moq
Internet-Draft
Intended status: Informational
Expires: 18 September 2026

L. Curley
17 March 2026

Media over QUIC - Lite
draft-lcurley-moq-lite-03

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 3
2. Rationale 3
3. Concepts 4
3.1. Session 4
3.2. Broadcast 5
3.3. Track 5
3.4. Group 6
3.5. Frame 6
4. Flow 6
4.1. Connection 6
4.2. Termination 6
4.3. Handshake 7
5. Streams 7
5.1. Bidirectional Streams 7
5.1.1. Announce 7
5.1.2. Subscribe 8
5.1.3. Fetch 8
5.1.4. Probe 9
6. Delivery 9
6.1. Prioritization 9
6.1.1. Priority 9
6.1.2. Ordered 10
6.2. Expiration 11
6.3. Unidirectional Streams 11
6.3.1. Group 11
7. Encoding 12
7.1. Message Length 12
7.2. STREAM_TYPE 12
7.3. ANNOUNCE_PLEASE 12
7.4. ANNOUNCE 13
7.5. SUBSCRIBE 13
7.6. SUBSCRIBE_UPDATE 14

7.7. SUBSCRIBE_OK 15
7.8. SUBSCRIBE_DROP 15
7.9. FETCH 16
7.10. PROBE 16
7.11. GROUP 17
7.12. FRAME 17
8. Appendix A: Changelog 17
8.1. moq-lite-03 17
8.2. moq-lite-02 18
8.3. moq-lite-01 18
9. Appendix B: Upstream Differences 18
9.1. Deleted Messages 19
9.2. Renamed Messages 20
9.3. Deleted Fields 20
10. Security Considerations 21
11. IANA Considerations 21
12. Normative References 21
Acknowledgments 21
Author’s Address 21

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 within a Group.

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.

The moq-lite version identifier is moq-lite-xx where xx is the two-digit draft version. For bare QUIC, this is negotiated as an ALPN token during the QUIC handshake. For WebTransport over HTTP/3, the QUIC ALPN remains h3 and the moq-lite version is advertised via the WT-Available-Protocols and WT-Protocol CONNECT headers.

The session is active immediately after the QUIC/WebTransport connection is established. Extensions are negotiated via stream probing: an endpoint opens a stream with an unknown type and the peer resets it if unsupported.

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, relationships, and encoding of tracks are determined by the application.

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 starts at a configurable Group (defaulting to the latest) and continues until a configurable end Group or until either the publisher or subscriber cancels the subscription.

The subscriber and publisher both indicate their delivery preference: - Priority indicates if Track A should be transmitted instead of Track B. - Ordered indicates if the Groups within a Track should be transmitted in order. - Max Latency indicates the maximum duration before a Group is abandoned.

The combination of these preferences 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 process or buffer groups out of order to avoid blocking on flow control.

3.5. Frame

A Frame is a payload of bytes within a Group.

A frame is used to represent a chunk of data with an upfront size. The contents are opaque to the moq-lite layer.

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

See the Section 3.1 section for ALPN negotiation and session activation details.

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
0x1	Announce	Subscriber
0x2	Subscribe	Subscriber
0x3	Fetch	Subscriber
0x4	Probe	Subscriber

Table 1

5.1.1. 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 ANNOUNCE messages for any matching broadcasts and any future changes. 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 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 messages.

5.1.2. 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 replies with a SUBSCRIBE_OK message followed by any number of SUBSCRIBE_DROP and additional SUBSCRIBE_OK messages. The first message on the response stream MUST be a SUBSCRIBE_OK; it is not valid to send a SUBSCRIBE_DROP before SUBSCRIBE_OK.

The publisher closes the stream (FIN) when every group from start to end has been accounted for, either via a GROUP stream (completed or reset) or a SUBSCRIBE_DROP message. Unbounded subscriptions (no end group) stay open until the publisher closes the stream to indicate the track has ended, or either endpoint resets. Either endpoint MAY reset/cancel the stream at any time.

5.1.3. Fetch

A subscriber opens a Fetch Stream (0x3) to request a single Group from a Track.

The subscriber sends a FETCH message containing the broadcast path, track name, priority, and group sequence. Unlike Group Streams (which MUST start with a GROUP message), the publisher responds with FRAME messages directly on the same bidirectional stream — there is no preceding GROUP header. The Subscribe ID and Group Sequence for the returned FRAME messages are implicit, taken from the original FETCH request. The publisher FINs the stream after the last frame, or resets the stream on error.

Fetch behaves like HTTP: a single request/response per stream.

5.1.4. Probe

A subscriber opens a Probe Stream (0x4) to measure the available bitrate of the connection.

The subscriber sends a PROBE message with a target bitrate on the bidirectional stream. The subscriber MAY send additional PROBE messages on the same stream to update the target bitrate; the publisher MUST treat each PROBE as a new target to attempt. The publisher SHOULD pad the connection to achieve the most recent target bitrate. The publisher periodically replies with PROBE messages on the same bidirectional stream containing the current measured bitrate.

If the publisher does not support PROBE (e.g., congestion controller is not exposed), it MUST reset the stream.

6. Delivery

The most important concept in moq-lite is how to deliver a subscription. QUIC can only improve the user experience if data is delivered out-of-order during congestion. This is the sole reason why data is divided into Broadcasts, Tracks, Groups, and Frames.

moq-lite consists of multiple groups being transmitted in parallel across separate streams. How these streams get transmitted over the network is very important, and yet has been distilled down into a few simple properties:

6.1. Prioritization

The Publisher and Subscriber both exchange Priority and Ordered values: - Priority determines which Track should be transmitted next. - Ordered determines which Group within the Track should be transmitted next.

A publisher SHOULD attempt to transmit streams based on these fields. This depends on the QUIC implementation and it may not be possible to get fine-grained control.

6.1.1. Priority

The Subscriber Priority is scoped to the connection. The Publisher Priority SHOULD be used to resolve conflicts or ties.

A conflict can occur when a relay tries to serve multiple downstream subscriptions from a single upstream subscription. Any upstream subscription SHOULD use the publisher priority, not some combination of different subscriber priorities.

Rather than try to explain everything, here's an example:

Example: There are two people in a conference call, Ali and Bob.

We subscribe to both of their audio tracks with priority 2 and video tracks with priority 1. This will cause equal priority for Ali and Bob while prioritizing audio. ali/audio + bob/audio: subscriber_priority=2 publisher_priority=2 ali/video + bob/video: subscriber_priority=1 publisher_priority=1

If Bob starts actively speaking, they can bump their publisher priority via a SUBSCRIBE_OK message. This would cause tracks be delivered in this order: bob/audio: subscriber_priority=2 publisher_priority=3 ali/audio: subscriber_priority=2 publisher_priority=2 bob/video: subscriber_priority=1 publisher_priority=2 ali/video: subscriber_priority=1 publisher_priority=1

The subscriber priority takes precedence, so we could override it if we decided to full-screen Ali's window: ali/audio subscriber_priority=4 publisher_priority=2 ali/video subscriber_priority=3 publisher_priority=1 bob/audio subscriber_priority=2 publisher_priority=3 bob/video subscriber_priority=1 publisher_priority=2

6.1.2. Ordered

The Subscriber Ordered field signals if older (0x1) or newer (0x0) groups should be transmitted first within a Track. The Publisher Ordered field MAY likewise be used to resolve conflicts.

An application SHOULD use ordered when it wants to provide a VOD-like experience, preferring to buffer old groups rather than skip them. An application SHOULD NOT use ordered when it wants to provide a live experience, preferring to skip old groups rather than buffer them.

Note that Section 6.2 is not affected by ordered. An old group may still be cancelled/skipped if it exceeds max_latency set by either peer. An application MUST support gaps and out-of-order delivery even when ordered is true.

6.2. Expiration

The Publisher and Subscriber both transmit a Max Latency value, indicating the maximum duration before a group is expired.

It is not crucial to aggressively expire groups thanks to Section 6.1. However, a lower priority group will still consume RAM, bandwidth, and potentially flow control. It is RECOMMENDED that an application set conservative limits and only resort to expiration when data is absolutely no longer needed.

A subscriber SHOULD expire groups based on the Subscriber Max Latency in SUBSCRIBE/SUBSCRIBE_UPDATE. A publisher SHOULD expire groups based on the Publisher Max Latency in SUBSCRIBE_OK. An implementation MAY use the minimum of both when determining when to expire a group.

Group age is computed relative to the latest group by sequence number. A group is never expired until at least the next group (by sequence number) has been received or queued. Once a newer group exists, a group is considered expired if the time between its arrival and the latest group’s arrival exceeds Max Latency. The arrival time is when the first byte of a group is received (subscriber) or queued (publisher). An expired group SHOULD BE reset at the QUIC level to avoid consuming flow control.

6.3. Unidirectional Streams

Unidirectional streams are used for data transmission.

ID	Stream	Creator
0x0	Group	Publisher

Table 2

6.3.1. 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.

7. Encoding

This section covers the encoding of each message.

7.1. 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.

7.2. 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 reset the stream if it receives an unknown stream type. Unknown stream types MUST NOT be treated as fatal; this enables extension negotiation via stream probing.

7.3. 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 ANNOUNCE messages for any matching and active broadcasts, followed by ANNOUNCE messages for any future updates. Implementations SHOULD consider reasonable limits on the number of matching broadcasts to prevent resource exhaustion.

7.4. 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 all prior ANNOUNCE messages on the same stream. A publisher 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),
  Hops (i),
}
```

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.

Hops: The number of hops from the origin publisher. This is used as a tiebreaker when there are multiple paths to the same broadcast. A relay SHOULD increment this value when forwarding an announcement.

7.5. 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 (8)
  Subscriber Ordered (8)
  Subscriber Max Latency (i)
  Start Group (i)
  End Group (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 priority of the subscription within the session, represented as a u8. The publisher SHOULD transmit _higher_ values first during congestion. See the Section 6.1 section for more information.

Subscriber Ordered: A single byte representing whether groups are transmitted in ascending (0x1) or descending (0x0) order. The publisher SHOULD transmit _older_ groups first during congestion if true. See the Section 6.1 section for more information.

Subscriber Max Latency: This value is encoded in milliseconds and represents the maximum age of a group relative to the latest group. The publisher SHOULD reset old group streams when the difference in arrival time between the group and the latest group exceeds this duration. See the Section 6.2 section for more information.

Start Group: The first group to deliver. A value of 0 means the latest group (default). A non-zero value is the absolute group sequence + 1.

End Group: The last group to deliver (inclusive). A value of 0 means unbounded (default). A non-zero value is the absolute group sequence + 1.

7.6. SUBSCRIBE_UPDATE

A subscriber can modify a subscription with a SUBSCRIBE_UPDATE message. A subscriber MAY send multiple SUBSCRIBE_UPDATE messages to update the subscription. The start and end group can be changed in either direction (growing or shrinking).

```
SUBSCRIBE_UPDATE Message {
  Message Length (i)
  Subscriber Priority (8)
  Subscriber Ordered (8)
  Subscriber Max Latency (i)
  Start Group (i)
  End Group (i)
}
```

See Section 5.1.2 for information about each field.

7.7. SUBSCRIBE_OK

A SUBSCRIBE_OK message is sent in response to a SUBSCRIBE. The publisher MAY send multiple SUBSCRIBE_OK messages to update the subscription. The first message on the response stream MUST be a SUBSCRIBE_OK; a SUBSCRIBE_DROP MUST NOT precede it.

```
SUBSCRIBE_OK Message {
  Type (i) = 0x0
  Message Length (i)
  Publisher Priority (8)
  Publisher Ordered (8)
  Publisher Max Latency (i)
  Start Group (i)
  End Group (i)
}
```

Type: Set to 0x0 to indicate a SUBSCRIBE_OK message.

Start Group: The resolved absolute start group sequence. A value of 0 means the start group is not yet known; the publisher MUST send a subsequent SUBSCRIBE_OK with a resolved value. A non-zero value is the absolute group sequence + 1.

End Group: The resolved absolute end group sequence (inclusive). A value of 0 means unbounded. A non-zero value is the absolute group sequence + 1.

See Section 5.1.2 for information about the other fields.

7.8. SUBSCRIBE_DROP

A SUBSCRIBE_DROP message is sent by the publisher on the Subscribe Stream when groups cannot be served.

```
SUBSCRIBE_DROP Message {
  Type (i) = 0x1
  Message Length (i)
  Start Group (i)
  End Group (i)
  Error Code (i)
}
```

Type: Set to 0x1 to indicate a SUBSCRIBE_DROP message.

Start Group: The first absolute group sequence in the dropped range.

End Group: The last absolute group sequence in the dropped range (inclusive).

Error Code: An application-specific error code. A value of 0 indicates no error; the groups are simply unavailable.

7.9. FETCH

FETCH is sent by a subscriber to request a single group from a track.

```
FETCH Message {
  Message Length (i)
  Broadcast Path (s)
  Track Name (s)
  Subscriber Priority (8)
  Group Sequence (i)
}
```

Broadcast Path: The broadcast path of the track to fetch from.

Track Name: The name of the track to fetch from.

Subscriber Priority: The priority of the fetch within the session, represented as a u8. See the Section 6.1 section for more information.

Group Sequence: The sequence number of the group to fetch.

The publisher responds with FRAME messages on the same stream. The publisher FINs the stream after the last frame, or resets on error.

7.10. PROBE

PROBE is used to measure the available bitrate of the connection.

```
PROBE Message {
  Message Length (i)
  Bitrate (i)
}
```

Bitrate: When sent by the subscriber (stream opener): the target bitrate in bits per second that the publisher should pad up to. When sent by the publisher (responder): the current measured bitrate in bits per second.

7.11. 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. A subscriber MUST handle gaps, potentially caused by congestion.

7.12. FRAME

The FRAME message is a payload within a group.

```
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.

8. Appendix A: Changelog

8.1. moq-lite-03

* Version negotiated via ALPN (moq-lite-xx) instead of SETUP messages.

* Removed Session, SessionCompat streams and SESSION_CLIENT/SESSION_SERVER/SESSION_UPDATE messages.

* Unknown stream types reset instead of fatal; enables extension negotiation via stream probing.

* Added FETCH stream for single group download.

* Added Start Group and End Group to SUBSCRIBE, SUBSCRIBE_UPDATE, and SUBSCRIBE_OK.

* Added SUBSCRIBE_DROP on Subscribe stream.

* Subscribe stream closed (FIN) when all groups accounted for.

* Added PROBE stream replacing SESSION_UPDATE bitrate.

* Removed ANNOUNCE_INIT message.

* Added Hops to ANNOUNCE.

* Added Subscriber Max Latency and Subscriber Ordered to SUBSCRIBE and SUBSCRIBE_UPDATE.

* Added Publisher Priority, Publisher Max Latency, and Publisher Ordered to SUBSCRIBE_OK.

* SUBSCRIBE_OK may be sent multiple times.

8.2. moq-lite-02

* Added SessionCompat stream.

* Editorial stuff.

8.3. moq-lite-01

* Added Message Length (i) to all messages.

9. Appendix B: Upstream Differences

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

* Streams instead of request IDs.

* Pull only: No unsolicited publishing.

* FETCH is HTTP-like (single request/response) vs MoqTransport FETCH (multiple groups).
* Extensions negotiated via stream probing instead of parameters.
* Both moq-lite and MoqTransport use ALPN for version identification.
* Names use utf-8 strings instead of byte arrays.
* Track Namespace is a string, not an array of any array of bytes.
* Subscriptions default to the latest group, not the latest object.
* No subgroups
* No group/object ID gaps
* No object properties
* No datagrams
* No paused subscriptions (forward=0)

9.1. Deleted Messages

* GOAWAY
* MAX_SUBSCRIBE_ID
* REQUESTS_BLOCKED
* SUBSCRIBE_ERROR
* UNSUBSCRIBE
* PUBLISH_DONE
* PUBLISH
* PUBLISH_OK
* PUBLISH_ERROR
* 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

9.2. Renamed Messages

* SUBSCRIBE_NAMESPACE -> ANNOUNCE_PLEASE
* SUBGROUP_HEADER -> GROUP

9.3. Deleted Fields

Some of these fields occur in multiple messages.

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

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

10. Security Considerations

TODO Security

11. IANA Considerations

This document has no IANA actions.

12. 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-17, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-moq-transport-17>.

[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