moq
Internet-Draft
Intended status: Informational
Expires: 16 May 2026

L. Curley

12 November 2025

Media over QUIC - Lite
draft-lcurley-moq-lite-02

Abstract

moq-lite is designed to fanout live content 1->N across the internet. It leverages QUIC to prioritize important content, avoiding head-of-line blocking while respecting encoding dependencies. While primarily designed for media, the transport is payload agnostic and can be proxied by relays/CDNs without knowledge of codecs, containers, or encryption keys.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Media Over QUIC Working Group mailing list (moq@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/moq/.

Source for this draft and an issue tracker can be found at https://github.com/kixelated/moq-drafts.

Status of This Memo

Table of Contents

1. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Rationale

This draft is based on MoqTransport [moqt]. The concepts, motivations, and terminology are very similar and when in doubt, refer to existing MoqTransport literature. A few things have been renamed (ex. object -> frame) to better align with media terminology.

I absolutely believe in the motivation and potential of Media over QUIC. The layering is phenomenal and addresses many of the problems with current live media protocols. I fully support the goals of the working group and the IETF process.

But it's been difficult to design such an experimental protocol via committee. MoqTransport has become too complicated.

There are too many messages, optional modes, and half-baked features. Too many hypotheses, too many potential use-cases, too many diametrically opposed opinions. This is expected (and even desired) as compromise gives birth to a standard.

But I believe the standardization process is hindering practical experimentation. The ideas behind MoQ can be proven now before being cemented as an RFC. We should spend more time building an _actual_ application and less time arguing about a hypothetical one.

moq-lite is the bare minimum needed for a real-time application aiming to replace WebRTC. Every feature from MoqTransport that is not necessary (or has not been implemented yet) has been removed for simplicity. This includes many great ideas (ex. group order) that may be added as they are needed. This draft is the current state, not the end state.

3. Concepts

moq-lite consists of:

* *Session*: An established QUIC connection between a client and server.

* *Broadcast*: A collection of Tracks from a single publisher.

* *Track*: An series of Groups, each of which can be delivered and decoded _out-of-order_.

* *Group*: An series of Frames, each of which must be delivered and decoded _in-order_.

* *Frame*: A sized payload of bytes representing a single moment in time.

The application determines how to split data into broadcast, tracks, groups, and frames. The moq-lite layer provides fanout, prioritization, and caching even for latency sensitive applications.

3.1. Session

A Session consists of a connection between a client and a server. There is currently no P2P support within QUIC so it’s out of scope for moq-lite.

A session is established after the necessary QUIC, WebTransport, and moq-lite handshakes have completed. The moq-lite handshake is simple and consists of version and extension negotiation.

While moq-lite is a point-to-point protocol, it’s intended to work end-to-end via relays. Each client establishes a session with a CDN edge server, ideally the closest one. Any broadcasts and subscriptions are transparently proxied by the CDN behind the scenes.

3.2. Broadcast

A Broadcast is a collection of Tracks from a single publisher. This corresponds to a MoqTransport’s "track namespace".

A publisher may produce multiple broadcasts, each of which is advertised via an ANNOUNCE message. The subscriber uses the ANNOUNCE_PLEASE message to discover available broadcasts. These announcements are live and can change over time, allowing for dynamic origin discovery.

A broadcast consists of any number of Tracks. The contents and relationship between these tracks is determined by the application or via an out-of-band mechanism.

3.3. Track

A Track is a series of Groups identified by a unique name within a Broadcast.

A track consists of a single active Group at any moment, called the "latest group". When a new Group is started, the previous Group is closed and may be dropped for any reason. The duration before an incomplete group is dropped is determined by the application and the publisher/subscriber’s latency target.

Every subscription is scoped to a single Track. A subscription will always start at the latest Group and continues until either the publisher or subscriber cancels the subscription.

A subscriber chooses the priority of each subscription, hinting to the publisher which Track should arrive first during congestion. This enables the most important content to arrive during network degradation while still respecting encoding dependencies.

3.4. Group

A Group is an ordered stream of Frames within a Track.

Each group consists of an append-only list of Frames. A Group is served by a dedicated QUIC stream which is closed on completion, reset by the publisher, or cancelled by the subscriber. This ensures that all Frames within a Group arrive reliably and in order.

In contrast, Groups may arrive out of order due to network congestion and prioritization. The application should be prepared to handle this with a jitter buffer at the group level.

3.5. Frame

A Frame is a payload of bytes within a Group.

A frame is used to represent a chunk of data with a known size. A frame should represent a single moment in time and avoid any buffering that would increase latency.

There’s no timestamp or metadata associated with a Frame, this is the responsibility of the application.

4. Flow

This section outlines the flow of messages within a moq-lite session. See the section for Messages section for the specific encoding.

4.1. Connection

moq-lite runs on top of WebTransport. WebTransport is a layer on top of QUIC and HTTP/3, required for web support. The API is nearly identical to QUIC with the exception of stream IDs.

How the WebTransport connection is authenticated is out-of-scope for this draft.

4.2. Termination

QUIC bidirectional streams have an independent send and receive direction. Rather than deal with half-open states, moq-lite combines both sides. If an endpoint closes the send direction of a stream, the peer MUST also close their send direction.

moq-lite contains many long-lived transactions, such as subscriptions and announcements. These are terminated when the underlying QUIC stream is terminated.

To terminate a stream, an endpoint may: - close the send direction (STREAM with FIN) to gracefully terminate (all messages are flushed). - reset the send direction (RESET_STREAM) to immediately terminate.

After resetting the send direction, an endpoint MAY close the recv direction (STOP_SENDING). However, it is ultimately the other peer’s responsibility to close their send direction.

4.3. Handshake

After a connection is established, the client opens a Session Stream and sends a SESSION_CLIENT message, to which the server replies with a SESSION_SERVER message. The session is active until either endpoint closes or resets the Session Stream.

This session handshake is used to negotiate the moq-lite version and any extensions. See the Extension section for more information.

5. Streams

moq-lite uses a bidirectional stream for each transaction. If the stream is closed, potentially with an error, the transaction is terminated.

5.1. Bidirectional Streams

Bidirectional streams are used for control streams. There's a 1-byte STREAM_TYPE at the beginning of each stream.

ID	Stream	Creator
0x0	Session	Client
0x1	Announce	Subscriber
0x2	Subscribe	Subscriber
0x20	SessionCompat	Client

Table 1

5.1.1. Session

The Session stream is used to establish the moq-lite session, negotiating the version and any extensions. This stream remains open for the duration of the session and its closure indicates the session is closed.

The client MUST open the Session Stream, write the Session Stream ID (0x0), and write a SESSION_CLIENT message. If the server does not support any of the client's versions, it MUST close the stream with an error code and MAY close the connection. Otherwise, the server replies with a SESSION_SERVER message to complete the handshake.

Afterwards, both endpoints MAY send SESSION_UPDATE messages. This is currently used to notify the other endpoint of a significant change in the session bitrate.

This draft's version is combined with the constant 0xff0dad00. For example, moq-lite-draft-04 is identified as 0xff0dad04.

5.1.2. SessionCompat

The SessionCompat stream exists to support moq-transport draft 11-14. This will be removed in a future version as moq-transport draft 15 uses ALPN instead.

The client writes a CLIENT_SETUP message on the SessionCompat stream and receives a SERVER_SETUP message in response.

Consult the MoqTransport ([moqt]) draft for more information about the encoding. Notably, each message contains a u16 length prefix instead of a VarInt (moq-lite).

If a moq-lite version is negotiated, this stream becomes a normal Session stream. If a moq-transport version is negotiated, this stream becomes the MoqTransport control stream.

5.1.3. Announce

A subscriber can open a Announce Stream to discover broadcasts matching a prefix.

The subscriber creates the stream with a ANNOUNCE_PLEASE message. The publisher replies with an ANNOUNCE_INIT message containing all currently active broadcasts that currently match the prefix, followed by ANNOUNCE messages for any changes.

The ANNOUNCE_INIT message contains an array of all currently active broadcast paths encoded as a suffix. Each path in ANNOUNCE_INIT can be treated as if it were an ANNOUNCE message with status active.

After ANNOUNCE_INIT, the publisher sends ANNOUNCE messages for any changes also encoded as a suffix. Each ANNOUNCE message contains one of the following statuses:

* active: a matching broadcast is available.

* ended: a previously active broadcast is no longer available.

Each broadcast starts as ended (unless included in ANNOUNCE_INIT) and MUST alternate between active and ended. The subscriber MUST reset the stream if it receives a duplicate status, such as two active statuses in a row or an ended without active. When the stream is closed, the subscriber MUST assume that all broadcasts are now ended.

Path prefix matching and equality is done on a byte-by-byte basis. There MAY be multiple Announce Streams, potentially containing overlapping prefixes, that get their own ANNOUNCE_INIT and ANNOUNCE messages.

5.1.4. Subscribe

A subscriber opens Subscribe Streams to request a Track.

The subscriber MUST start a Subscribe Stream with a SUBSCRIBE message followed by any number of SUBSCRIBE_UPDATE messages. The publisher MUST reply with an SUBSCRIBE_OK message.

The publisher SHOULD close the stream after the track has ended. Either endpoint MAY reset/cancel the stream at any time.

5.2. Unidirectional Streams

Unidirectional streams are used for data transmission.

ID	Stream	Creator
0x0	Group	Publisher

Table 2

5.2.1. Group

A publisher creates Group Streams in response to a Subscribe Stream.

A Group Stream MUST start with a GROUP message and MAY be followed by any number of FRAME messages. A Group MAY contain zero FRAME messages, potentially indicating a gap in the track. A frame MAY contain an empty payload, potentially indicating a gap in the group.

Both the publisher and subscriber MAY reset the stream at any time. This is not a fatal error and the session remains active. The subscriber MAY cache the error and potentially retry later.

6. Encoding

This section covers the encoding of each message.

6.1. Message Length

Most messages are prefixed with a variable-length integer indicating the number of bytes in the message payload that follows. This length field does not include the length of the varint length itself.

An implementation SHOULD close the connection with a PROTOCOL_VIOLATION if it receives a message with an unexpected length. The version and extensions should be used to support new fields, not the message length.

6.2. STREAM_TYPE

All streams start with a short header indicating the stream type.

```
STREAM_TYPE {
  Stream Type (i)
}
```

The stream ID depends on if it’s a bidirectional or unidirectional stream, as indicated in the Streams section. A receiver MUST close the session if it receives an unknown stream type.

6.3. SESSION_CLIENT

The client initiates the session by sending a SESSION_CLIENT message.

```
SESSION_CLIENT Message {
  Message Length (i)
  Supported Versions Count (i)
  Supported Version (i)
  Extension Count (i)
  [
    Extension ID (i)
    Extension Payload (b)
  ]...
}
```

6.4. SESSION_SERVER

The server responds with the selected version and any extensions.

```
SESSION_SERVER Message {
  Message Length (i)
  Selected Version (i)
  Extension Count (i)
  [
    Extension ID (i)
    Extension Payload (b)
  ]...
}
```

6.5. SESSION_UPDATE

```
SESSION_UPDATE Message {
  Message Length (i)
  Session Bitrate (i)
}
```

Session Bitrate: The estimated bitrate of the QUIC connection in bits per second. This SHOULD be sourced directly from the QUIC congestion controller. A value of 0 indicates that this information is not available.

6.6. ANNOUNCE_PLEASE

A subscriber sends an ANNOUNCE_PLEASE message to indicate it wants to receive an ANNOUNCE message for any broadcasts with a path that starts with the requested prefix.

```
ANNOUNCE_PLEASE Message {
  Message Length (i)
  Broadcast Path Prefix (s),
}
```

Broadcast Path Prefix: Indicate interest for any broadcasts with a path that starts with this prefix.

The publisher MUST respond with an ANNOUNCE_INIT message containing any matching and active broadcasts, followed by ANNOUNCE messages for any updates. Implementations SHOULD consider reasonable limits on the number of matching broadcasts to prevent resource exhaustion.

6.7. ANNOUNCE_INIT

A publisher sends an ANNOUNCE_INIT message immediately after receiving an ANNOUNCE_PLEASE to communicate all currently active broadcasts that match the requested prefix. Only the suffixes are encoded on the wire, as the full path can be constructed by prepending the requested prefix.

This message is useful to avoid race conditions, as ANNOUNCE_INIT does not trickle in like ANNOUNCE messages. For example, an API server that wants to list the current participants could issue an ANNOUNCE_PLEASE and immediately return the ANNOUNCE_INIT response. Without ANNOUNCE_INIT, the API server would have use a timer to wait until ANNOUNCE to guess when all ANNOUNCE messages have been received.

```
ANNOUNCE_INIT Message {
  Message Length (i)
  Suffix Count (i),
  [
    Broadcast Path Suffix (s),
  ]...
}
```

Suffix Count: The number of active broadcast path suffixes that follow. This can be 0. A publisher MUST NOT include duplicate suffixes in a single ANNOUNCE_INIT message.

Broadcast Path Suffix: Each suffix is combined with the broadcast path prefix from ANNOUNCE_PLEASE to form the full broadcast path. This includes all currently active broadcasts matching the prefix.

6.8. ANNOUNCE

A publisher sends an ANNOUNCE message to advertise a change in broadcast availability. Only the suffix is encoded on the wire, as the full path can be constructed by prepending the requested prefix.

The status is relative to the ANNOUNCE_INIT and all prior ANNOUNCE messages combined. A client MUST ONLY alternate between status values (from active to ended or vice versa).

```
ANNOUNCE Message {
  Message Length (i)
  Announce Status (i),
  Broadcast Path Suffix (s),
}
```

Announce Status: A flag indicating the announce status.

* ended (0): A path is no longer available.

* active (1): A path is now available.

Broadcast Path Suffix: This is combined with the broadcast path prefix to form the full broadcast path.

6.9. SUBSCRIBE

SUBSCRIBE is sent by a subscriber to start a subscription.

```
SUBSCRIBE Message {
  Message Length (i)
  Subscribe ID (i)
  Broadcast Path (s)
  Track Name (s)
  Subscriber Priority (i)
}
```

Subscribe ID: A unique identifier chosen by the subscriber. A Subscribe ID MUST NOT be reused within the same session, even if the prior subscription has been closed.

Subscriber Priority: The transmission priority of the subscription relative to all other active subscriptions within the session. The publisher SHOULD transmit _higher_ values first during congestion.

6.10. SUBSCRIBE_UPDATE

A subscriber can modify a subscription with a SUBSCRIBE_UPDATE message.

```
SUBSCRIBE_UPDATE Message {
  Message Length (i)
  Subscriber Priority (i)
}
```

Subscriber Priority: The new subscriber priority; see SUBSCRIBE.

6.11. SUBSCRIBE_OK

The SUBSCRIBE_OK is sent in response to a SUBSCRIBE.

```
SUBSCRIBE_OK Message {
  Message Length = 0
}
```

That’s right, it’s an empty message at the moment.

6.12. GROUP

The GROUP message contains information about a Group, as well as a reference to the subscription being served.

```
GROUP Message {
  Message Length (i)
  Subscribe ID (i)
  Group Sequence (i)
}
```

Subscribe ID: The corresponding Subscribe ID. This ID is used to distinguish between multiple subscriptions for the same track.

Group Sequence: The sequence number of the group. This SHOULD increase by 1 for each new group.

6.13. FRAME

The FRAME message is a payload at a specific point of time.

```
FRAME Message {
  Message Length (i)
  Payload (b)
}
```

Payload: An application specific payload. A generic library or relay MUST NOT inspect or modify the contents unless otherwise negotiated.

7. Appendix A: Changelog

7.1. moq-lite-02

* Added SessionCompat stream.

* Editorial stuff.

7.2. moq-lite-01

* Added ANNOUNCE_INIT.

* Added Message Length (i) to all messages.

8. Appendix B: Upstream Differences

A quick comparison of moq-lite and moq-transport-14:

* Streams instead of request IDs.

* Pull only: No unsolicited publishing.

* Uses HTTP for VOD instead of FETCH.

* Extensions instead of parameters.
* Names use utf-8 strings instead of byte arrays.
* Track Namespace is a string, not an array of any array of bytes.
* Subscriptions start at the latest group, not the latest object.
* No subgroups
* No group/object ID gaps
* No object properties
* No datagrams
* No paused subscriptions (forward=0)

8.1. Deleted Messages

* GOAWAY
* MAX_SUBSCRIBE_ID
* REQUESTS_BLOCKED
* SUBSCRIBE_ERROR
* UNSUBSCRIBE
* PUBLISH_DONE
* PUBLISH
* PUBLISH_OK
* PUBLISH_ERROR
* FETCH
* FETCH_OK
* FETCH_ERROR
* FETCH_CANCEL
* FETCH_HEADER

* TRACK_STATUS
* TRACK_STATUS_OK
* TRACK_STATUS_ERROR
* PUBLISH_NAMESPACE
* PUBLISH_NAMESPACE_OK
* PUBLISH_NAMESPACE_ERROR
* PUBLISH_NAMESPACE_CANCEL
* SUBSCRIBE_NAMESPACE_OK
* SUBSCRIBE_NAMESPACE_ERROR
* UNSUBSCRIBE_NAMESPACE
* OBJECT_DATAGRAM

8.2. Renamed Messages

* SUBSCRIBE_NAMESPACE -> ANNOUNCE_PLEASE
* SUBGROUP_HEADER -> GROUP

8.3. Deleted Fields

Some of these fields occur in multiple messages.

* Request ID
* Track Alias
* Group Order
* Filter Type
* StartGroup
* StartObject
* EndGroup
* Expires

* ContentExists
* Largest Group ID
* Largest Object ID
* Parameters
* Subgroup ID
* Object ID
* Object Status
* Extension Headers

9. Security Considerations

TODO Security

10. IANA Considerations

This document has no IANA actions.

11. Normative References

[moqt] Nandakumar, S., Vasiliev, V., Swett, I., and A. Frindell, "Media over QUIC Transport", Work in Progress, Internet-Draft, draft-ietf-moq-transport-15, 20 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-moq-transport-15>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

Acknowledgments

TODO acknowledge.

Author’s Address

Luke Curley

Email: kixelated@gmail.com