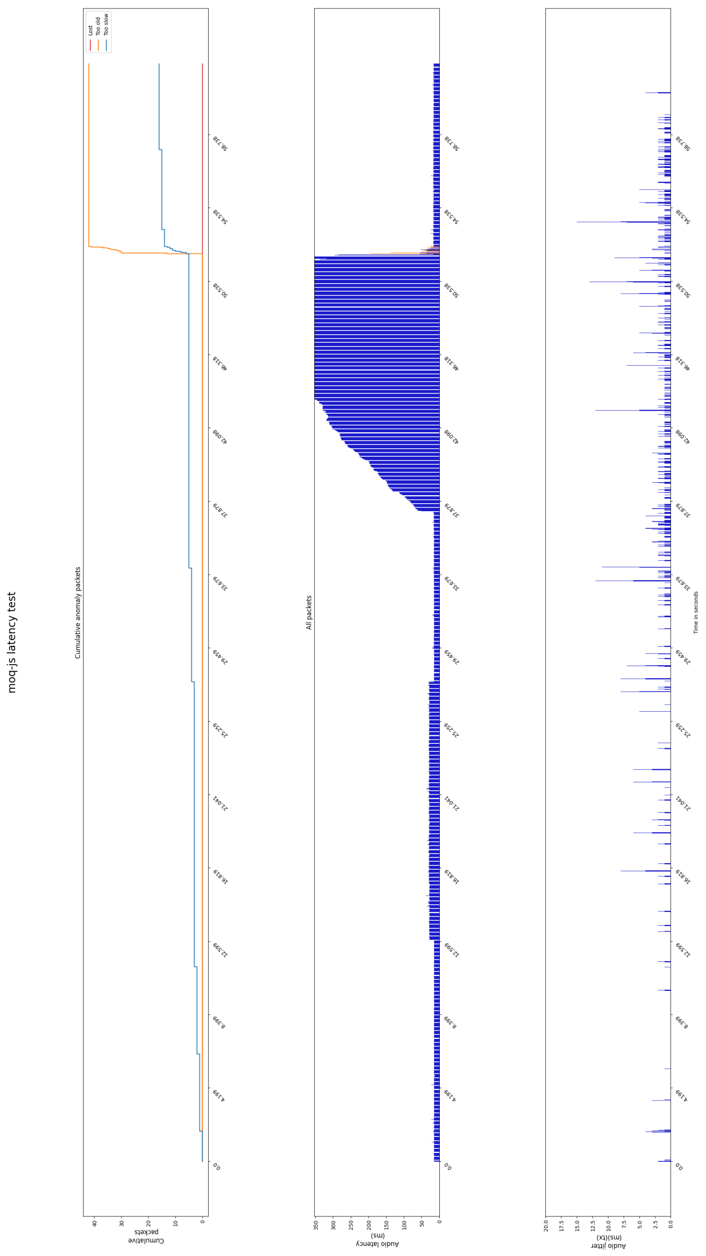


Figure 8.7. Stream mode cumulative packet loss.

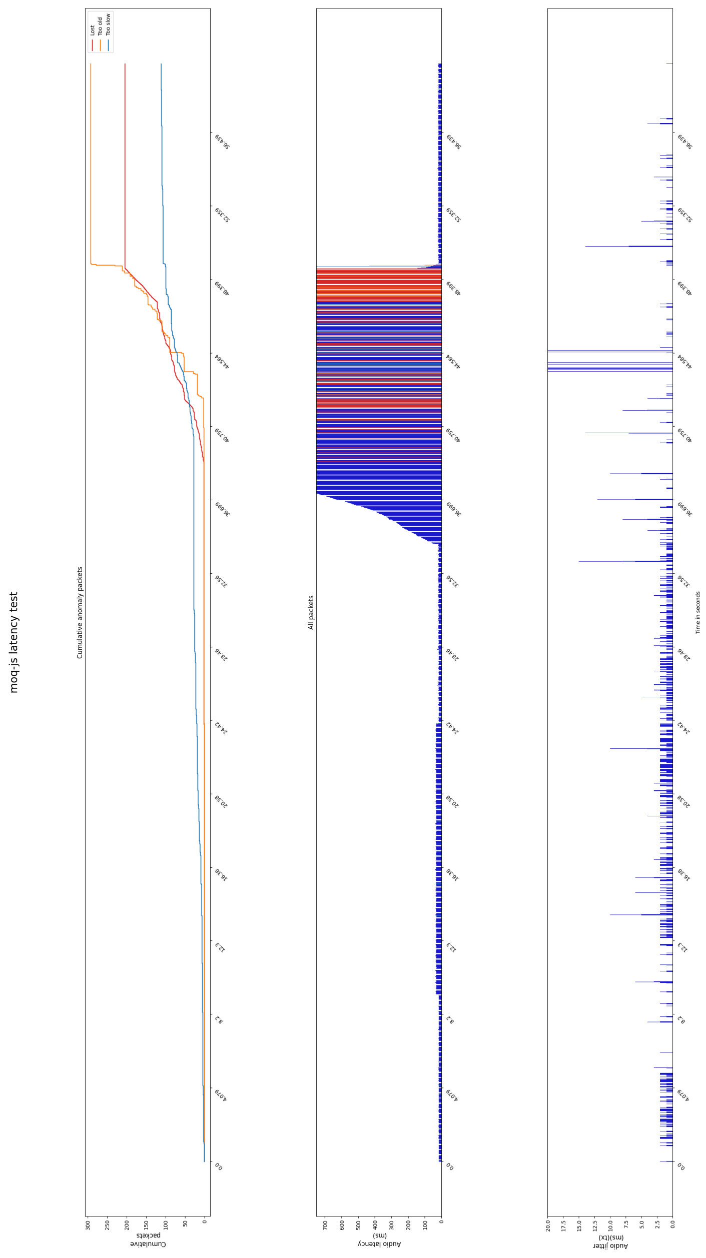


Figure 8.8. Datagram mode cumulative packet loss.

Table 8.27. Bandwidth Usage Summary for a128 400k→300k scenario (stream and datagram).

Scenario	Mode	Direction	Total (MB)	Average (kB/s)	Max (kB/s)
a128 400k→300k	Stream	Download	2.36	40.91	86.46
	Stream	Upload	3.98	69.04	149.35
	Datagram	Download	2.89	50.17	173.07
	Datagram	Upload	4.22	73.26	166.11

Scenario	Mode	LOST	LOST (%)	TOO OLD	TOO OLD (%)	TOO SLOW	TOO SLOW (%)	Not rec.	Not rec. (%)	Total
a128 400k→300k	Stream	0	0.00	42	1.31	16	0.50	58	1.81	3200
	Datagram	204	6.04	292	8.64	111	3.28	607	17.96	3380

Table 8.28. Packet loss statistics for audio 128 kbps, 400 kbps to 300 kbps bandwidth limitation.

Empirical observations:

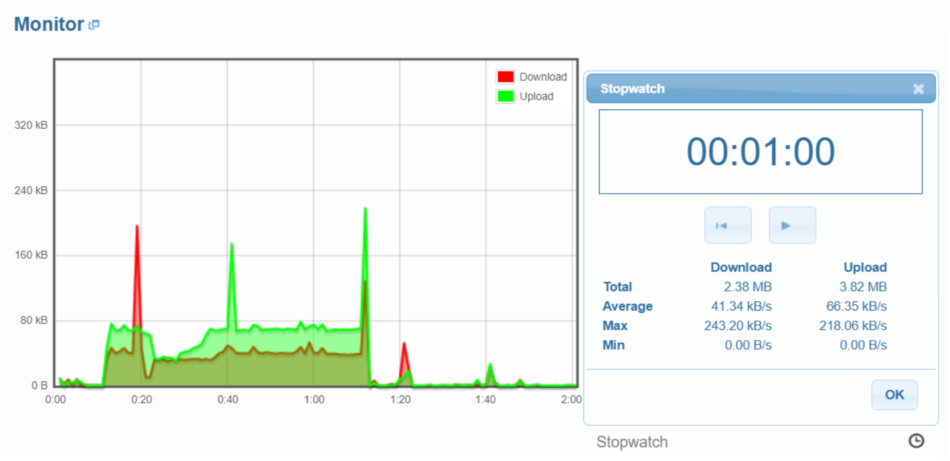
During periods of stricter bandwidth limitation (especially 300 kbps), datagram mode playback is more affected by skips and latency spikes, whereas stream mode maintains a more stable audio experience, with only minor delays. As bandwidth drops from 400

kbps to 300 kbps, stream mode delivers nearly all packets with only minor lateness, resulting in stable playback. Datagram mode, in contrast, experiences significant packet loss and late arrivals, leading to more frequent audio skips and a marked decrease in quality during constrained periods. This illustrates the superior robustness of stream delivery in

constrained bandwidth scenarios.

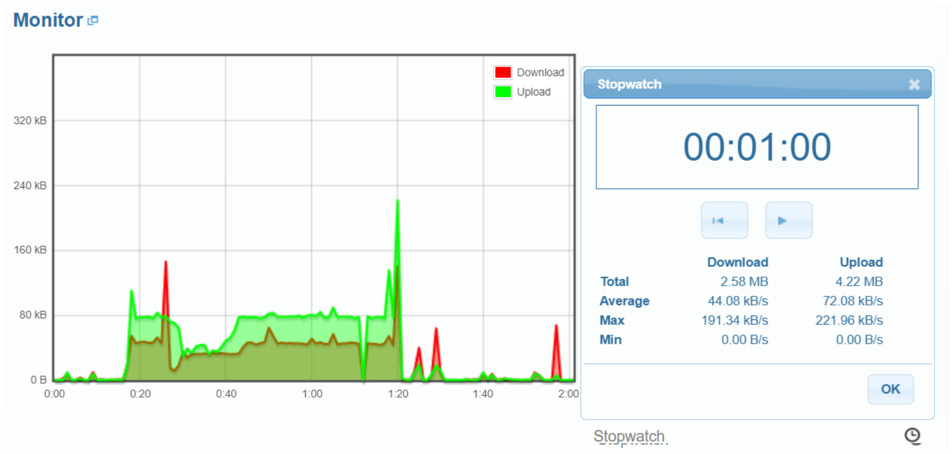
8.6.2 Scenario: Audio 128 kbps, Bandwidth limited 500 kbps to 250 kbps

Media	Opus audio (128 kbps)
Video	—
Total duration	60 s
Bandwidth pattern	10 s unlimited → 15 s @500 kbps → 10 s unlimited → 15 s @250 kbps
Other impairments	None



	Download	Upload
Total	2.38 MB	3.82 MB
Average	41.34 kB/s	66.35 kB/s
Max	243.20 kB/s	218.06 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.9. Stream mode bandwidth usage (60s).



	Download	Upload
Total	2.58 MB	4.22 MB
Average	44.08 kB/s	72.08 kB/s
Max	191.34 kB/s	221.96 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.10. Datagram mode bandwidth usage (60s).

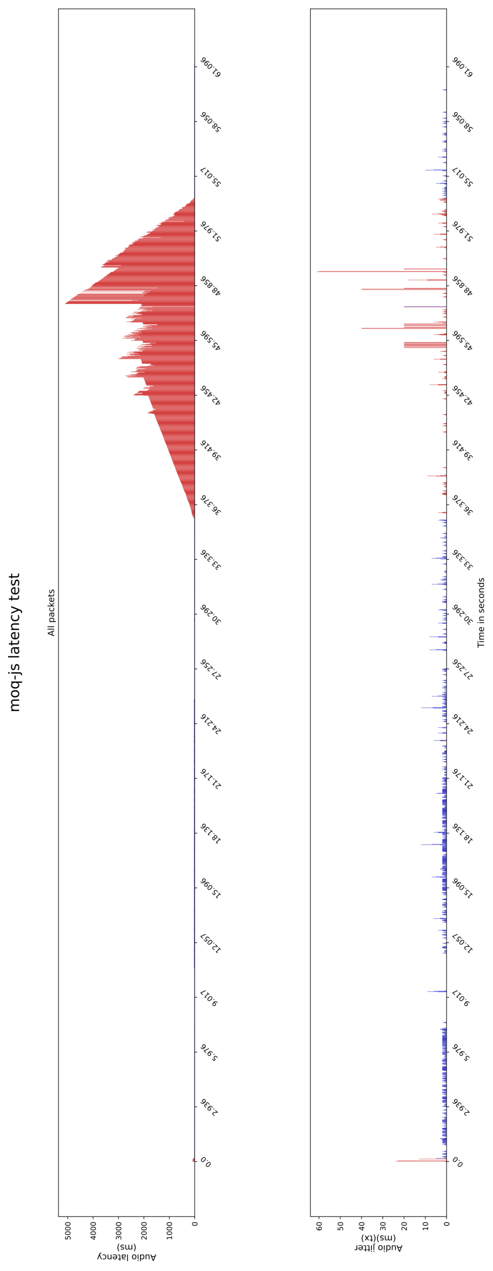


Figure 8.11. Stream mode latency visualization (latency over time).

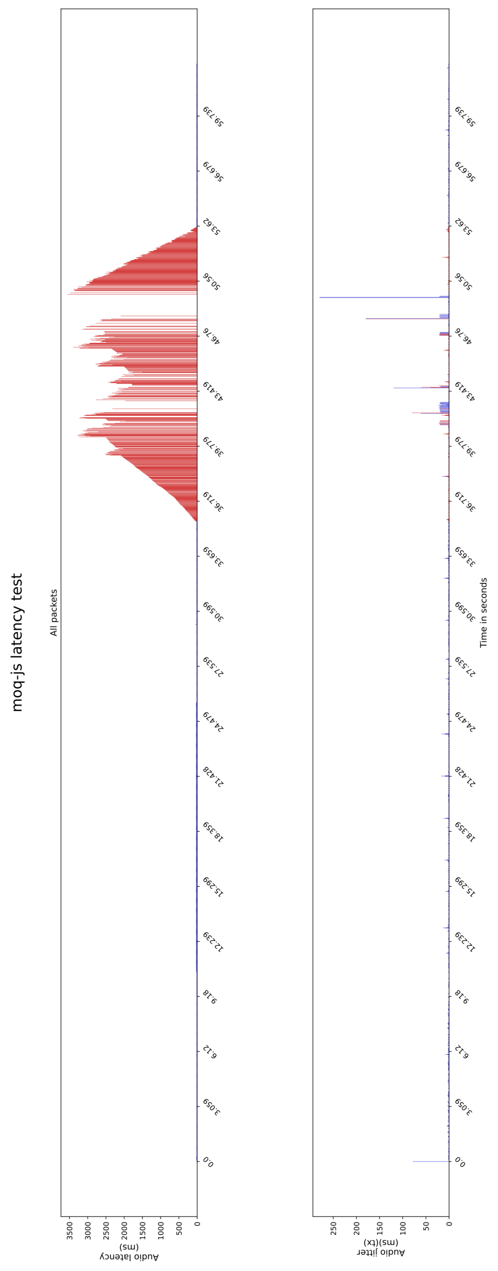


Figure 8.12. Datagram mode latency visualization (latency over time).

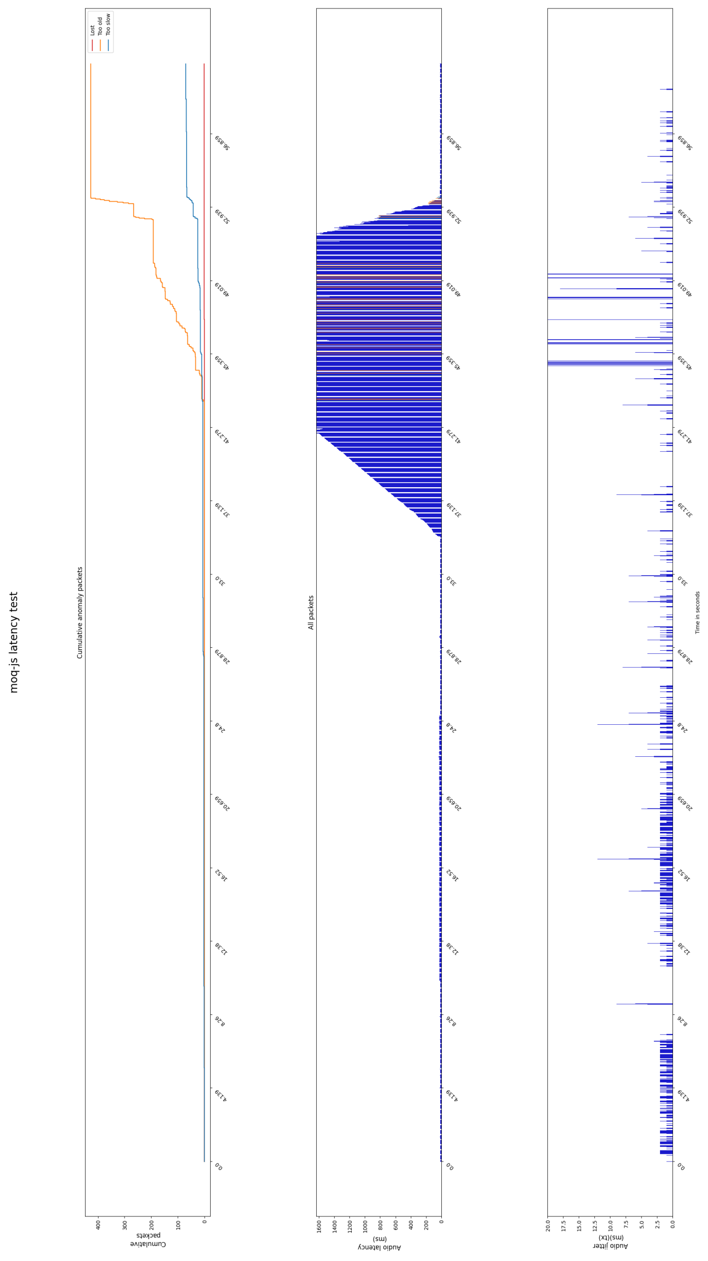


Figure 8.13. Stream mode cumulative packet loss.

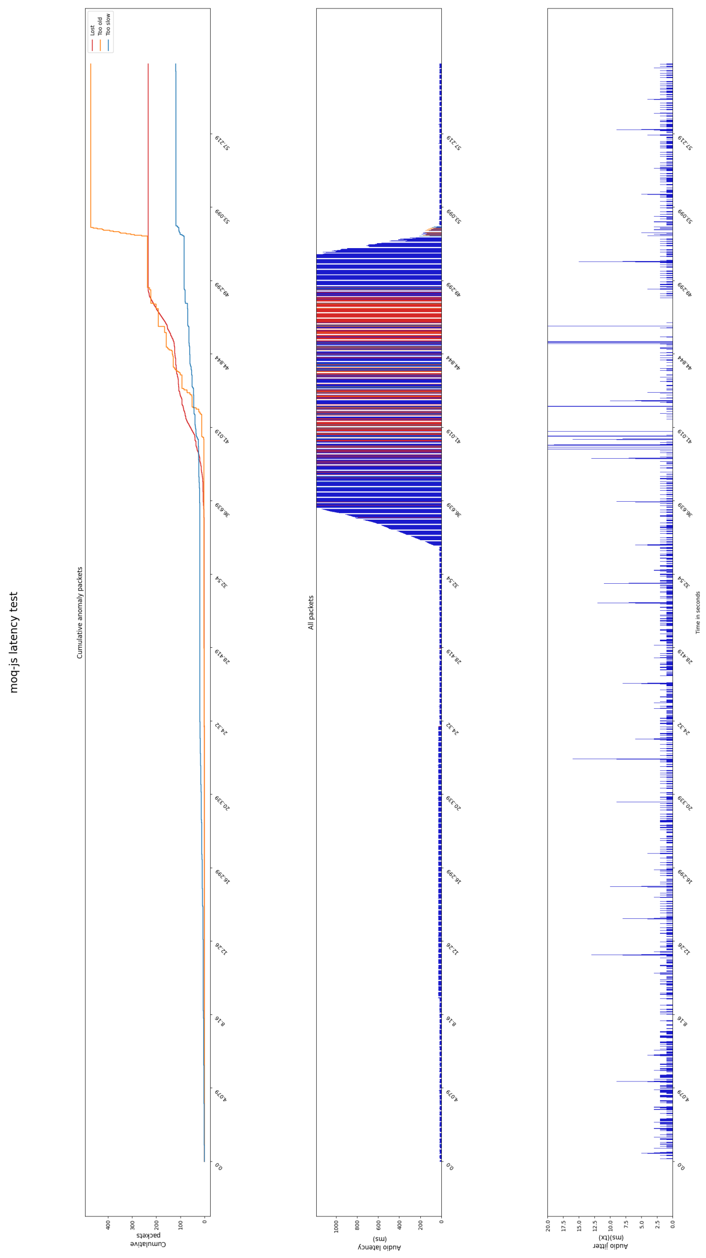


Figure 8.14. Datagram mode cumulative packet loss.

Table 8.29. Bandwidth Usage Summary for a128 500k→250k scenario (stream and datagram).

Scenario	Mode	Direction	Total (MB)	Average (kB/s)	Max (kB/s)
a128 500k→250k	Stream	Download	2.38	41.34	243.20
	Stream	Upload	3.82	66.35	218.06
	Datagram	Download	2.58	44.08	191.34
	Datagram	Upload	4.22	72.08	221.96

Scenario	Mode	LOST	LOST (%)	TOO OLD	TOO OLD (%)	TOO SLOW	TOO SLOW (%)	Not rec.	Not rec. (%)	Total
a128 500k→250k	Stream	1	0.03	427	12.14	70	1.99	498	14.16	3517
	Datagram	233	6.53	471	13.20	119	3.34	823	23.07	3568

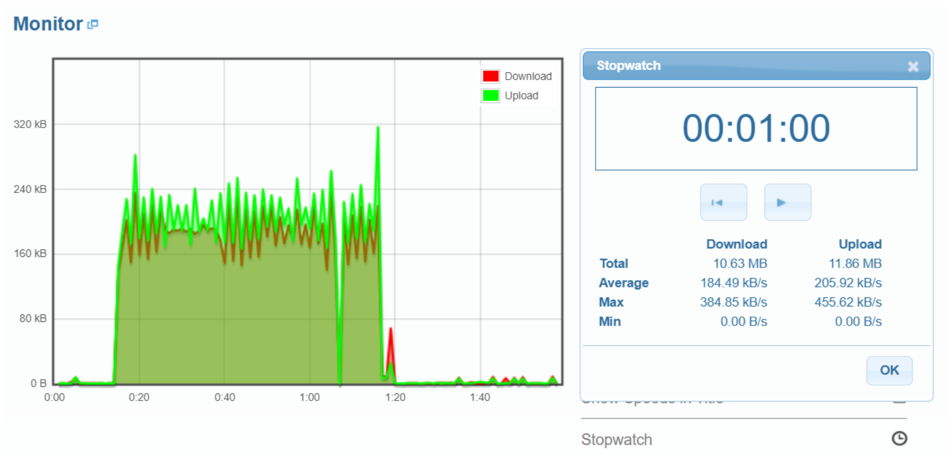
Table 8.30. Packet loss statistics for audio 128 kbps, 500 kbps to 250 kbps bandwidth limitation.

Empirical observations:

In this scenario with more severe bandwidth reduction, datagram mode shows further increased loss and late packets, leading to severe playback interruptions. Stream mode maintains very low packet loss but exhibits moderate lateness and is generally able to maintain audio continuity. Playback in datagram mode is more sensitive to the lower 250 kbps cap, with frequent skips and higher latency spikes compared to stream mode. The difference in user experience is even more apparent: stream mode audio remains comprehensible, while datagram mode becomes unstable.

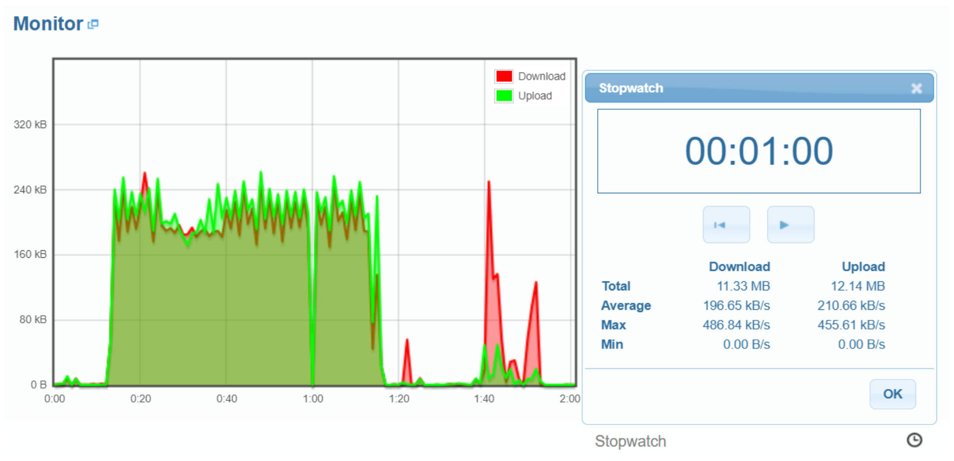
Scenario: Audio 128 kbps, Video 1000 kbps, Bandwidth limited 2 Mbit/s to 1.5 Mbit/s (Test 1)

Media	Opus audio (128 kbps)
Video	VP8 video (1000 kbps)
Total duration	60 s
Bandwidth pattern	10 s unlimited → 15 s @2 Mbps → 10 s unlimited → 15 s @1.5 Mbps
Other impairments	None



	Download	Upload
Total	10.63 MB	11.86 MB
Average	184.49 kB/s	205.92 kB/s
Max	384.85 kB/s	455.62 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.15. Stream mode bandwidth usage (60s).



	Download	Upload
Total	11.33 MB	12.14 MB
Average	196.65 kB/s	210.66 kB/s
Max	486.84 kB/s	455.61 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.16. Datagram mode bandwidth usage (60s).

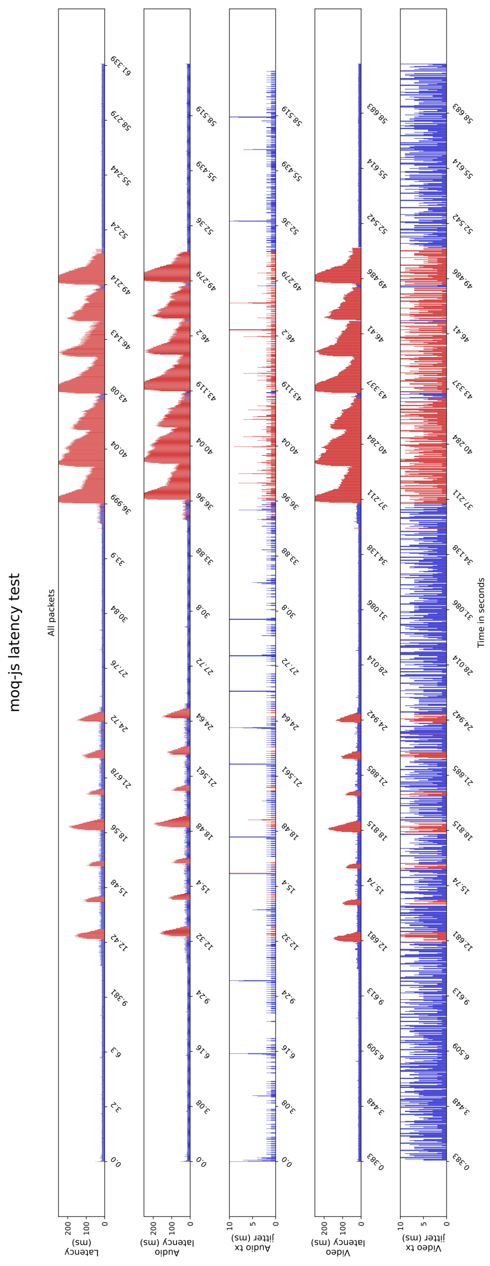


Figure 8.17. Stream mode latency visualization (latency over time).

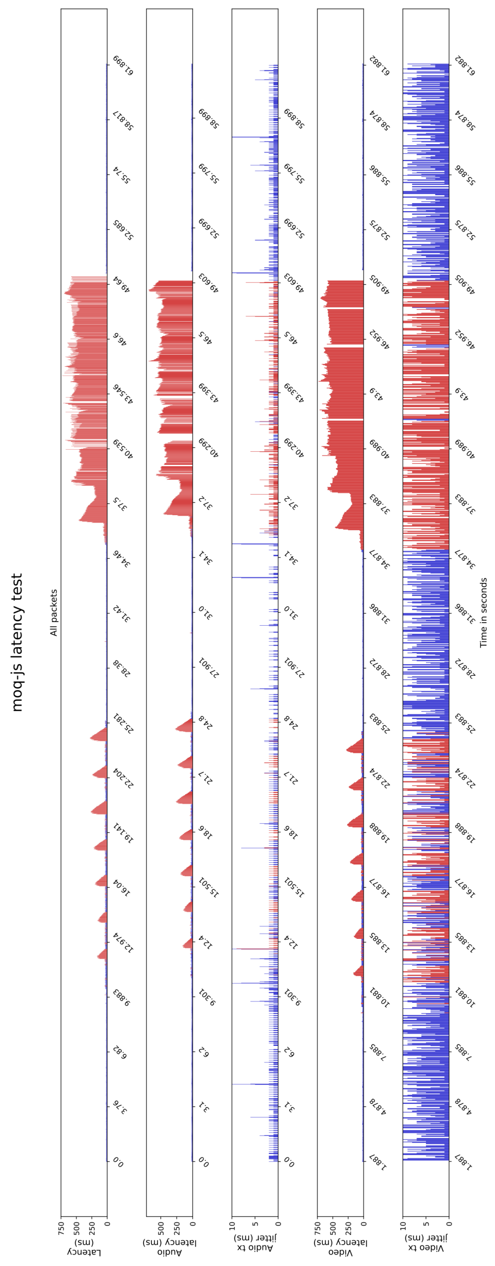


Figure 8.18. Datagram mode latency visualization (latency over time).

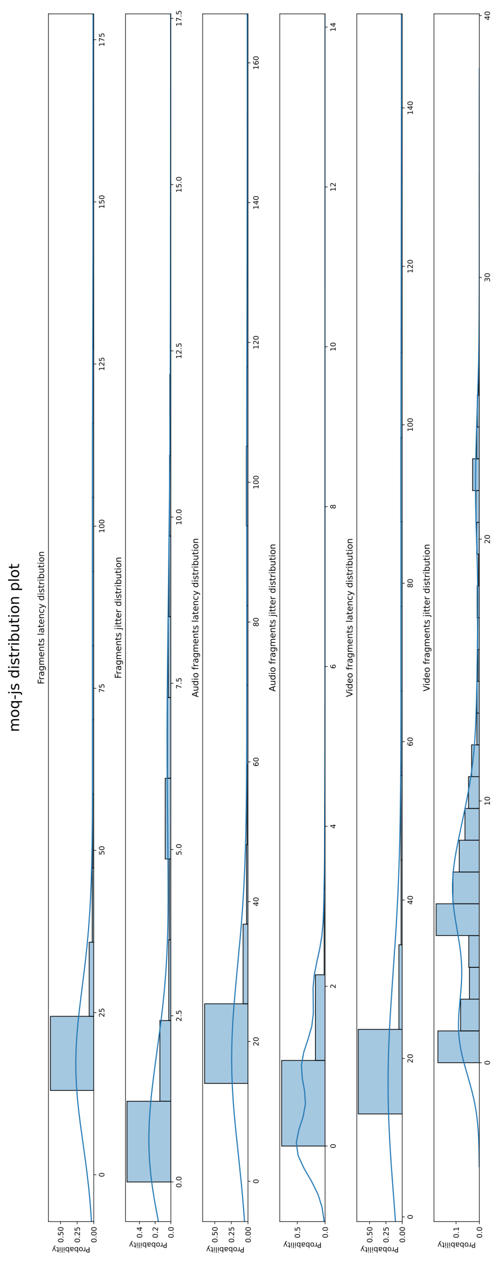


Figure 8.19. Stream mode latency distribution.

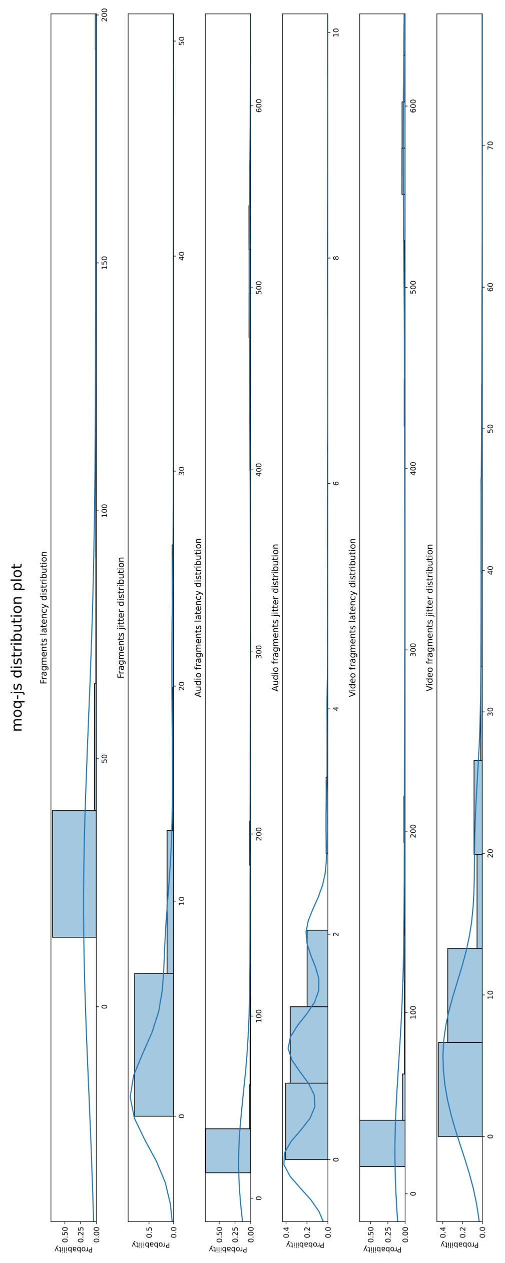


Figure 8.20. Datagram mode latency distribution.

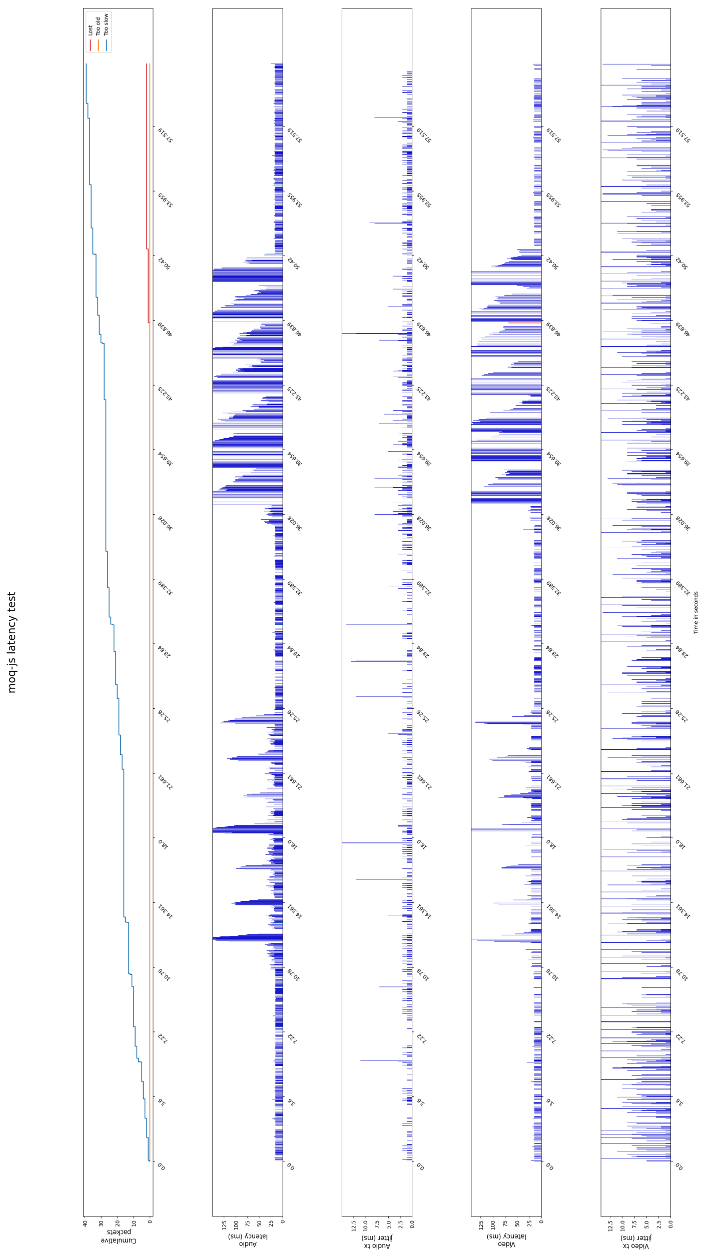


Figure 8.21. Stream mode cumulative packet loss.

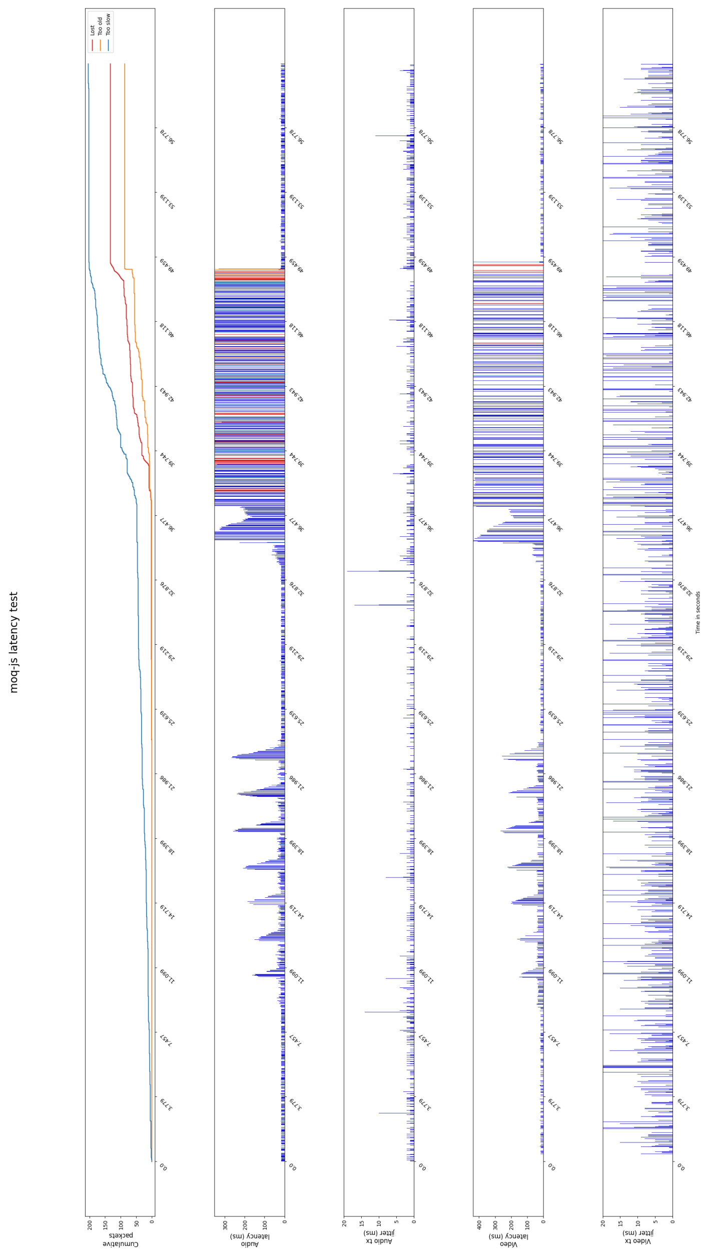


Figure 8.22. Datagram mode cumulative packet loss.

Table 8.31. Bandwidth Usage Summary for a128 v1000 2→1.5M (1) scenario (stream and datagram).

Scenario	Mode	Direction	Total (MB)	Average (kB/s)	Max (kB/s)
a128 v1000 2→1.5M (1)	Stream	Download	10.63	184.49	384.85
	Stream	Upload	11.86	205.92	455.62
	Datagram	Download	11.33	196.65	486.84
	Datagram	Upload	12.14	210.66	455.61

Scenario	Mode	LOST	LOST (%)	TOO OLD	TOO OLD (%)	TOO SLOW	TOO SLOW (%)	Not rec.	Not rec. (%)	Total
a128 v1000 2→1.5M (1)	Stream	2	0.04	0	0.00	39	0.79	41	0.83	4920
	Datagram	133	2.60	87	1.70	204	3.98	424	8.28	5120

Table 8.32. Packet loss statistics for audio 128 kbps, video 1000 kbps, bandwidth from 2 Mbit/s to 1.5 Mbit/s (test 1).

Empirical observations:

At moderate video bitrates and bandwidth drops, stream mode maintains negligible lost packets and low lateness.

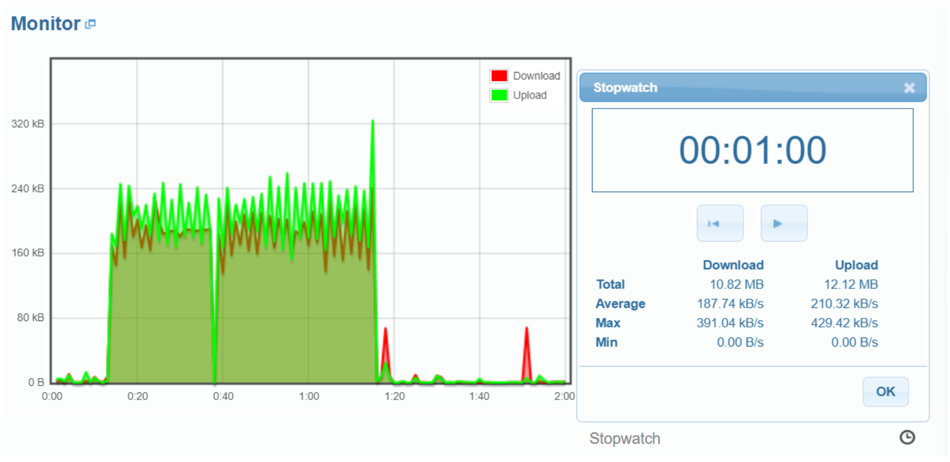
Datagram mode shows higher loss and latency, indicating greater vulnerability to bandwidth constraints and timing.

In datagram mode, the video stream exhibits frame drops and, occasionally, freezing or artifacts when the bandwidth drops to 1.5 Mbit/s, with playback not always recovering immediately after restrictions are lifted.

Stream mode maintains video and audio delivery without freezing, although some stuttering and minor delays are observed under constraint.

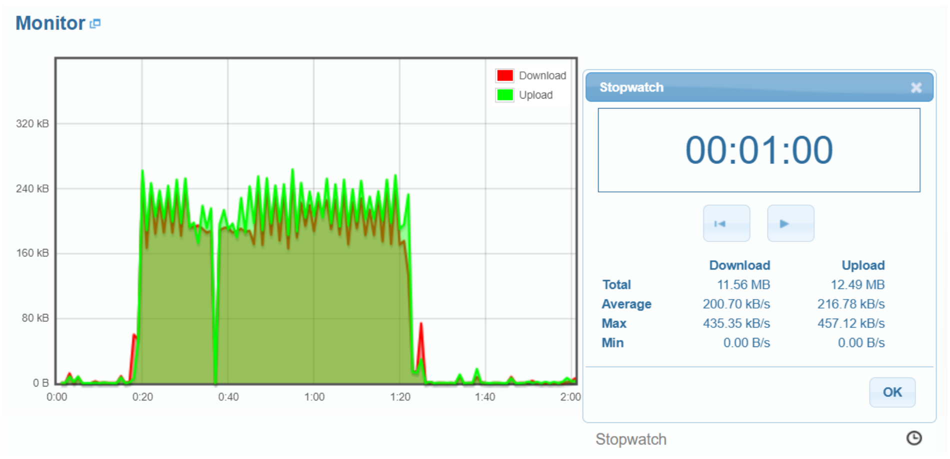
8.6.3 Scenario: Audio 128 kbps, Video 1000 kbps, Bandwidth limited 2 Mbit/s to 1.5 Mbit/s (Test 2)

Media	Opus audio (128 kbps)
Video	VP8 video (1000 kbps)
Total duration	60 s
Bandwidth pattern	10 s unlimited → 15 s @2 Mbps → 10 s unlimited → 15 s @1.5 Mbps
Other impairments	None



	Download	Upload
Total	10.82 MB	12.12 MB
Average	187.74 kB/s	210.32 kB/s
Max	391.04 kB/s	429.42 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.23. Stream mode bandwidth usage (60s).



	Download	Upload
Total	11.56 MB	12.49 MB
Average	200.70 kB/s	216.78 kB/s
Max	435.35 kB/s	457.12 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.24. Datagram mode bandwidth usage (60s).

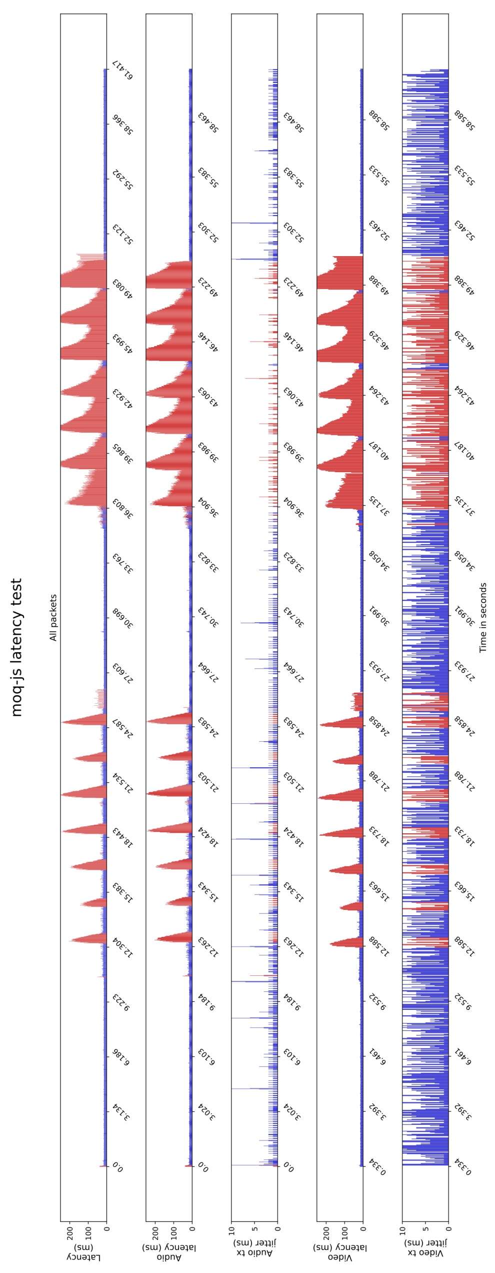


Figure 8.25. Stream mode latency visualization (latency over time).

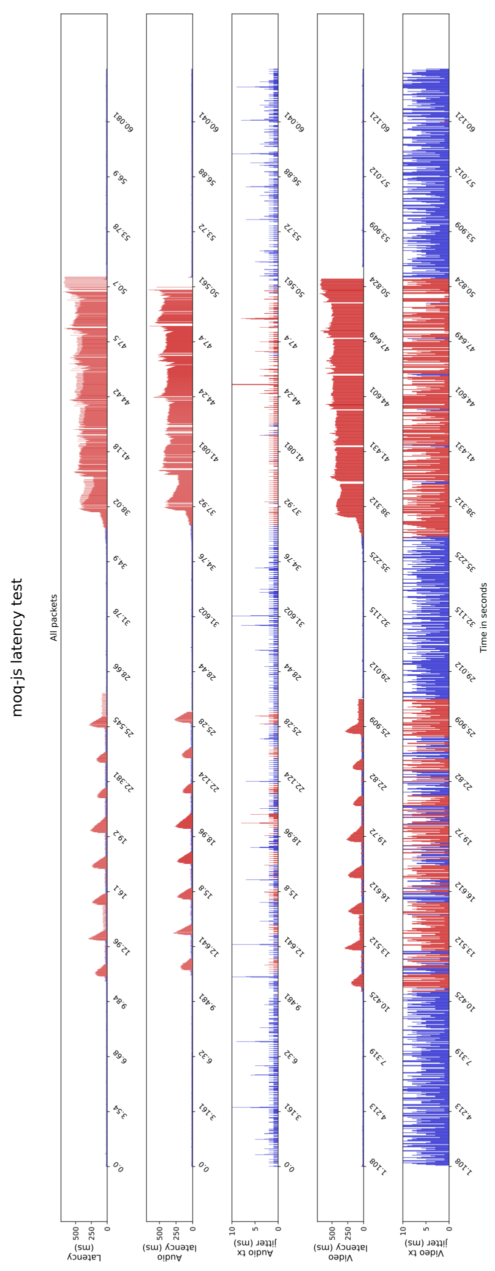


Figure 8.26. Datagram mode latency visualization (latency over time).

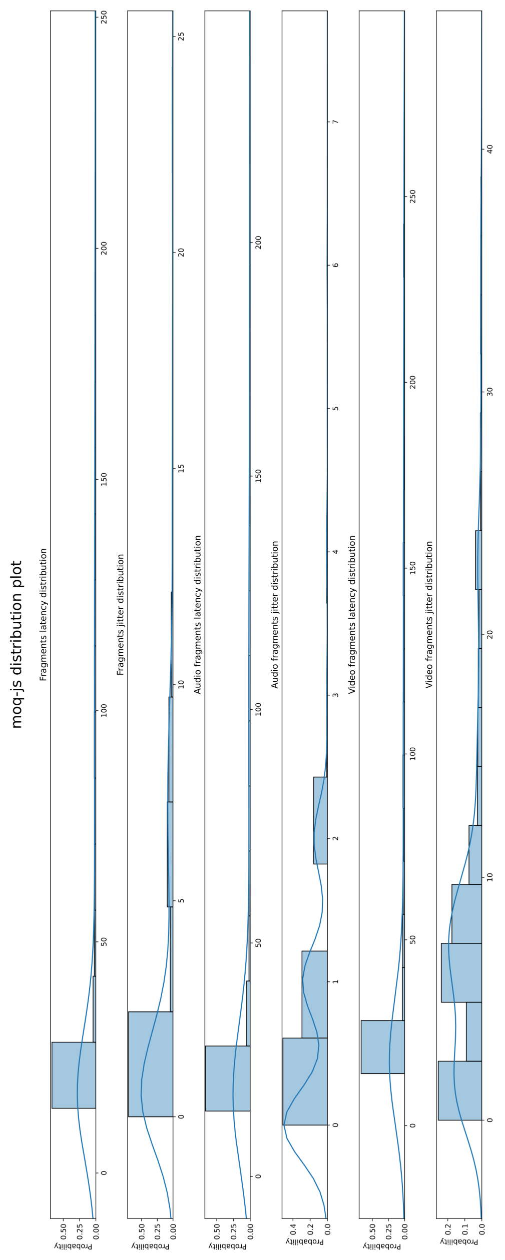


Figure 8.27. Stream mode latency distribution.

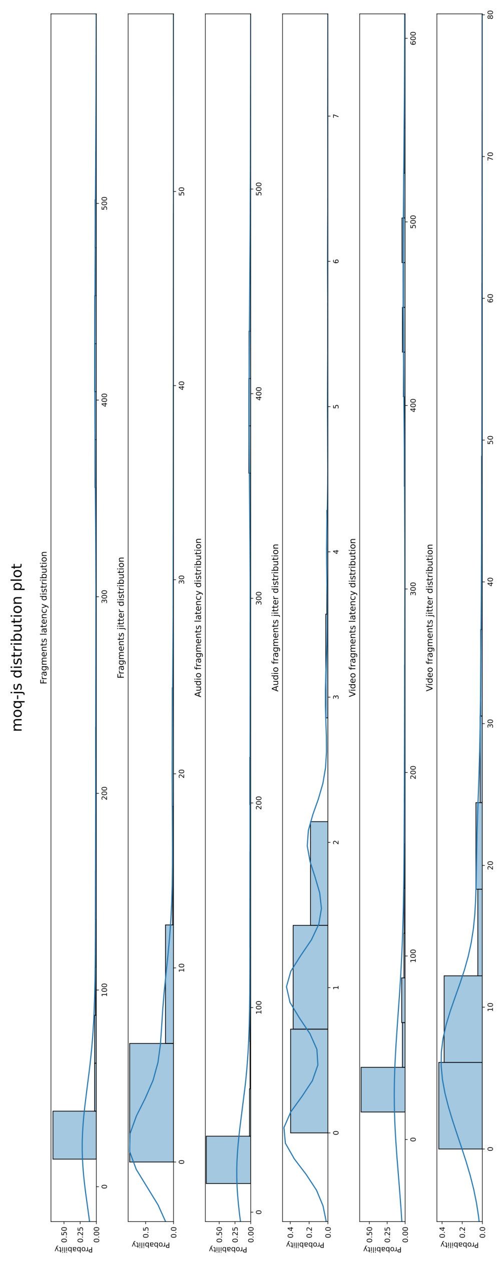


Figure 8.28. Datagram mode latency distribution.

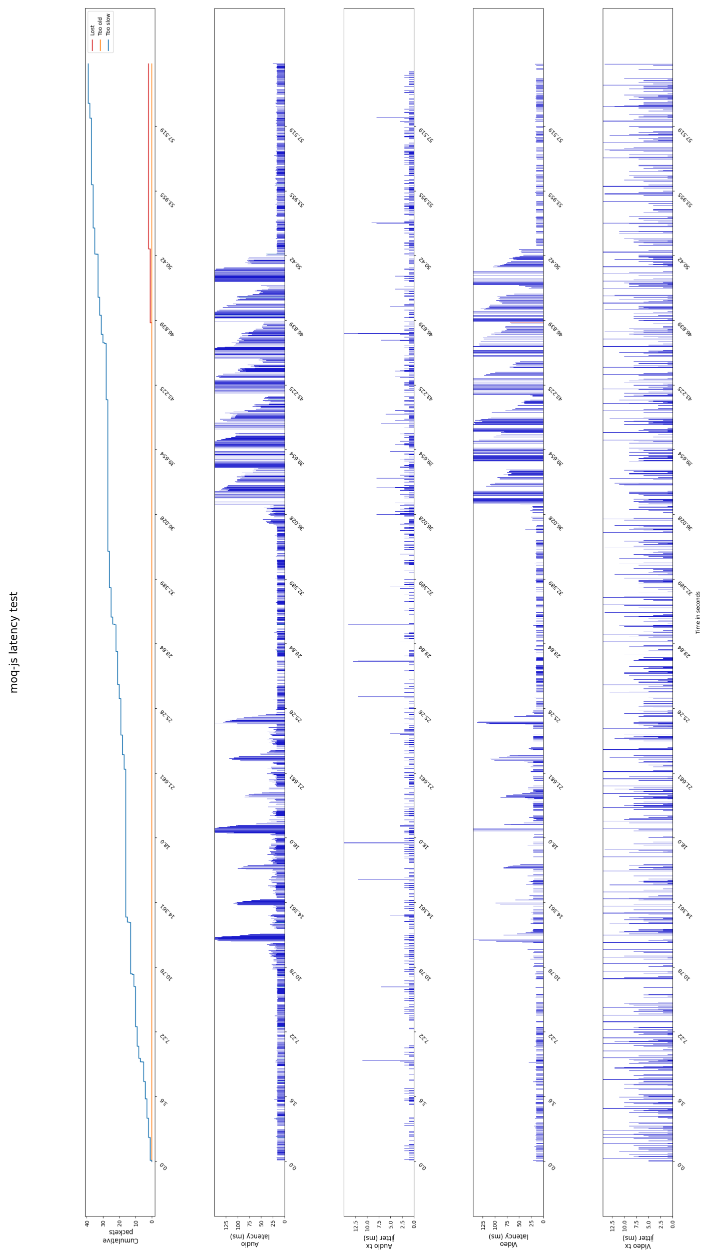


Figure 8.29. Stream mode cumulative packet loss.

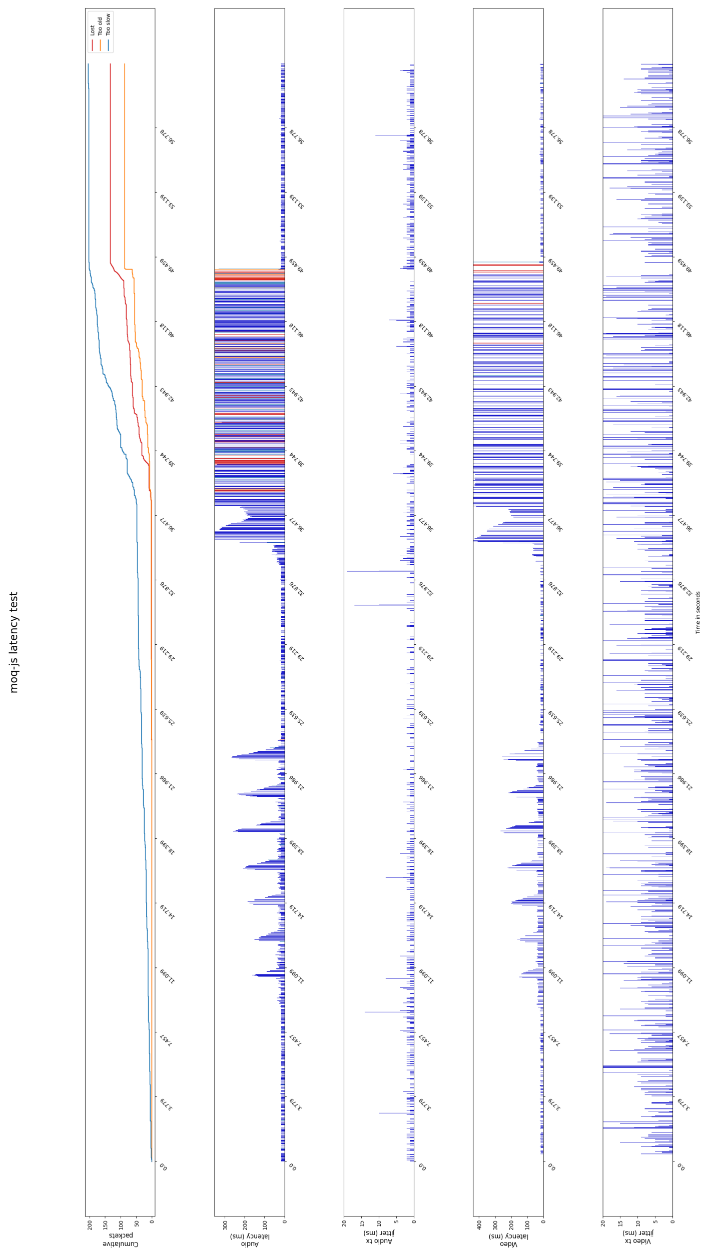


Figure 8.30. Datagram mode cumulative packet loss.

Table 8.33. Bandwidth Usage Summary for a128 v1000 2→1.5M (2) scenario (stream and datagram).

Scenario	Mode	Direction	Total (MB)	Average (kB/s)	Max (kB/s)
a128 v1000 2→1.5M (2)	Stream	Download	10.82	187.74	391.04
	Stream	Upload	12.12	210.32	429.42
	Datagram	Download	11.56	200.70	435.35
	Datagram	Upload	12.49	216.78	457.12

Scenario	Mode	LOST	LOST (%)	TOO OLD	TOO OLD (%)	TOO SLOW	TOO SLOW (%)	Not rec.	Not rec. (%)	Total
a128 v1000 2→1.5M (2)	Stream	6	0.12	7	0.14	17	0.35	30	0.61	4909
	Datagram	138	2.66	89	1.71	176	3.39	403	7.76	5195

Table 8.34. Packet loss statistics for audio 128 kbps, video 1000 kbps, bandwidth from 2 Mbit/s to 1.5 Mbit/s (test 2).

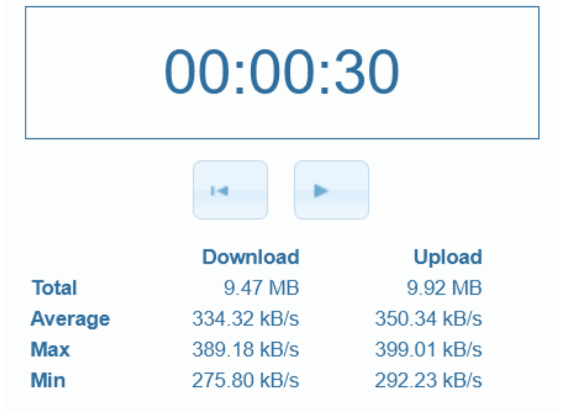
Empirical observations:

Datagram mode again displays higher sensitivity to bandwidth caps, with video freezes and incomplete recovery after limits are lifted. Stream mode playback stutters under constraint but quickly returns to normal when bandwidth is restored.

This second test run confirms prior observations: stream mode is resilient with low loss rates, while datagram mode shows higher packet loss and delay-related errors impacting stream integrity.

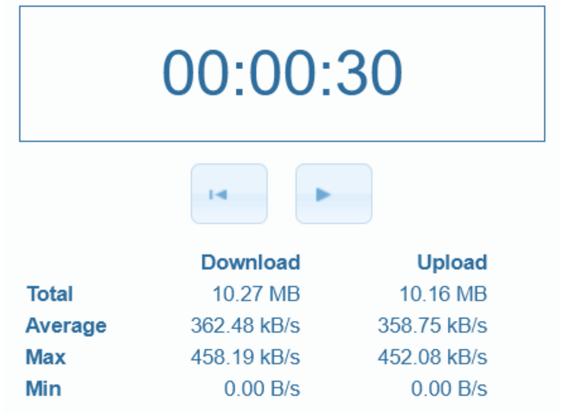
Scenario: Audio 128 kbps, Video 2000 kbps, No Bandwidth Limits

Media	Opus audio (128 kbps)
Video	VP8 video (2000 kbps)
Total duration	30 s
Bandwidth pattern	Unlimited
Other impairments	None



	Download	Upload
Total	9.47 MB	9.92 MB
Average	334.32 kB/s	350.34 kB/s
Max	389.18 kB/s	399.01 kB/s
Min	275.80 kB/s	292.23 kB/s

Figure 8.31. Stream mode bandwidth usage (30s).



	Download	Upload
Total	10.27 MB	10.16 MB
Average	362.48 kB/s	358.75 kB/s
Max	458.19 kB/s	452.08 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.32. Datagram mode bandwidth usage (30s).

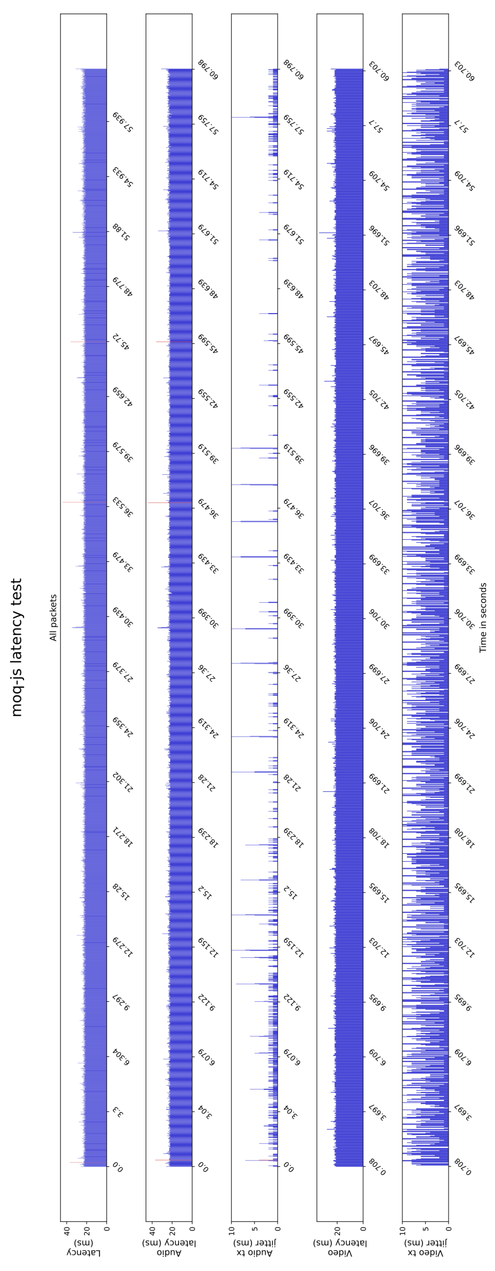


Figure 8.33. Stream mode latency visualization (no limits).

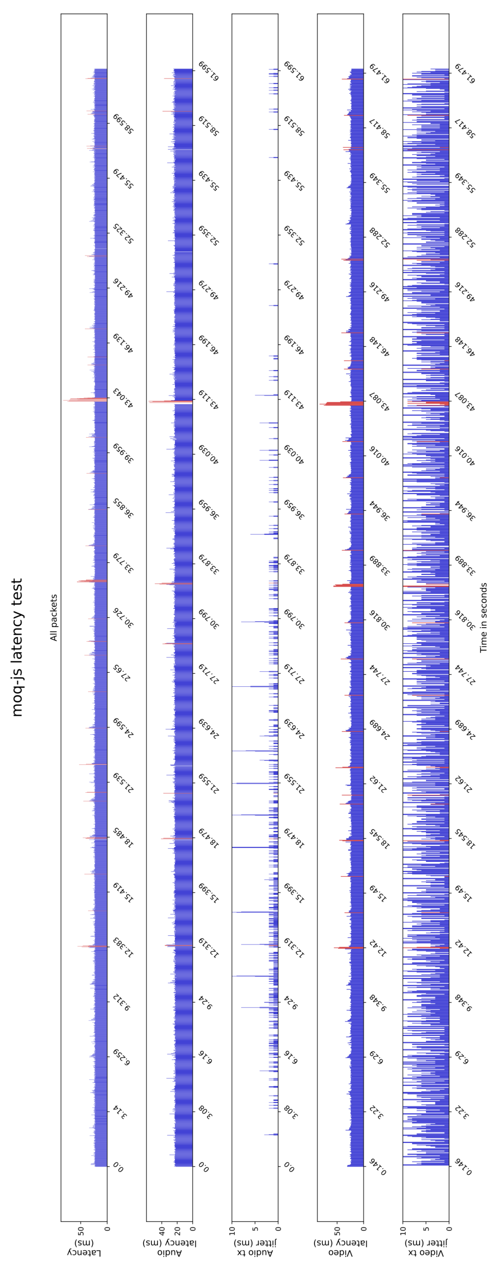


Figure 8.34. Datagram mode latency visualization (no limits).

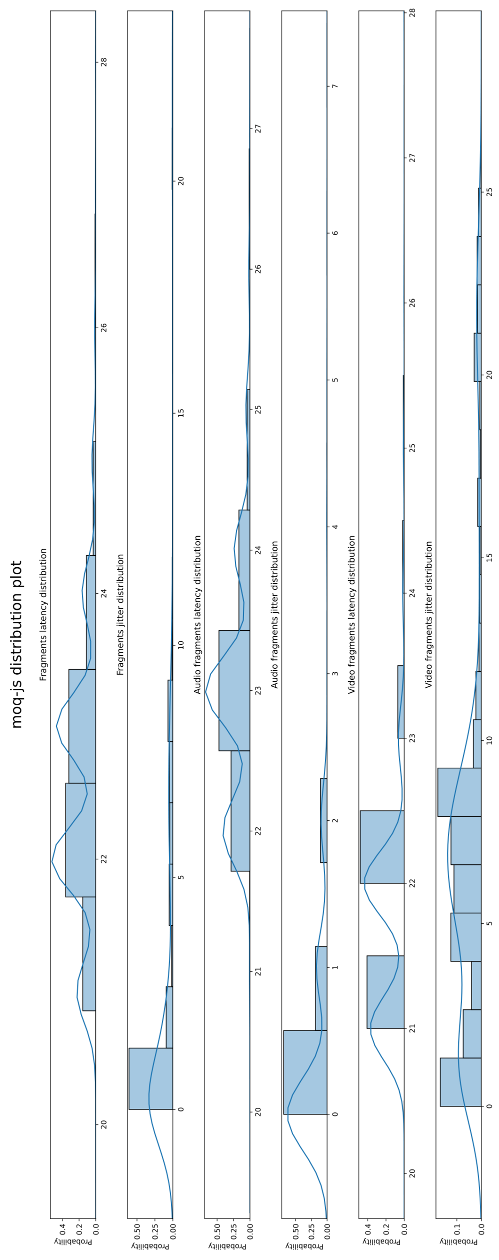


Figure 8.35. Stream mode latency distribution (no limits).

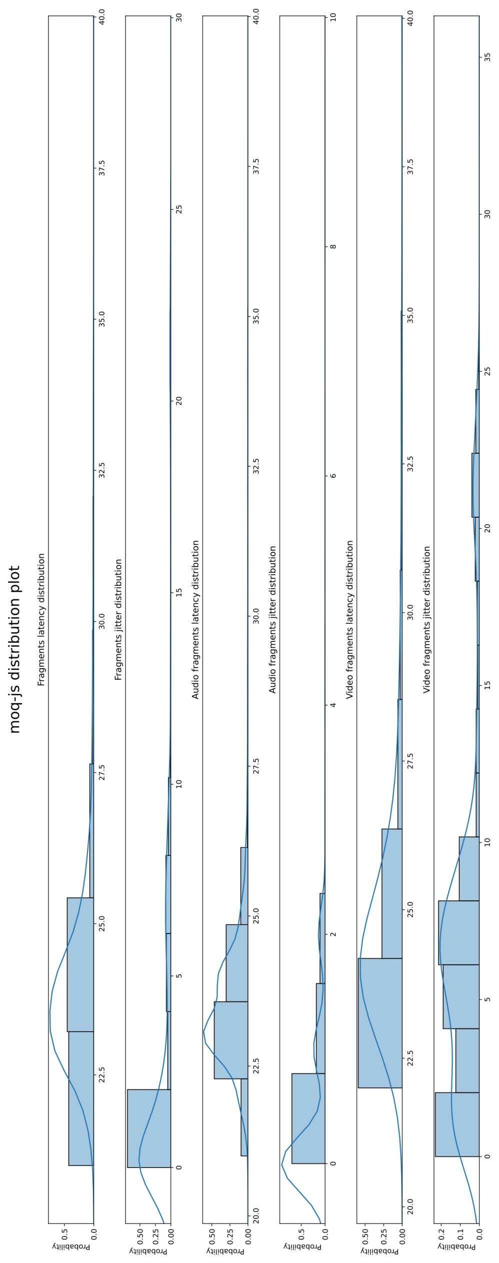


Figure 8.36. Datagram mode latency distribution (no limits).

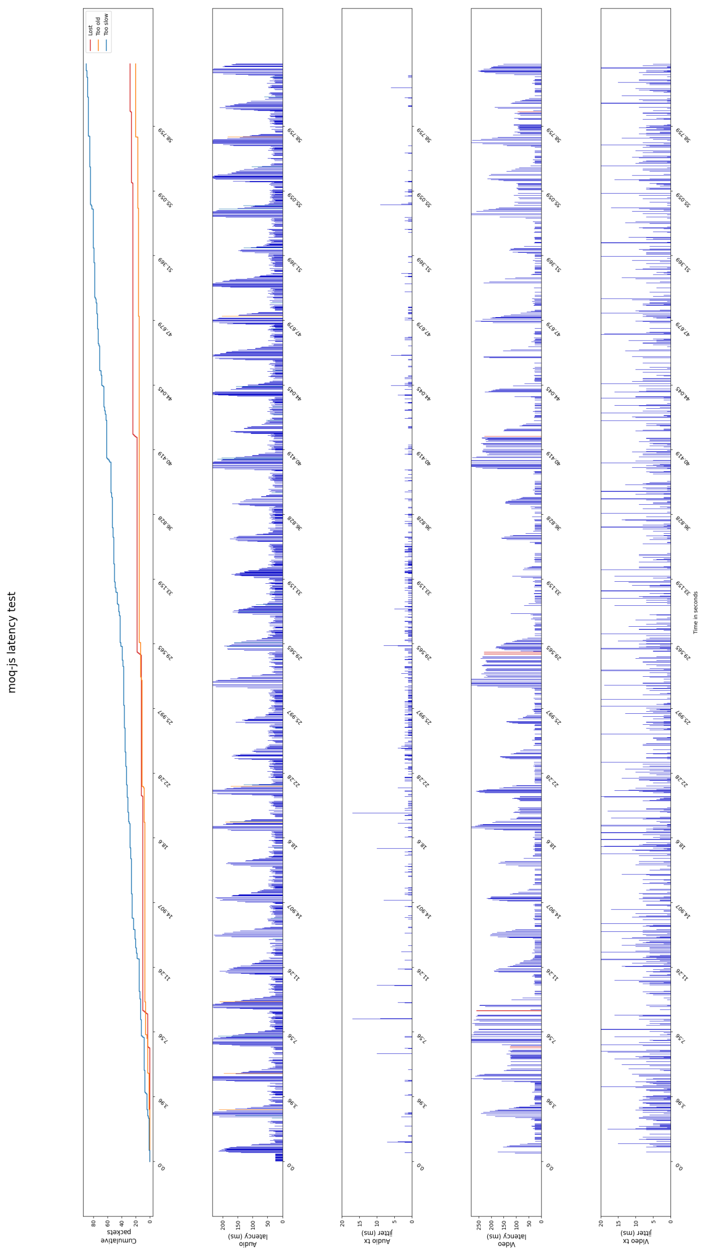


Figure 8.37. Stream mode cumulative packet loss.

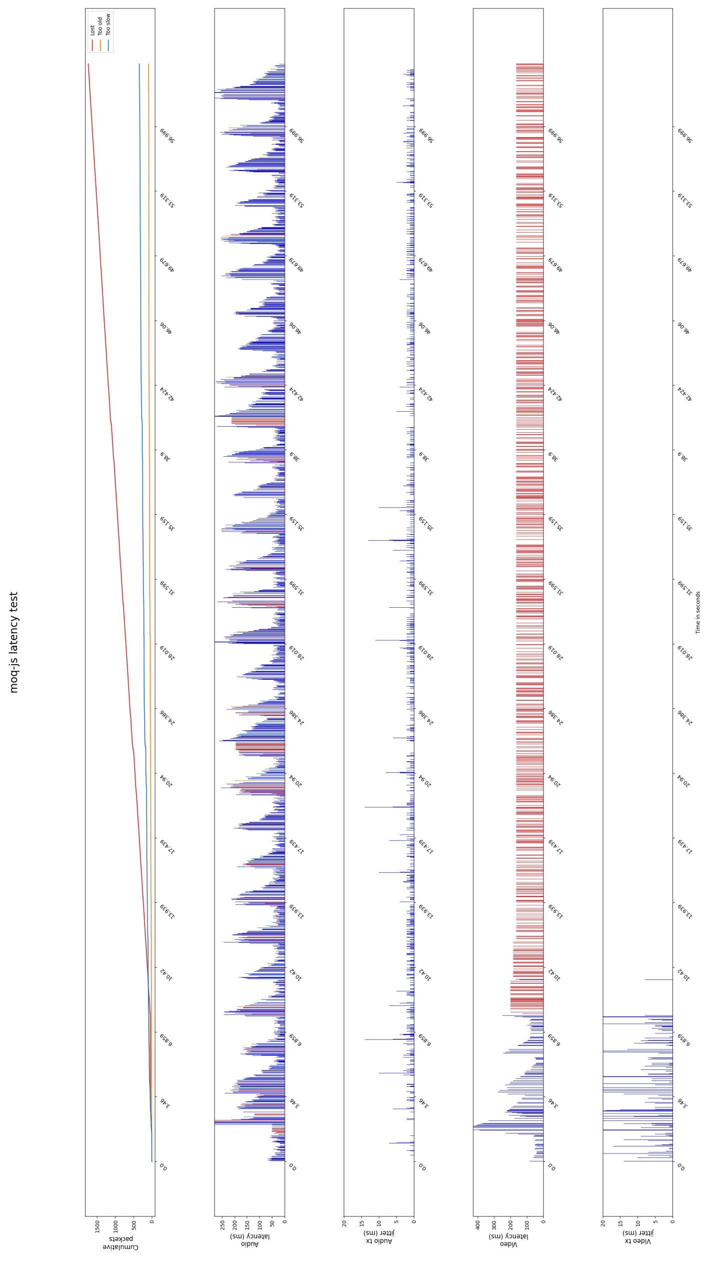


Figure 8.38. Datagram mode cumulative packet loss.

Table 8.35. Bandwidth Usage Summary for a128 v2000 3M scenario (stream and datagram).

Scenario	Mode	Direction	Total (MB)	Average (kB/s)	Max (kB/s)
a128 v2000 3M	Stream	Download	9.47	334.32	389.18
	Stream	Upload	9.92	350.34	399.01
	Datagram	Download	10.27	362.48	458.19
	Datagram	Upload	10.16	358.75	452.08

Scenario	Mode	LOST	LOST (%)	TOO OLD	TOO OLD (%)	TOO SLOW	TOO SLOW (%)	Not rec.	Not rec. (%)	Total
a128 v2000 3M	Stream	28	0.55	20	0.39	90	1.77	138	2.71	5083
	Datagram	1741	33.00	89	1.69	343	6.50	2173	41.19	5276

Table 8.36. Packet loss statistics for audio 128 kbps, video 2000 kbps, 3 Mbit/s bandwidth.

Empirical observations:

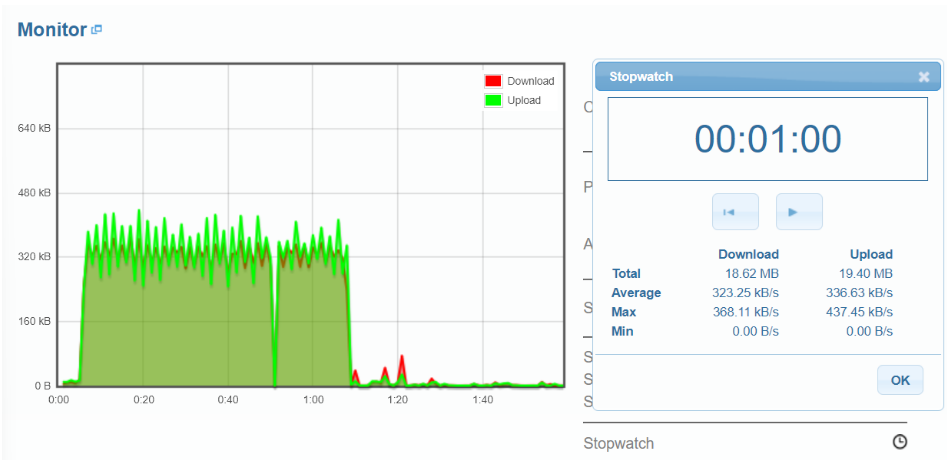
Without bandwidth constraints, both stream and datagram modes perform well, providing smooth playback with minimal latency.

Stream mode stands out for its regularity and efficiency, while, datagram mode still shows slightly more variability in instantaneous bandwidth usage and sporadic packet loss peaks, which do not perceptibly affect playback but suggest lower efficiency and reliability.

Despite minor extra variability, datagram mode still delivers a satisfactory experience under ideal conditions.

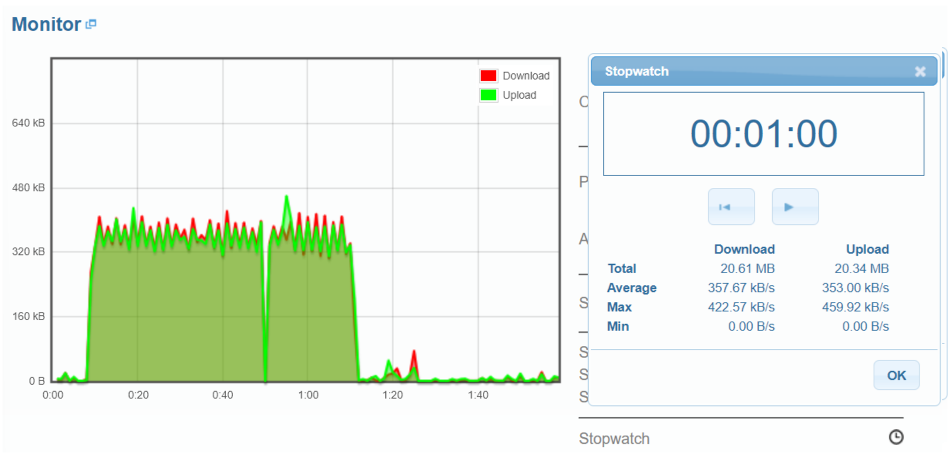
8.6.4 Scenario: Audio 128 kbps, Video 2000 kbps, Bandwidth limited 4 Mbit/s to 3 Mbit/s

Media	Opus audio (128 kbps)
Video	VP8 video (2000 kbps)
Total duration	60 s
Bandwidth pattern	10 s unlimited → 15 s @4 Mbps → 10 s unlimited → 15 s @3 Mbps
Other impairments	None



	Download	Upload
Total	18.62 MB	19.40 MB
Average	323.25 kB/s	336.63 kB/s
Max	368.11 kB/s	437.45 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.39. Stream mode bandwidth usage (60s).



	Download	Upload
Total	20.61 MB	20.34 MB
Average	357.67 kB/s	353.00 kB/s
Max	422.57 kB/s	459.92 kB/s
Min	0.00 B/s	0.00 B/s

Figure 8.40. Datagram mode bandwidth usage (60s).

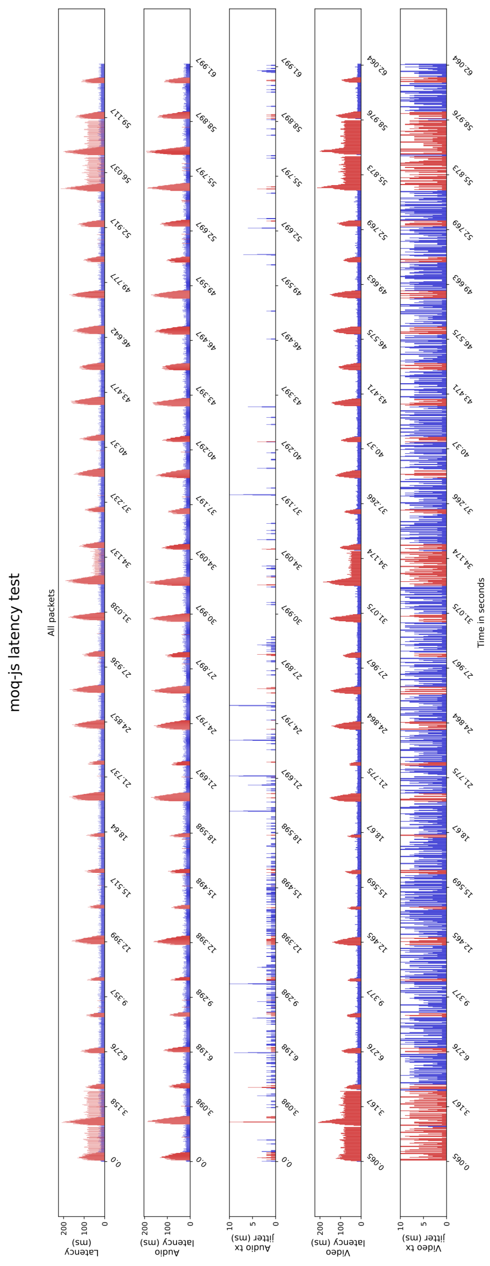


Figure 8.41. Stream mode latency visualization (4 Mbit/s limit).

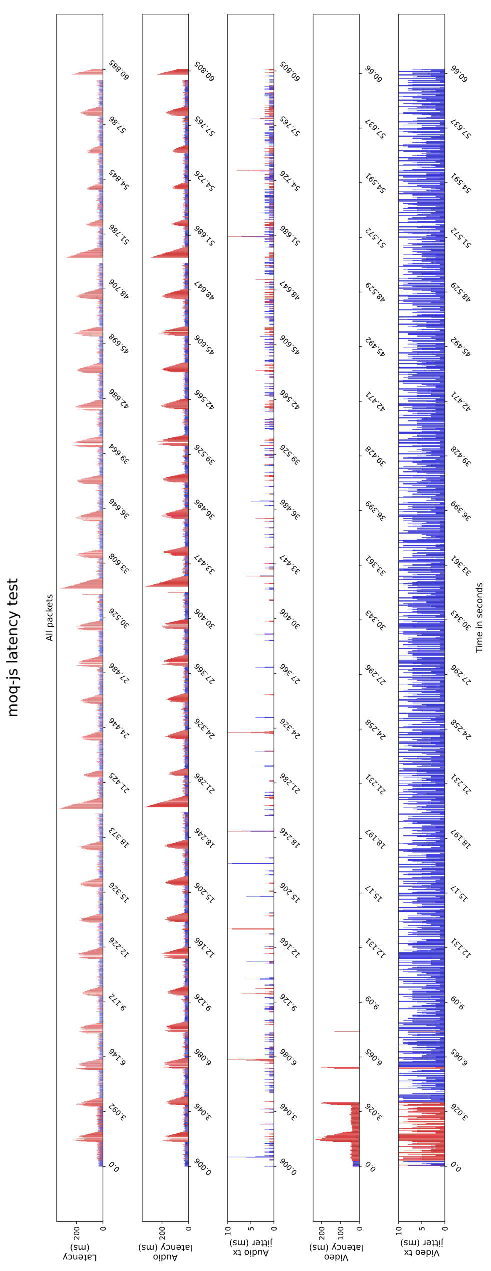


Figure 8.42. Datagram mode latency visualization (4 Mbit/s limit).

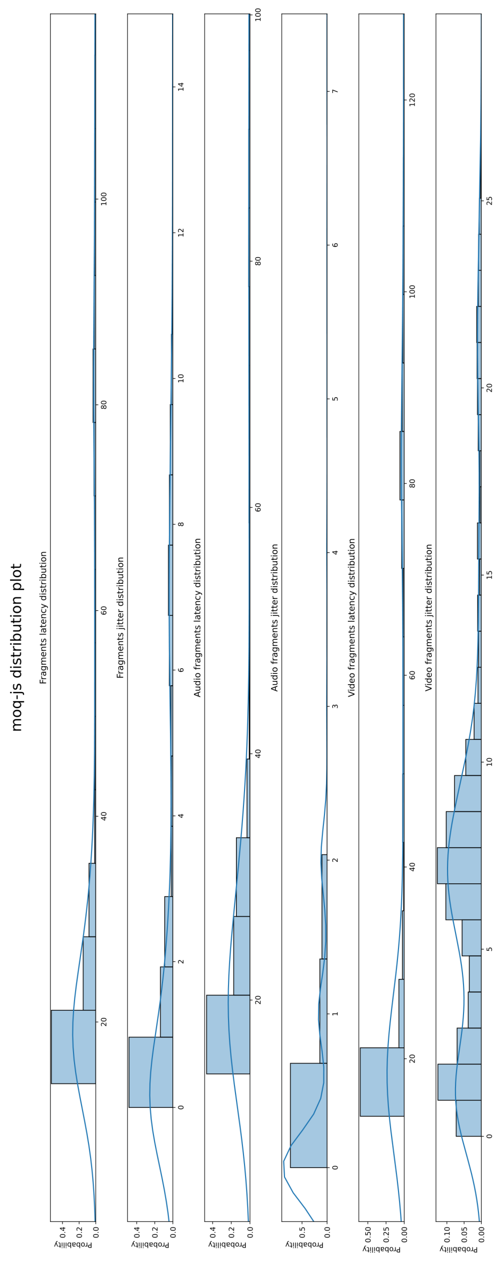


Figure 8.43. Stream mode latency distribution (4 Mbit/s limit).

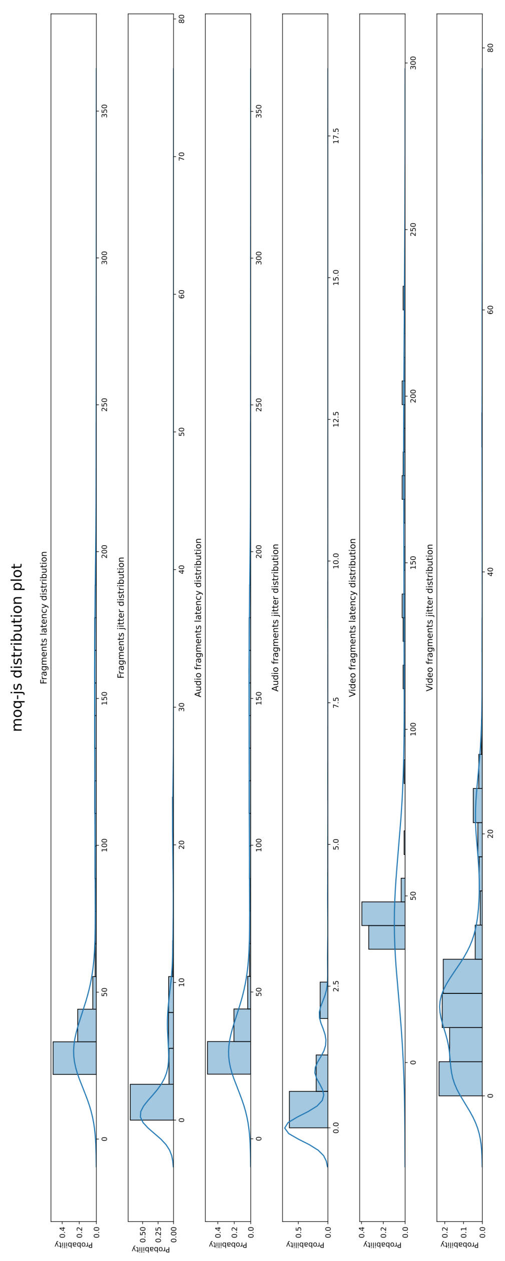


Figure 8.44. Datagram mode latency distribution (4 Mbit/s limit).

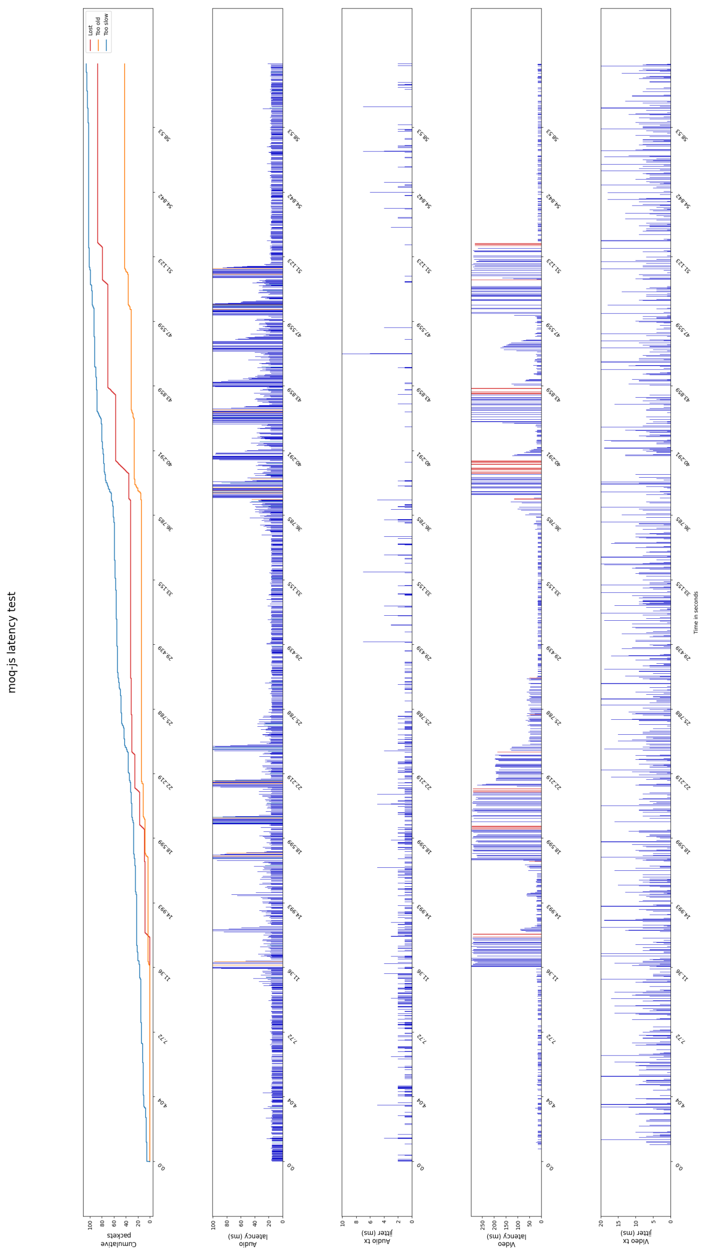


Figure 8.45. Stream mode cumulative packet loss.

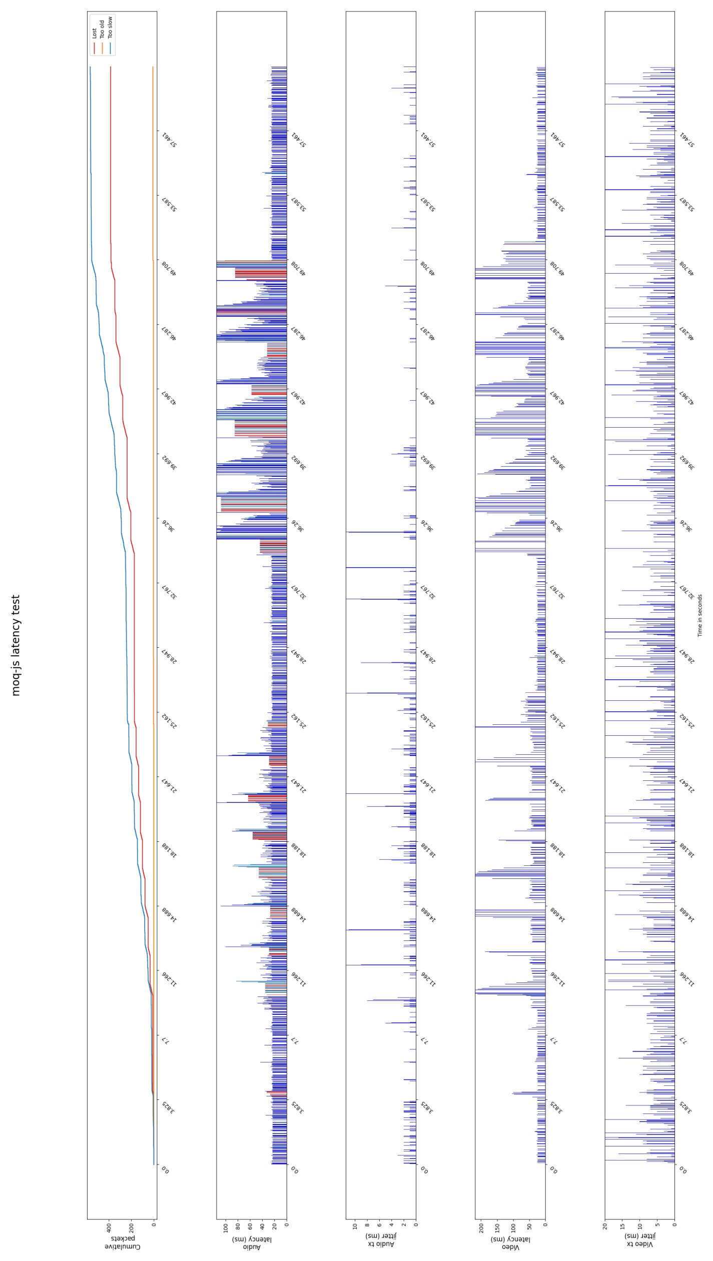


Figure 8.46. Datagram mode cumulative packet loss.

Table 8.37. Bandwidth Usage Summary for a128 v2000 4→3M scenario (stream and datagram).

Scenario	Mode	Direction	Total (MB)	Average (kB/s)	Max (kB/s)
a128 v2000 4→3M	Stream	Download	18.62	323.25	368.11
	Stream	Upload	19.40	336.63	437.45
	Datagram	Download	20.61	357.67	422.57
	Datagram	Upload	20.34	353.00	459.92

Scenario	Mode	LOST	LOST (%)	TOO OLD	TOO OLD (%)	TOO SLOW	TOO SLOW (%)	Not rec.	Not rec. (%)	Total
a128 v2000 4→3M	Stream	87	1.71	42	0.83	106	2.08	235	4.62	5089
	Datagram	383	6.98	8	0.15	564	10.28	955	17.41	5486

Table 8.38. Packet loss statistics for audio 128 kbps, video 2000 kbps, bandwidth reduced from 4 Mbit/s to 3 Mbit/s.

Empirical observations:

Bandwidth drops have a clear negative impact on datagram mode playback, which is more affected by freezing and stalling, especially as bandwidth drops to 3 Mbit/s.

Video can freeze and does not always recover upon removal of the cap, while stream mode is more robust, only suffering minor playback delays and quickly returning to normal after constraints are lifted.

Datagram mode suffers greater packet loss and latency issues, confirming its reduced robustness under bandwidth constraints compared to stream mode.

8.6.5 Global Bandwidth Usage Summary and Analysis

The table below presents a consolidated overview of **bandwidth utilization** across all experimental scenarios, encompassing both **Stream** and **Datagram transmission modes**.

For each scenario, **total**, **average**, and **peak bandwidth usage** values are reported separately for **upload** and **download** directions, providing a detailed perspective on resource consumption under varying network and media conditions.

Table 8.39. Summary of bandwidth usage statistics across all scenarios.

Scenario	Mode	Direction	Total (MB)	Average (kB/s)	Max (kB/s)
a128 400k→300k	Stream	Download	2.36	40.91	86.46
		Upload	3.98	69.04	149.35
	Datagram	Download	2.89	50.17	173.07
		Upload	4.22	73.26	166.11
a128 500k→250k	Stream	Download	2.38	41.34	243.20
		Upload	3.82	66.35	218.06
	Datagram	Download	2.58	44.08	191.34
		Upload	4.22	72.08	221.96
a128 v1000 2→1.5M (1)	Stream	Download	10.63	184.49	384.85
		Upload	11.86	205.92	455.62
	Datagram	Download	11.33	196.65	486.84
		Upload	12.14	210.66	455.61
a128 v1000 2→1.5M (2)	Stream	Download	10.82	187.74	391.04
		Upload	12.12	210.32	429.42
	Datagram	Download	11.56	200.70	435.35
		Upload	12.49	216.78	457.12
a128 v2000 3M	Stream	Download	9.47	334.32	389.18
		Upload	9.92	350.34	399.01
	Datagram	Download	10.27	362.48	458.19
		Upload	10.16	358.75	452.08
a128 v2000 4→3M	Stream	Download	18.62	323.25	368.11
		Upload	19.40	336.63	437.45
	Datagram	Download	20.61	357.67	422.57
		Upload	20.34	353.00	459.92

Examining the results, several notable trends become evident. First, **datagram-based transmission consistently requires greater bandwidth** than stream-based transmission across all tested scenarios.

This effect is most pronounced in configurations featuring **higher media bitrates**—particularly those including both **audio and video streams**—where datagram overhead becomes more significant due to redundant framing and custom media reassembly mechanisms at the application layer.

Even in **lower-bitrate, audio-only scenarios**, datagram mode exhibits a **non-negligible increase in bandwidth usage** compared to stream mode.

Additionally, the difference in **peak bandwidth usage** between the two modes is particularly striking. **Datagram mode** frequently produces **higher instantaneous bandwidth spikes**, a behavior that may amplify the risk of **congestion and packet loss** in networks with limited or variable capacity. **Stream mode**, by contrast, tends to exhibit **smoother and more predictable bandwidth usage profiles**, with lower peaks and reduced variance over time.

The summary provided in Table 8.39 serves as a **quantitative foundation** for these observations, and informs the comparative analysis of protocol performance throughout this chapter.

8.6.6 Bandwidth Usage Comparison Chart

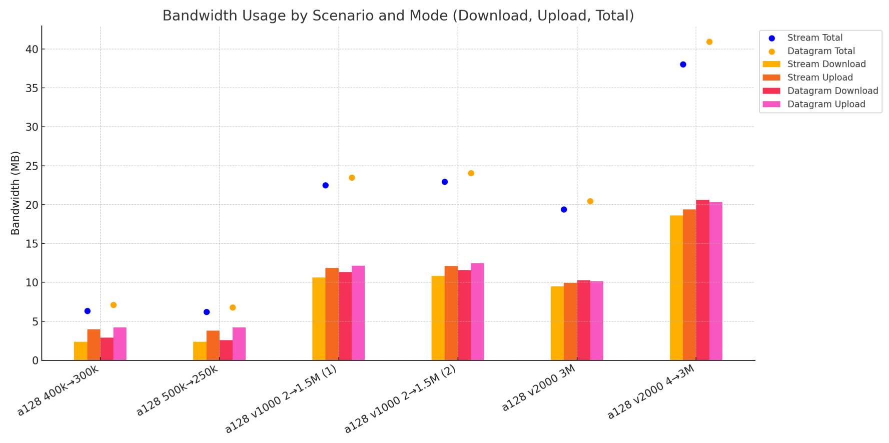


Figure 8.47. Comparison of total bandwidth usage (download + upload) across scenarios for Stream and Datagram modes.

8.6.7 Global Packet Loss Summary and Analysis

The following table presents a comprehensive, scenario-by-scenario analysis of packet loss characteristics across all experimental conditions, providing both absolute counts and normalized percentages for each loss category.

The table disaggregates results for the two evaluated transport modes—**Stream** and **Datagram**—to highlight protocol-specific behaviors under diverse real-time media workloads and network stressors.

Table 8.40. Summary of packet loss statistics across all scenarios (percentages refer to total packets sent).

Scenario	**Mode**	**LOST**	**TOO OLD**	**TOO SLOW**	**Not rec.**	**LOST** (%)	**TOO OLD** (%)	**TOO SLOW** (%)	**Not rec.** (%)	**Total**
a128 400k→300k	Stream	0	42	16	58	0.00	1.31	0.50	1.81	3200
	Datagram	204	292	111	607	6.04	8.64	3.28	17.96	3380
a128 500k→250k	Stream	1	427	70	498	0.03	12.14	1.99	14.16	3517
	Datagram	233	471	119	823	6.53	13.20	3.34	23.07	3568
a128 v1000 2→1.5M (1)	Stream	2	0	39	41	0.04	0.00	0.79	0.83	4920
	Datagram	133	87	204	424	2.60	1.70	3.98	8.28	5120
a128 v1000 2→1.5M (2)	Stream	6	7	17	30	0.12	0.14	0.35	0.61	4909
	Datagram	138	89	176	403	2.66	1.71	3.39	7.76	5195
a128 v2000 3M	Stream	28	20	90	138	0.55	0.39	1.77	2.71	5083
	Datagram	1741	89	343	2173	33.00	1.69	6.50	41.19	5276
a128 v2000 4→3M	Stream	87	42	106	235	1.71	0.83	2.08	4.62	5089
	Datagram	383	8	564	955	6.98	0.15	10.28	17.41	5486
Mean	Stream	20.67	89.67	56.33	166.67	0.41	2.13	1.41	3.59	4451
	Datagram	455.67	171.00	236.83	717.50	9.41	4.52	5.30	15.45	4630

Column interpretation and loss categories:

- **LOST**: Packets that were never received at the destination, representing pure data loss due to severe network impairment or protocol inefficiency.
- **TOO OLD**: Packets delivered so late that their playback deadline had passed, making them unusable and directly impacting perceived smoothness, especially in interactive or low-latency applications.
- **TOO SLOW**: Packets delivered with significant delay but still within their playable window—potentially leading to jitter, minor stalling, or perceptible delays.
- **Not Received**: The sum of the above categories, indicating all packets failing to contribute to successful playback, whether due to outright loss or excessive latency. This serves as the primary indicator of end-user Quality of Experience (QoE) degradation.

- All percentages are normalized over the total number of packets sent in each scenario, enabling fair comparison across highly variable test configurations.

Detailed scenario trends:

- **Low Bitrate Audio Scenarios (a128 400k→300k, a128 500k→250k):** For both stream and datagram, absolute numbers of lost packets are generally lower due to the smaller data volumes. However, even here, datagram mode consistently exhibits significantly higher **LOST** and **Not Received** percentages—up to 23% in the most severe case (500k→250k).

  This confirms that datagram transmission is acutely vulnerable to even modest bandwidth restrictions, while stream mode keeps total loss rates typically below 15% and absolute **LOST** counts at or near zero.

  Notably, stream mode shows increased **TOO OLD** packets under extreme bandwidth scarcity, indicating packets are delayed rather than lost. This behavior—characteristic of QUIC's in-order delivery—may lead to audio jitter or subtle stalls, but is preferable to the abrupt skips and silences caused by outright datagram loss.

- **Audio-Video Scenarios at Moderate Bitrate (a128 v1000 2→1.5M):** With the introduction of video streams, the contrast between protocols becomes more pronounced. Datagram mode routinely loses hundreds of packets per minute, with **Not Received** percentages ranging from 7.76% to 8.28%, and **LOST** rates up to 2.6%. Stream mode remains highly resilient, with **Not Received** rates under 1% and negligible outright loss.

  The low **TOO OLD** and **TOO SLOW** values in these scenarios for stream mode indicate that the protocol is not only preventing loss, but also managing jitter and delay very effectively. In contrast, datagram mode presents a more even spread of late and lost packets, contributing to perceptible degradation in both audio and video playback, especially during or immediately after bandwidth restriction periods.

- **High Bitrate/No Limit (a128 v2000 3M):** This scenario demonstrates the "best case" for both protocols. Here, both stream and datagram achieve very low **LOST** and **Not Received** percentages (stream: 2.71%, datagram: 41.19%), with almost all delivered packets arriving in time for playback. Nevertheless, the **Not Received** rate for datagram remains alarmingly high for a scenario with ample bandwidth, indicating structural inefficiency or protocol overhead in datagram mode.

  The fact that stream mode's loss rates remain below 3% even at high throughput further attests to the inherent robustness of stream-based transport for bulk media.

- **High Bitrate + Restriction (a128 v2000 4→3M):** In the most demanding scenario, datagram mode performance degrades sharply, with **Not Received** rates of 17.41% and **LOST** rates nearing 7%. Video playback in these cases is prone to frequent freezes and visible artifacts, while stream mode continues to limit **Not Received** to just 4.62%—demonstrating graceful degradation and rapid recovery after bandwidth is restored.

Interpretative synthesis

Across all tested conditions, the collected evidence clearly shows that **datagram mode is significantly less robust** than stream mode in every practical metric of real-time media reliability.

Specifically, datagram mode consistently experiences **higher rates of outright packet loss, more late or slow packets, and a greater total number of unusable transmissions**. This is well illustrated by the comparison of LOST packet percentages across scenarios (see Figure 8.48).

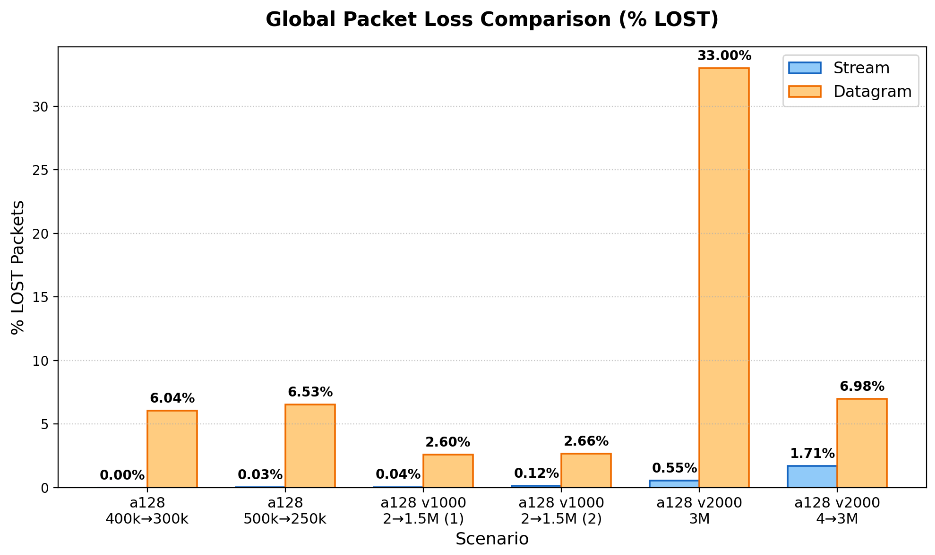


Figure 8.48. Comparison of LOST packet percentage across all scenarios for Stream and Datagram modes.

As depicted in Figure 8.48, datagram mode routinely exhibits LOST rates several times higher than stream mode, particularly in scenarios with more severe bandwidth restrictions or higher media bitrates. For example, in the a128 v2000 3M scenario, the percentage of lost packets in datagram mode exceeds 30%, compared to less than 1% for stream mode.

Furthermore, this gap in robustness becomes even more pronounced as traffic volume increases or bandwidth becomes more constrained. The trend is evident not only in outright losses, but also in the number of packets that arrive **too late** for playback—an aspect that directly impacts the perceived smoothness of media applications. Figure 8.49 shows the comparative incidence of packets arriving too late ("TOO OLD") for each mode and scenario.

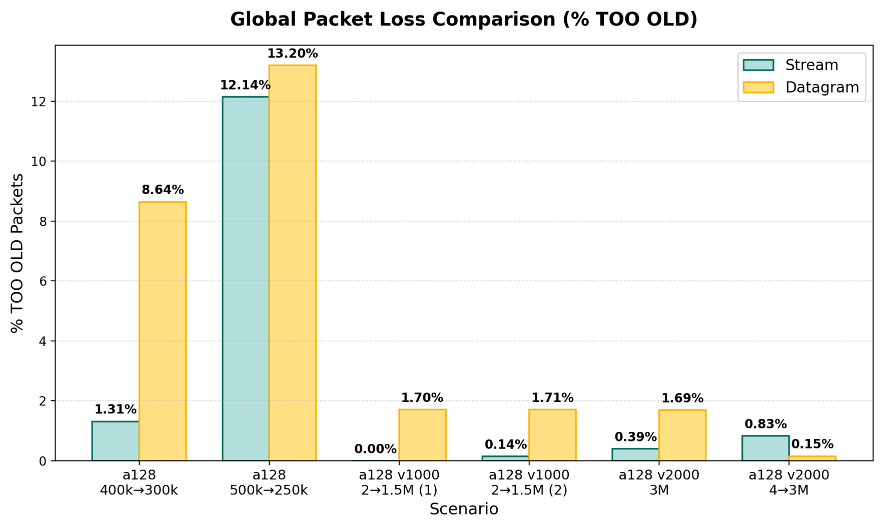


Figure 8.49. Comparison of TOO OLD packet percentage (packets arriving too late) for Stream and Datagram, scenario by scenario.

Here, we observe that stream mode, while generally delivering more packets successfully, may exhibit slightly higher percentages of TOO OLD packets in the most constrained scenarios. This reflects its mechanism of attempting to deliver all packets in order, resulting in some delayed arrivals rather than pure losses.

However, as shown in Figure 8.50, these delayed packets are still usually preferable in terms of user experience, as they lead to brief stalling or jitter, instead of abrupt skips or gaps in playback.

In fact, Figure 8.50 highlights that, in datagram mode, a considerable proportion of packets are marked as TOO SLOW, especially under heavy network load, contributing to noticeable artifacts such as stuttering or freezes in audio and video streams. A comprehensive view of all error categories—LOST, TOO OLD, and TOO SLOW—is provided in Figure 8.51. This visual synthesis confirms that the **total fraction of unusable packets** (including all loss types) is consistently much higher for datagram mode than for stream mode across all test scenarios.

For example, in high-bandwidth audio-video scenarios with network constraints, the total "Not Received" rate in datagram mode can exceed 40% (see also Table 8.40), compared to values below 5% for stream mode. Even under ideal conditions, datagram mode shows occasional spikes in lost and delayed packets that are largely absent in stream-based transmission.

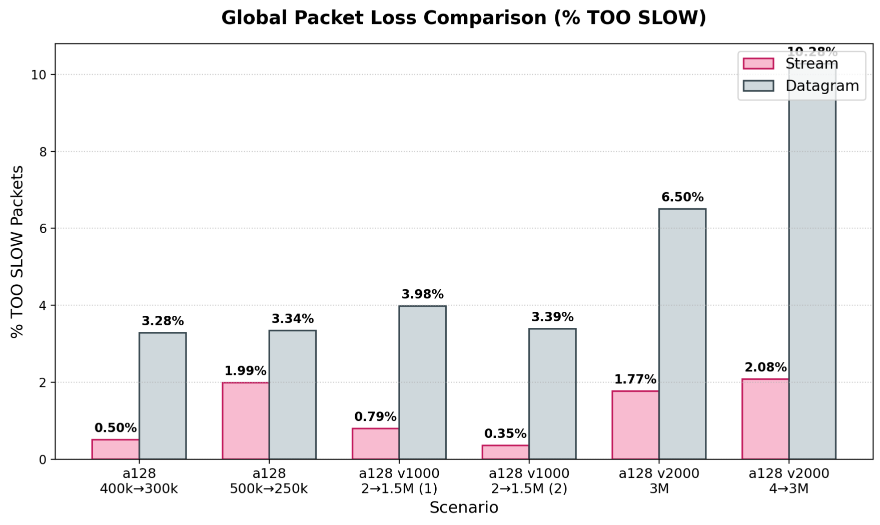


Figure 8.50. Comparison of TOO SLOW packet percentage (packets arriving slowly) for Stream and Datagram, per test scenario.

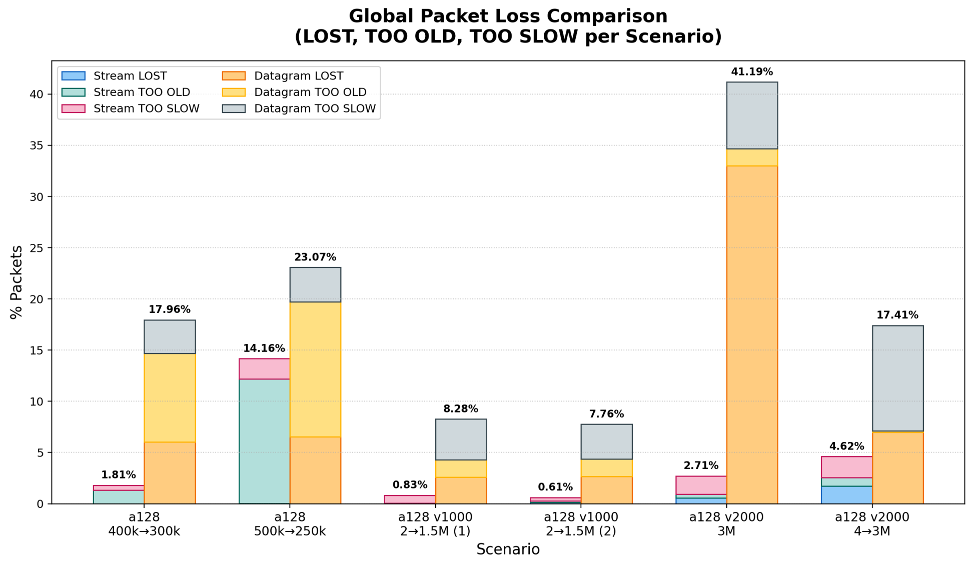


Figure 8.51. Global comparison of LOST, TOO OLD, and TOO SLOW packet percentages for Stream and Datagram transmission across all tested scenarios.

In conclusion, the quantitative and visual evidence provided by these figures and summarized in Table 8.40 make clear that, for any latency- and loss-sensitive application—especially those involving video or mixed media—stream-based transmission offers far greater reliability, predictability, and end-user quality than its datagram counterpart.

The robustness of stream mode is especially evident in the ability to maintain media continuity and minimize outright losses, even as network conditions degrade.

8.7 Evaluation of Experimental Results and Comparative Analysis

8.7.1 Comparative Results Analysis

This section provides a comprehensive and visually guided analysis of the comparative results obtained from all experimental scenarios, systematically contrasting the behavior of QUIC stream and datagram transport modes. The discussion draws extensively from the bandwidth, latency, and loss graphs and tables in Chapter 8, connecting quantitative outcomes to qualitative end-user experience.

Bandwidth Overhead and Protocol Efficiency

A primary observation from the experimental data (see Table 8.39) is that **QUIC datagram mode consistently incurs higher bandwidth usage and peak rates** than stream mode, regardless of media configuration. This effect is evident both in summary (Figure 8.47) and in detailed scenario bandwidth usage plots:

- In low-bitrate audio scenarios, such as **Opus 128 kbps with bandwidth drops** (*a128 400k→300k*, Figures 8.1, 8.2), the datagram mode shows noticeably higher average and peak usage, especially during limited phases. - In combined audio-video scenarios (e.g., **Opus 128 kbps + VP8 1000 kbps**, bandwidth drops), datagram's burstiness is even more pronounced (see Figures 8.15, 8.16).

The difference is particularly marked in the **"Max (kB/s)"** values reported in Table 8.39: datagram mode consistently produces more frequent, larger spikes—sometimes double those seen in stream mode. This pattern reflects both additional protocol overhead (per-packet metadata and custom framing) and a tendency toward burstier traffic patterns due to fragmentation and reassembly at the application level.

Scenario Highlight: High Bitrate, No Limit In unconstrained scenarios (e.g., *a128 v2000 3M*, Figures 8.31, 8.32), datagram mode achieves high throughput but still displays greater instantaneous bandwidth fluctuation—see also the variability in the corresponding latency plots (8.33, 8.34).

Playback Robustness and Error Resilience

A clear qualitative and quantitative gap emerges in playback stability and error resilience under all but the most ideal conditions:

Audio-only Scenarios - **Stream mode** delivers nearly all packets in time, even under bandwidth reduction (*a128 400k→300k*, Table 8.28), leading to continuous playback with only minor stalling (see empirical notes in that section). - **Datagram mode** suffers much greater packet loss and late arrivals, with frequent audio skips and dropouts during restricted periods (see cumulative packet loss plots, Figures 8.8, 8.14).

Mixed Audio-Video Scenarios - In moderate restriction scenarios (e.g., *a128 v1000 2→1.5M*, Figures 8.17, 8.18), stream mode retains smooth playback, with only transient stuttering; datagram mode, by contrast, displays severe frame loss and persistent freezes, as confirmed by cumulative loss graphs (8.22) and distribution plots (8.20).

- In validation runs (*a128 v1000 2→1.5M (2)*, Figures 8.25, 8.26), the same patterns recur, reinforcing the robustness of stream mode and the persistent vulnerability of datagram mode to congestion.

- At the highest tested loads (*a128 v2000 4→3M*), datagram mode exhibits "unrecoverable" video freezes, and extremely high packet loss rates (see Figures 8.46, 8.44), while stream mode continues to recover playback after constraint removal.

Audio Transmission The comparative evaluation for audio is particularly instructive and highlights a counterintuitive result:

- **Datagram Mode**: While datagram mode shows *higher absolute packet loss rates* in almost every audio scenario (see Table 8.28, Figures 8.8, 8.4), these losses are **rarely perceptible** to listeners. Each lost packet in Opus at 128 kbps typically represents a small, independently concealed audio frame (often only a few milliseconds), and the codec's built-in packet loss concealment ensures that short, isolated gaps do not produce audible artifacts. Consequently, even periods of statistically substantial loss often result in audio that sounds continuous and natural, with at most subtle, fleeting glitches.

- **Stream Mode**: Stream mode provides lower overall packet loss (often near zero), but its reliance on strict in-order delivery and buffer-based recovery introduces a different kind of artifact. When a retransmission or buffer underrun occurs, the audio may "hang" or briefly pause—creating a *noticeable and clearly audible* stall for the end user, even if the absolute number of missing frames is much lower. This is visible in Figures 8.3 and 8.5: outliers are rare, but can cause momentary playback interruption.

Video Transmission Clear differences emerge in video scenarios, especially under bandwidth constraints:

- **Datagram Mode**: In scenarios with reduced bandwidth (e.g., Figure 8.46), datagram mode frequently leads to unrecoverable playback freezes and visible frame loss. As demonstrated in Figures 8.44 and 8.42, latency spikes and out-of-order delivery amplify these artifacts. In many cases, playback fails to recover even after bandwidth is restored, due to unrecoverable application-layer loss.

- **Stream Mode**: Even during heavy congestion, stream mode typically replaces freezes with temporary frame skips or brief stuttering (Figures 8.41 and 8.43). After bandwidth constraints are lifted, stream mode rapidly resumes smooth playback, as confirmed by cumulative loss graphs (Figure 8.45).

Interpretative insight: In practice, a protocol with more measured packet loss (datagram) can yield *less perceptible* degradation in audio playback compared to a protocol with fewer lost packets but a tendency toward audible buffer underruns (stream). This is a consequence of modern audio codecs' resilience and the psychoacoustic masking of short gaps, as well as the abrupt nature of stream stalls.

Cumulative Evidence from Summary Tables and Graphs - Table 8.40 provides a scenario-by-scenario breakdown of packet loss types. The contrast is stark: **stream mode's "Not Received" packets almost always remain below 5%**, while **datagram mode ranges from 8% to over 41%**, depending on traffic volume and network impairment. - The stacked bar and line plots in Figures 8.51, 8.48, 8.49, and 8.50 visually summarize these differences, revealing a persistent and often widening gap in favor of stream mode, especially for "LOST" and "TOO SLOW" packets.

Latency, Jitter, and Timing Stability

Latency visualizations and distribution plots present further evidence of protocol-level differences:

- **Stream mode** maintains consistently low median latencies, with narrow, symmetric distributions and few outliers—even under constrained bandwidth (see Figures 8.19, 8.35, 8.43). - **Datagram mode** suffers from visible "long tail" effects and heavy jitter outliers in all but the least-stressed scenarios (see Figures 8.20, 8.36, 8.44), with latency spikes sometimes exceeding the playable window and causing late or dropped packets (see also the corresponding cumulative loss plots).

In audio scenarios, the practical impact is increased likelihood of audible dropouts for datagram mode; in video, the effect is even more severe, leading to frame drops and freezing.

Resource Utilization (CPU Overhead)

As observed in host resource metrics (not shown in summary tables but described in implementation sections), **CPU usage is generally comparable** between modes during normal operation. However, under heavy fragmentation or loss, datagram mode triggers additional CPU load due to application-level reassembly, sequencing, and error correction logic—effects that are avoided by stream mode's integrated, transport-level reliability.

Integrated Interpretation and Visual Synthesis

To make these findings visually accessible and evidence-based, the following figures should be used as direct references within this discussion:

- **Bandwidth Usage Comparison**: Figure 8.47 (total and instantaneous), Figures 8.1–8.40 (scenario-specific). - **Latency and Jitter Distributions**: Figures 8.19–8.44 (scenarios), 8.49, 8.50 (across all). - **Packet Loss Characteristics**: Table 8.40 (full scenario statistics), Figures 8.13–8.46 (cumulative per-scenario), Figure 8.51 (aggregate). - **Empirical Playback Observations**: As summarized in the “empirical observations” for each scenario, e.g., stream mode resumes playback rapidly (Figures 8.17, 8.41), while datagram mode frequently does not recover (Figures 8.42).

8.7.2 Discussion and Design Implications

The experimental evidence highlights clear operational trade-offs between the two QUIC transport modes:

- **QUIC Stream Mode** is the most robust, providing resilience to loss and maintaining stable playback for both audio and video under a range of conditions. Its built-in retransmission and ordering enable graceful recovery after congestion, minimizing frame and packet loss, with quick restoration after network constraints subside.
- **QUIC Datagram Mode**, although capable of lower latency in ideal networks, is vulnerable to loss and out-of-order delivery, leading to frame drops and unrecoverable failures—especially for video streams. Audio in datagram mode, however, illustrates that high measured loss rates can still result in continuous, perceptually robust playback, due to codec design and short packet duration.

Implications for application design: Datagram mode may be considered for delay-sensitive, loss-tolerant applications, or where the application implements advanced loss concealment, FEC, or server-side buffering. For high-quality, reliable media delivery (broadcast, conferencing, streaming), stream mode remains the preferred choice—especially when video is present or strict reliability is required. For audio-only workloads, application designers should carefully weigh perceptual outcomes: statistical packet loss in datagram mode is often inaudible, but stream mode’s rare stalling may have more noticeable impact.

In summary, QUIC stream mode consistently delivers superior reliability, lower effective packet loss, and stable media playback across real-world network conditions. Datagram mode can match or surpass it in perceived audio continuity for certain workloads, but only at the cost of higher statistical loss and a requirement for resilient codec or application design. For robust, high-quality user experience—especially with video—stream-based transport is strongly recommended unless ultra-low-latency requirements dictate otherwise.
As visually documented in Table 8.40 and the full set of figures 8.47–8.51, stream-based transmission ensures the best balance of reliability, predictability, and quality for modern real-time media delivery, especially in the face of variable or challenging network environments.

Chapter 9

Conclusions

This thesis has presented a rigorous, empirically grounded evaluation of Media over QUIC, leveraging a hybrid test environment with enhanced logging and analytics capabilities, by building and validating a hybrid MoQ testbed composed of a JavaScript client (moq-js) and a Rust relay (moq-rs), leveraging QUIC's partial-reliability (datagram) and full-reliability (stream) capabilities.

The testing infrastructure combines browser-based media streaming with a remote QUIC relay, offering insights into Media over QUIC performance across real-world network paths. The layered logging and analysis toolchain enables fine-grained evaluation of media delivery metrics, crucial for validating protocol behavior under different operational conditions.

By systematically varying media types, bitrates, and transport modes over a 1 Gbps fiber link, this work demonstrates the practical trade-offs between QUIC stream and datagram transport modes within the MoQ protocol context, particularly in low-latency real-time media applications.

While datagram offers architectural flexibility and possibly lower theoretical overhead, it requires robust custom logic at the application layer to handle ordering, loss, and reassembly.

Our experiments revealed that such custom implementations—although functional—introduce inefficiencies and greater playback instability, especially under bandwidth constraints. Stream mode, in contrast, benefits from built-in flow control and ordering mechanisms and leads to more stable playback and better user experience in most conditions.

The comparative analysis of stream and datagram transport modes under realistic network scenarios yields several key conclusions:

- **Reliability vs. Flexibility:** QUIC streams are inherently more reliable, preserving playback continuity and quality even in challenging conditions. Their robustness derives from built-in retransmission, ordering, and flow control, which shield end users from transient losses at the expense of latency.

- **Performance Under Stress:** QUIC datagrams offer compelling advantages in low-latency and low-overhead scenarios. However, in practice, their lack of built-in reliability exposes applications to significant playback risks; our results show that, without custom recovery mechanisms, datagram transport can result in catastrophic failure modes for video.
- **Media Type Sensitivity:** Audio, due to lower bitrate and higher tolerance to loss, is well-served by both transport modes, with only minor artifacts observed in challenging conditions. Video, conversely, demands the reliability guarantees offered by stream mode unless substantial investment is made in robust application-level recovery.

To support this analysis, we also designed and implemented:

- A custom **MoQ logger** for both JS and Rust-based relays, capable of tracking send/receive events, CPU load, and retransmission metadata.
- A suite of plotting tools:
  - `plotMoqjsTimestamp.py` — detailed timeline views of latency, jitter, and retransmissions.
  - `plotMoqjsDistribution.py` — distribution and KDE visualizations for multiple track types.
- Integration with WireShark logs and export tools for retransmission analysis.

These tools allowed a deep inspection of MoQ delivery behavior at runtime, enabling data-informed optimization of both publishing and receiving logic.

Future work could explore:

- Hybrid transport strategies combining stream and datagram modes adaptively.
- Extension of logger tooling to support live metrics export (e.g., Prometheus).
- Enhanced WARP segment encoding strategies to reduce bandwidth cost in object-level fragmentation.

In summary, while QUIC datagram mode holds promise for specific ultra-low-latency applications, QUIC stream mode currently provides the best overall user experience and service robustness in real-world media streaming. Although custom datagram support for MoQ in browsers is feasible and demonstrates key protocol flexibility, for most practical applications—especially under constrained network conditions—stream mode remains the preferred and more robust solution.

At the same time, these results affirm that the MoQ architecture, coupled with intelligent relay strategies, can flexibly address diverse real-time media requirements, while the analytical and experimental tools developed in this thesis provide a foundation for continued exploration and optimization of next-generation media transport protocols.

Bibliography

[1] Akamai Technologies. What is http? Akamai Glossary.

[2] Apple Inc. and Microsoft Corporation. Iso/iec 23000-19:2018 – common media application format (cmaf) for segmented media. ISO/IEC JTC1/SC29, 2018. Standard.

[3] M. Belshe, R. Peon, and M. Thomson. Hypertext transfer protocol version 2 (http/2). Technical Report RFC 7540, IETF, May 2015.

[4] M. Bishop. Hypertext transfer protocol version 3 (http/3). Technical Report RFC 9114, IETF, June 2022.

[5] Cloudflare. What is http/3? Cloudflare Learning Center, 2024.

[6] R. Even, P. Roesler, R. Pantos, and S. Nandakumar. Warp: A streaming format for media over quic. Internet-draft, IETF MoQ Working Group, 2024. draft-even-moq-warp-03.

[7] Google. Webtransport overview. Chrome for Developers, 2024.

[8] R. Hamilton et al. Webtransport over http/3. Internet-draft, IETF WebTransport Working Group, 2024. draft-ietf-webtrans-http3-12.

[9] P. Hesmans and O. Bonaventure. Http/3 performance over quic. In *Proceedings of the 2020 Workshop on the Evolution, Performance, and Interoperability of QUIC*. ACM, 2020.

[10] IETF Media over QUIC (MoQ) Working Group. Media over quic architecture. Internet-Draft, work in progress, 2024.

[11] IETF MoQ Working Group. Media over quic (moq). IETF Datatracker.

[12] IETF MoQ Working Group. Media over quic transport protocol. Internet-draft, IETF, 2024. Work in Progress.

[13] International Organization for Standardization. Information technology — dynamic adaptive streaming over http (dash) — part 1: Media presentation description and segment formats, 2022. ISO/IEC 23009-1:2022.

[14] J. Iyengar and M. Thomson. Quic: A udp-based multiplexed and secure transport. Technical Report RFC 9000, IETF, May 2021.

[15] J. Iyengar and M. Thomson. Quic: A udp-based multiplexed and secure transport. Technical Report RFC 9000, IETF, 2021.

[16] M. Kühlewind and B. Trammell. Manageability of the quic transport protocol. Technical Report RFC 9312, IETF, October 2022.

[17] G. Law et al. Warp: The web application real-time protocol streaming format. Internet-draft, IETF, June 2024. draft-law-moq-warpstreamingformat-03.

[18] J. Law. Warp streaming format. Internet-draft, IETF, 2023.

[19] C. Mueller, J. F. Paris, and D. Singer. Low-latency cmaf for http streaming. Apple Inc., 2020.

[20] R. Nandakumar, R. Even, K. Pugin, S. Gouaillard, and S. Corlay. Meta: Media transport over quic, 2022. MoQ IETF Presentations.

[21] S. Nandakumar, R. Even, P. Roesler, R. Pantos, and P. Gouaillard. Rush: Reliable upload streaming for media over quic. Internet-draft, IETF MoQ Working Group, 2024. draft-nandakumar-moq-rush-02.

[22] R. Pant, A. A. Seshadri, T. Kocher, et al. Low latency streaming at scale with ll-hls. Apple Worldwide Developers Conference (WWDC), 2020. Accessed: 2024-06-29.

[23] R. Pantos, P. Roesler, and K. Law. Media over quic: Design and implementation at scale. Twitch Engineering Blog, 2023.

[24] T. Pauly, E. Kinnear, and D. Schinazi. Using quic datagrams. Technical Report RFC 9221, IETF, March 2022.

[25] J. Postel. User datagram protocol. Technical Report RFC 768, IETF, August 1980.

[26] J. Postel. Transmission control protocol. Technical Report RFC 793, IETF, September 1981.

[27] K. Pugin. Rush ingest format. Internet-draft, IETF, 2023.

[28] P. Roesler, R. Even, and S. Nandakumar. Media over quic architecture. Internet-draft, IETF MoQ Working Group, 2024. draft-ietf-moq-arch-06.

[29] P. Roesler, R. Even, and S. Nandakumar. Media over quic transport. Internet-draft, IETF MoQ Working Group, 2024. draft-ietf-moq-transport-06.

[30] Upwork. What is http/2? how http/2 works & why it's important. Upwork Resources, 2024.

[31] WebCodecs Community Group. Webcodecs api. W3C Recommendation, 2024.

[32] WebTransport Community Group. Webtransport api. W3C Recommendation, 2024.