Workgroup: Independent Submission
Internet-Draft: draft-lcurley-moq-transport-00
Published: 26 May 2023
Intended Status: Informational
Expires: 27 November 2023
Authors: L. Curley (Twitch) K. Pugin (Meta) S. Nandakumar (Cisco) V. Vasiliev (Google)

Media over QUIC Transport

Abstract

This document defines the core behavior for Media over QUIC Transport (MOQT), a media transport protocol over QUIC. MOQT allows a producer of media to publish data and have it consumed via subscription by a multiplicity of endpoints. It supports intermediate content distribution networks and is designed for high scale and low latency distribution.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 27 November 2023.

Table of Contents

1. Introduction

Media Over QUIC Transport (MOQT) is a transport protocol that utilizes the QUIC network protocol [QUIC], either directly or via WebTransport [WebTransport], for the dissemination of media. MOQT utilizes a publish/subscribe workflow in which producers of media publish data in response to subscription requests from a multiplicity of endpoints. MOQT supports wide range of use-cases with different resiliency and latency (live, interactive) needs without compromising the scalability and cost effectiveness associated with content delivery networks.

MOQT is a generic protocol is designed to work in concert with multiple MoQ Streaming Formats. These MoQ Streaming Formats define how content is encoded, packaged, and mapped to MOQT objects, along with policies for discovery and subscription.

* Section 2 describes the object model employed by MOQT.

* Section 3 covers aspects of setting up a MOQT session.

* Section 4 covers protocol considerations on prioritization schemes and congestion response overall.

* Section 5 covers behavior at the relay entities.

* Section 6 covers how messages are encoded on the wire.

1.1. Motivation

The development of MOQT is driven by goals in a number of areas - specifically latency, the robustness of QUIC, workflow efficiency and relay support.

1.1.1. Latency

HTTP Adaptive Streaming (HAS) has been successful at achieving scale although often at the cost of latency. Latency is necessary to correct for variable network throughput. Ideally live content is consumed at the same bitrate it is produced. End-to-end latency would be fixed and only subject to encoding and transmission delays. Unfortunately, networks have variable throughput, primarily due to congestion. Attempting to deliver content encoded at a higher bitrate than the network can support causes queuing along the path

from producer to consumer. The speed at which a protocol can detect and respond to queuing determines the overall latency. TCP-based protocols are simple but are slow to detect congestion and suffer from head-of-line blocking. UDP-based protocols can avoid queuing, but the application is now responsible for the complexity of fragmentation, congestion control, retransmissions, receiver feedback, reassembly, and more. One goal of MOQT is to achieve the best of both these worlds: leverage the features of QUIC to create a simple yet flexible low latency protocol that can rapidly detect and respond to congestion.

1.1.2. Leveraging QUIC

The parallel nature of QUIC streams can provide improvements in the face of loss. A goal of MOQT is to design a streaming protocol to leverage the transmission benefits afforded by parallel QUIC streams as well exercising options for flexible loss recovery. Applying [QUIC] to HAS via HTTP/3 has not yet yielded generalized improvements in throughput. One reason for this is that sending segments down a single QUIC stream still allows head-of-line blocking to occur.

1.1.3. Universal

Internet delivered media today has protocols optimized for ingest and separate protocols optimized for distribution. This protocol switch in the distribution chain necessitates intermediary origins which re-package the media content. While specialization can have its benefits, there are gains in efficiency to be had in not having to re-package content. A goal of MOQT is to develop a single protocol which can be used for transmission from contribution to distribution. A related goal is the ability to support existing encoding and packaging schemas, both for backwards compatibility and for interoperability with the established content preparation ecosystem.

1.1.4. Relays

An integral feature of a protocol being successful is its ability to deliver media at scale. Greatest scale is achieved when third-party networks, independent of both the publisher and subscriber, can be leveraged to relay the content. These relays must cache content for distribution efficiency while simultaneously routing content and deterministically responding to congestion in a multi-tenant network. A goal of MOQT is to treat relays as first-class citizens of the protocol and ensure that objects are structured such that information necessary for distribution is available to relays while the media content itself remains opaque and private.

1.2. Terms 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.

TODO: The terms defined here doesn't capture the ongoing discussions within the Working Group (either as part of requirements or architecture documents). This section will be updated to reflect the discussions.

Commonly used terms in this document are described below.

Client: The party initiating a transport session.

Congestion: Packet loss and queuing caused by degraded or overloaded networks.

Consumer: A QUIC endpoint receiving media over the network.

Endpoint: A QUIC Client or a QUIC Server.

Group: A temporal sequence of objects. A group represents a join point in a track. See (Section 2.2).

Object: An object is an addressable unit whose payload is a sequence of bytes. Objects form the base element in the MOQT model. See (Section 2.1).

Producer: A QUIC endpoint sending media over the network.

Server: The party accepting an incoming transport session.

Track: An encoded bitstream. Tracks contain a sequential series of one or more groups and are the subscribable entity with MOQT.

Transport session: A raw QUIC connection or a WebTransport session.

1.3. Notational Conventions

This document uses the conventions detailed in Section 1.3 of [QUIC] when describing the binary encoding.

This document also defines an additional field type for binary data:

x (b): Indicates that x consists of a variable length integer, followed by that many bytes of binary data.

2. Object Model

MOQT has a hierarchical object model for data, comprised of objects, groups and tracks.

2.1. Objects

The basic data element of MOQT is an object. An object is an addressable unit whose payload is a sequence of bytes. All objects belong to a group, indicating ordering and potential dependencies. Section 2.2 Objects are comprised of two parts: metadata and a payload. The metadata is never encrypted and is always visible to relays. The payload portion may be encrypted, in which case it is only visible to the producer and consumer. The application is solely responsible for the content of the object payload. This includes the underlying encoding, compression, any end-to-end encryption, or authentication. A relay **MUST NOT** combine, split, or otherwise modify object payloads.

2.2. Groups

A group is a collection of objects and is a sub-unit of a track (Section 2.3). Objects within a group **SHOULD NOT** depend on objects in other groups. A group behaves as a join point for subscriptions. A new subscriber might not want to receive the entire track, and may instead opt to receive only the latest group(s). The sender then selectively transmits objects based on their group membership.

2.3. Track

A track is a sequence of groups (Section 2.2). It is the entity against which a consumer issues a subscription request. A subscriber can request to receive individual tracks starting at a group boundary, including any new objects pushed by the producer while the track is active.

2.3.1. Track Naming and Scopes

In MOQT, every track has a track name and a track namespace associated with it. A track name identifies an individual track within the namespace.

A tuple of a track name and a track namespace together is known as a full track name:

```
Full Track Name = Track Namespace Track Name
```

A MOQT scope is a set of servers (as identified by their connection URIs) for which full track names are guaranteed to be unique. This implies that within a single MOQT scope, subscribing to the same

full track name would result in the subscriber receiving the data for the same track. It is up to the application using MOQT to define how broad or narrow the scope has to be. An application that deals with connections between devices on a local network may limit the scope to a single connection; by contrast, an application that uses multiple CDNs to serve media may require the scope to include all of those CDNs.

The full track name is the only piece of information that is used to identify the track within a given MOQT scope and is used as cache key. MOQT does not provide any in-band content negotiation methods similar to the ones defined by HTTP ([RFC9110], Section 10); if, at a given moment in time, two tracks within the same scope contain different data, they have to have different full track names.

```
Example: 1
Track Namespace = live.example.com/meeting/123/member/alice/
Track Name = audio
Full Track Name = live.example.com/meeting/123/member/alice/audio

Example: 2
Track Namespace = live.example.com/
Track Name = uaCafDkl123/audio
Full Track Name = live.example.com/uaCafDkl123/audio

Example: 3
Track Namespace = security-camera.example.com/camera1/
Track Name = hd-video
Full Track Name = security-camera.example.com/camera1/hd-video
```

2.3.2. Connection URL

Each track **MAY** have one or more associated connection URLs specifying network hosts through which a track may be accessed. The syntax of the Connection URL and the associated connection setup procedures are specific to the underlying transport protocol usage Section 3.

3. Sessions

3.1. Session establishment

This document defines a protocol that can be used interchangeably both over a QUIC connection directly [QUIC], and over WebTransport [WebTransport]. Both provide streams and datagrams with similar semantics (see [I-D.ietf-webtrans-overview], Section 4); thus, the main difference lies in how the servers are identified and how the connection is established.

3.1.1. WebTransport

A MOQT server that is accessible via WebTransport can be identified using an HTTPS URI ([RFC9110], Section 4.2.2). A MOQT session can be established by sending an extended CONNECT request to the host and the path indicated by the URI, as described in [WebTransport], Section 3.

3.1.2. QUIC

A MOQT server that is accessible via native QUIC can be identified by a URI with a "moq" scheme. The "moq" URI scheme is defined as follows, using definitions from [RFC3986]:

```
moq-URI = "moq" "://" authority path-abempty [ "?" query ]
```

The authority portion **MUST NOT** contain a non-empty host portion. The moq URI scheme supports the /.well-known/ path prefix defined in [RFC8615].

This protocol does not specify any semantics on the path-abempty and query portions of the URI. The contents of those are left up to the application.

The client can establish a connection to a MoQ server identified by a given URI by setting up a QUIC connection to the host and port identified by the authority section of the URI. The path-abempty and query portions of the URI are communicated to the server using the PATH parameter (Section 6.1.1.2) which is sent in the SETUP message at the start of the session. The ALPN value [RFC7301] used by the protocol is moq-00.

3.2. Session initialization

The first stream opened is a client-initiated bidirectional stream where the peers exchange SETUP messages (Section 6.1). The subsequent streams **MAY** be either unidirectional or bidirectional. For exchanging content, an application would typically send a unidirectional stream containing a single OBJECT message (Section 6.2), as putting more than one object into one stream may create head-of-line blocking delays. However, if one object has a hard dependency on another object, putting them on the same stream could be a valid choice.

3.3. Cancellation

A QUIC stream **MAY** be canceled at any point with an error code. The producer does this via a RESET_STREAM frame while the consumer requests cancellation with a STOP_SENDING frame.

When using order, lower priority streams will be starved during congestion, perhaps indefinitely. These streams will consume resources and flow control until they are canceled. When nearing resource limits, an endpoint **SHOULD** cancel the lowest priority stream with error code 0.

The sender **MAY** cancel streams in response to congestion. This can be useful when the sender does not support stream prioritization.

TODO: this section actually describes stream cancellation, not session cancellation. Is this section required, or can it be deleted, or added to a new "workflow" section.

3.4. Termination

The transport session can be terminated at any point. When native QUIC is used, the session is closed using the CONNECTION_CLOSE frame ([QUIC], Section 19.19). When WebTransport is used, the session is closed using the CLOSE_WEBTRANSPORT_SESSION capsule ([WebTransport], Section 5).

The application **MAY** use any error message and **SHOULD** use a relevant code, as defined below:

Code	Reason
0x0	Session Terminated
0x1	Generic Error
0x2	Unauthorized
0x10	GOAWAY

Table 1

*Session Terminated: No error occurred; however the endpoint wishes to terminate the session.

*Generic Error: An unclassified error occurred.

*Unauthorized: The endpoint breached an agreement, which **MAY** have been pre-negotiated by the application.

*GOAWAY: The endpoint successfully drained the session after a GOAWAY was initiated (Section 6.9).

4. Prioritization and Congestion Response

TODO: This is a placeholder section to capture details on how the MOQT protocol deals with prioritization and congestion overall.

This section is expected to cover details on:

*Prioritization Schemes.

*Congestion Algorithms and impacts.

*Mapping considerations for one object per stream vs multiple objects per stream.

*Considerations for merging multiple streams across domains onto single connection and interactions with specific prioritization schemes.

4.1. Order Priorities and Options

At the point of this writing, the working group has not reached consensus on several important goals, such as:

*Ensuring that objects are delivered in the order intended by the emitter

*Allowing nodes and relays to skip or delay some objects to deal with congestion

*Ensuring that emitters can accurately predict the behavior of relays

*Ensuring that when relays have to skip and delay objects belonging to different tracks that they do it in a predictable way if tracks are explicitly coordinated and in a fair way if they are not.

The working group has been considering two alternatives: marking objects belonging to a track with an explicit "send order"; and, defining algorithms combining tracks, priorities and object order within a group. The two proposals are listed in Section 4.1.1 and Section 4.1.2. We expect further work before a consensus is reached.

4.1.1. Proposal - Send Order

Media is produced with an intended order, both in terms of when media should be presented (PTS) and when media should be decoded (DTS). As stated in the introduction, the network is unable to maintain this ordering during congestion without increasing latency.

The encoder determines how to behave during congestion by assigning each object a numeric send order. The send order **SHOULD** be followed when possible, to ensure that the most important media is delivered when throughput is limited. Note that the contents within each object are still delivered in order; this send order only applies to the ordering between objects.

A sender **MUST** send each object over a dedicated QUIC stream. The QUIC library should support prioritization (Section 4) such that streams are transmitted in send order.

A receiver **MUST NOT** assume that objects will be received in send order, for the following reasons:

- Newly encoded objects can have a smaller send order than outstanding objects.
- Packet loss or flow control can delay the send of individual streams.
- The sender might not support QUIC stream prioritization.

TODO: Refer to Congestion Response and Prioritization Section for further details on various proposals.

4.1.2. Proposal - Ordering by Priorities

Media is produced as a set of layers, such as for example low definition and high definition, or low frame rate and high frame rate. Each object belonging to a track and a group has two attributes: the object-id, and the priority (or layer).

When nodes or relays have to choose which object to send next, they apply the following rules:

- within the same group, objects with a lower priority number (e.g. P1) are always sent before objects with a numerically greater priority number (e.g., P2)
- within the same group, and the same priority level, objects with a lower object-id are always sent before objects with a higher object-id.
- objects from later groups are normally always sent before objects of previous groups.

The latter rule is generally agreed as a way to ensure freshness, and to recover quickly if queues and delays accumulate during a congestion period. However, there may be cases when finishing the transmission of an ongoing group results in better user experience than strict adherence to the freshness rule. We expect that that the working group will eventually reach consensus and define meta data that controls this behavior.

There have been proposals to allow emitters to coordinate the allocation of layer priorities across multiple coordinated tracks. At this point, these proposals have not reached consensus.

5. Relays

Relays are leveraged to enable distribution scale in the MoQ architecture. Relays can be used to form an overlay delivery network, similar in functionality to Content Delivery Networks (CDNs). Additionally, relays serve as policy enforcement points by validating subscribe and publish requests at the edge of a network.

5.1. Subscriber Interactions

Subscribers interact with the Relays by sending a "SUBSCRIBE REQUEST" (Section 6.3) control message for the tracks of interest. Relays **MUST** ensure subscribers are authorized to access the content associated with the Full Track Name. The authorization information can be part of subscription request itself or part of the encompassing session. The specifics of how a relay authorizes a user are outside the scope of this specification.

The subscriber making the subscribe request is notified of the result of the subscription, via "SUBSCRIBE OK" (Section 6.4) or the "SUBSCRIBE ERROR" Section 6.5 control message.

For successful subscriptions, the publisher maintains a list of subscribers for each full track name. Each new OBJECT belonging to the track is forwarded to each active subscriber, dependent on the congestion response. A subscription remains active until it expires, until the publisher of the track stops producing objects or there is a subscription error (see Section 6.5).

Relays **MAY** aggregate authorized subscriptions for a given track when multiple subscribers request the same track. Subscription aggregation allows relays to make only a single forward subscription for the track. The published content received from the forward subscription request is cached and shared among the pending subscribers.

5.2. Publisher Interactions

Publishing through the relay starts with publisher sending "ANNOUNCE" control message with a Track Namespace (Section 2.3).

Relays **MUST** ensure that publishers are authorized by:

* Verifying that the publisher is authorized to publish the content associated with the set of tracks whose Track Namespace matches the announced namespace. Specifics of where the authorization happens, either at the relays or forwarded for further processing, depends on the way the relay is managed and is application specific (typically based on prior business agreement).

Relays respond with "ANNOUNCE OK" and/or "ANNOUNCE ERROR" control messages providing the results of announcement.

OBJECT message header carry short hop-by-hop Track Id that maps to the Full Track Name (see Section 6.4). Relays use the Track ID of an incoming OBJECT message to identify its track and find the active subscribers for that track. Relays **MUST NOT** depend on OBJECT payload content for making forwarding decisions and **MUST** only depend on the fields, such as priority order and other metadata properties in the OBJECT message header. Unless determined by congestion response, Relays **MUST** forward the OBJECT message to the matching subscribers.

5.3. Relay Discovery and Failover

TODO: This section shall cover aspects of relay failover and protocol interactions.

5.4. Restoring connections through relays

TODO: This section shall cover reconnect considerations for clients when moving between the Relays.

5.5. Congestion Response at Relays

TODO: Refer to Section 4. Add details to describe relay behavior when merging or splitting streams and interactions with congestion response.

5.6. Relay Object Handling

MOQT encodes the delivery information for a stream via OBJECT headers (Section 6.2).

A relay **MUST** treat the object payload as opaque. A relay **MUST NOT** combine, split, or otherwise modify object payloads. A relay **SHOULD** prioritize streams (Section 4) based on the send order/priority.

A relay that reads from a stream and writes to stream in order will introduce head-of-line blocking. Packet loss will cause stream data to be buffered in the QUIC library, awaiting in order delivery, which will increase latency over additional hops. To mitigate this, a relay **SHOULD** read and write QUIC stream data out of order subject to flow control limits. See section 2.2 in [QUIC].

6. Messages

Both unidirectional and bidirectional QUIC streams contain sequences of length-delimited messages.

```
MOQT Message {
  Message Type (i),
  Message Length (i),
  Message Payload (..),
}
```

Figure 1: MOQT Message

The Message Length field contains the length of the Message Payload field in bytes. A length of 0 indicates the message is unbounded and continues until the end of the stream.

ID	Messages
0x0	OBJECT (Section 6.2)
0x1	SETUP (Section 6.1)
0x3	SUBSCRIBE REQUEST (Section 6.3)
0x4	SUBSCRIBE OK (Section 6.4)
0x5	SUBSCRIBE ERROR (Section 6.5)
0x6	ANNOUNCE (Section 6.6)
0x7	ANNOUNCE OK (Section 6.7)
0x8	ANNOUNCE ERROR (Section 6.8)
0x10	GOAWAY (Section 6.9)

Table 2

6.1. SETUP

The SETUP message is the first message that is exchanged by the client and the server; it allows the peers to establish the mutually supported version and agree on the initial configuration before any objects are exchanged. It is a sequence of key-value pairs called SETUP parameters; the semantics and format of which can vary based on whether the client or server is sending. To ensure future extensibility of MOQT, the peers **MUST** ignore unknown setup parameters. TODO: describe GREASE for those.

The wire format of the SETUP message is as follows:

```
SETUP Parameter {
  Parameter Key (i),
  Parameter Value Length (i),
  Parameter Value (..),
}

Client SETUP Message Payload {
  Number of Supported Versions (i),
  Supported Version (i) ...,
  SETUP Parameters (..) ...,
}

Server SETUP Message Payload {
  Selected Version (i),
  SETUP Parameters (..) ...,
}
```

Figure 2: MOQT SETUP Message

The Parameter Value Length field indicates the length of the Parameter Value.

The client offers the list of the protocol versions it supports; the server **MUST** reply with one of the versions offered by the client. If the server does not support any of the versions offered by the client, or the client receives a server version that it did not offer, the corresponding peer **MUST** close the connection.

The SETUP parameters are described in the Section 6.1.1 section.

6.1.1. SETUP Parameters

Every parameter **MUST** appear at most once within the SETUP message. The peers **SHOULD** verify that and close the connection if a parameter appears more than once.

The ROLE parameter is mandatory for the client. All of the other parameters are optional.

6.1.1.1. ROLE parameter

The ROLE parameter (key 0x00) allows the client to specify what roles it expects the parties to have in the MOQT connection. It has three possible values:

0x01: Only the client is expected to send objects on the connection. This is commonly referred to as the ingestion case.

0x02: Only the server is expected to send objects on the connection. This is commonly referred to as the delivery case.

0x03:
Both the client and the server are expected to send objects.

The client **MUST** send a ROLE parameter with one of the three values specified above. The server **MUST** close the connection if the ROLE parameter is missing, is not one of the three above-specified values, or it is different from what the server expects based on the application.

6.1.1.2. PATH parameter

The PATH parameter (key 0x01) allows the client to specify the path of the MoQ URI when using native QUIC ([QUIC]). It **MUST NOT** be used by the server, or when WebTransport is used. If the peer receives a PATH parameter from the server, or when WebTransport is used, it **MUST** close the connection.

When connecting to a server using a URI with the "moq" scheme, the client **MUST** set the PATH parameter to the path-abempty portion of the URI; if query is present, the client **MUST** concatenate ?, followed by the query portion of the URI to the parameter.

6.2. OBJECT

A OBJECT message contains a range of contiguous bytes from from the specified track, as well as associated metadata required to deliver, cache, and forward it.

The format of the OBJECT message is as follows:

```
OBJECT Message {
  Track ID (i),
  Group Sequence (i),
  Object Sequence (i),
  Object Send Order (i),
  Object Payload (b),
}
```

Figure 3: MOQT OBJECT Message

*Track ID: The track identifier obtained as part of subscription and/or publish control message exchanges.

*Group Sequence : The object is a member of the indicated group Section 2.2 within the track.

*Object Sequence: The order of the object within the group. The sequence starts at 0, increasing sequentially for each object within the group.

*Object Send Order: An integer indicating the object send order Section 4.1.1 or priority Section 4.1.2 value.

*Object Payload: An opaque payload intended for the consumer and **SHOULD NOT** be processed by a relay.

6.3. SUBSCRIBE REQUEST

A receiver issues a SUBSCRIBE REQUEST to a publisher to request a track.

The format of SUBSCRIBE REQUEST is as follows:

```
Track Request Parameter {
  Track Request Parameter Key (i),
  Track Request Parameter Length (i),
  Track Request Parameter Value (..),
}

SUBSCRIBE REQUEST Message {
  Full Track Name Length (i),
  Full Track Name (...),
  Track Request Parameters (..) ...
}
```

Figure 4: MOQT SUBSCRIBE REQUEST Message

*Full Track Name: Identifies the track as defined in (Section 2.3.1).

*Track Request Parameters: As defined in Section 6.6.1.

On successful subscription, the publisher **SHOULD** start delivering objects from the group sequence and object sequence as defined in the Track Request Parameters.

6.4. SUBSCRIBE OK

A SUBSCRIBE OK control message is sent for successful subscriptions.

```
SUBSCRIBE OK
{
  Full Track Name Length(i),
  Full Track Name(...),
  Track ID(i),
  Expires (i)
}
```

Figure 5: MOQT SUBSCRIBE OK Message

*Full Track Name: Identifies the track for which this response is provided.

*Track ID: Session specific identifier that is used as an alias for the Full Track Name in the Track ID field of the OBJECT (Section 6.2) message headers of the requested track. Track IDs are generally shorter than Full Track Names and thus reduce the overhead in OBJECT messages.

*Expires: Time in milliseconds after which the subscription is no longer valid. A value of 0 indicates that the subscription stays active until it is explicitly unsubscribed.

6.5. SUBSCRIBE ERROR

A publisher sends a SUBSCRIBE ERROR control message in response to a failed SUBSCRIBE REQUEST.

```
SUBSCRIBE ERROR
{
  Full Track Name Length(i),
  Full Track Name(...),
  Error Code (i),
  Reason Phrase Length (i),
  Reason Phrase (...),
}
```

Figure 6: MOQT SUBSCRIBE ERROR Message

*Full Track Name: Identifies the track in the request message for which this response is provided.

*Error Code: Identifies an integer error code for subscription failure.

*Reason Phrase Length: The length in bytes of the reason phrase.

*Reason Phrase: Provides the reason for subscription error and Reason Phrase Length field carries its length.

6.6. ANNOUNCE

The publisher sends the ANNOUNCE control message to advertise where the receiver can route SUBSCRIBE REQUESTs for tracks within the announced Track Namespace. The receiver verifies the publisher is authorized to publish tracks under this namespace.

```
ANNOUNCE Message {
  Track Namespace Length(i),
  Track Namespace,
  Track Request Parameters (..) ...,
}
```

Figure 7: MOQT ANNOUNCE Message

*Track Namespace: Identifies a track's namespace as defined in (Section 2.3.1)

*Track Request Parameters: The parameters are defined in Section 6.6.1.

6.6.1. Track Request Parameters

The Track Request Parameters identify properties of the track requested in either the ANNOUNCE or SUSBCRIBE REQUEST control messages. The peers **MUST** close the connection if there are duplicates. The Parameter Value Length field indicates the length of the Parameter Value.

6.6.1.1. GROUP SEQUENCE Parameter

The GROUP SEQUENCE parameter (key 0x00) identifies the group within the track to start delivering objects. The publisher **MUST** start delivering the objects from the most recent group, when this parameter is omitted. This parameter is applicable in SUBSCRIBE REQUEST message.

6.6.1.2. OBJECT SEQUENCE Parameter

The OBJECT SEQUENCE parameter (key 0x01) identifies the object with the track to start delivering objects. The GROUP SEQUENCE parameter **MUST** be set to identify the group under which to start delivery. The publisher **MUST** start delivering from the beginning of the selected group when this parameter is omitted. This parameter is applicable in SUBSCRIBE REQUEST message.

6.6.1.3. AUTHORIZATION INFO Parameter

AUTHORIZATION INFO parameter (key 0x02) identifies track's authorization information. This parameter is populated for cases where the authorization is required at the track level. This parameter is applicable in SUBSCRIBE REQUEST and ANNOUNCE messages.

6.7. ANNOUNCE OK

The receiver sends an ANNOUNCE OK control message to acknowledge the successful authorization and acceptance of an ANNOUNCE message.

```
ANNOUNCE OK
{
  Track Namespace
}
```

Figure 8: MOQT ANNOUNCE OK Message

*Track Namespace: Identifies the track namespace in the ANNOUNCE message for which this response is provided.

6.8. ANNOUNCE ERROR

The receiver sends an ANNOUNCE ERROR control message for tracks that failed authorization.

```
ANNOUNCE ERROR
{
  Track Namespace Length(i),
  Track Namespace(...),
  Error Code (i),
  Reason Phrase Length (i),
  Reason Phrase (...),
}
```

Figure 9: MOQT ANNOUNCE ERROR Message

*Track Namespace: Identifies the track namespace in the ANNOUNCE message for which this response is provided.

*Error Code: Identifies an integer error code for announcement failure.

*Reason Phrase: Provides the reason for announcement error and Reason Phrase Length field carries its length.

6.9. GOAWAY

The server sends a GOAWAY message to force the client to reconnect. This is useful for server maintenance or reassignments without severing the QUIC connection. The server can be a producer or a consumer.

The server:

*__MAY__ initiate a graceful shutdown by sending a GOAWAY message.

*__MUST__ close the QUIC connection after a timeout with the GOAWAY error code (Section 3.4).

* **MAY** close the QUIC connection with a different error code if there is a fatal error before shutdown.

* **SHOULD** wait until the GOAWAY message and any pending streams have been fully acknowledged, plus an extra delay to ensure they have been processed.

The client:

* **MUST** establish a new transport session upon receipt of a GOAWAY message, assuming it wants to continue operation.

* **SHOULD** establish the new transport session using a different QUIC connection to that on which it received the GOAWAY message.

* **SHOULD** remain connected on both connections for a short period, processing objects from both in parallel.

7. Security Considerations

TODO: Expand this section.

7.1. Resource Exhaustion

Live content requires significant bandwidth and resources. Failure to set limits will quickly cause resource exhaustion.

MOQT uses QUIC flow control to impose resource limits at the network layer. Endpoints **SHOULD** set flow control limits based on the anticipated bitrate.

Endpoints **MAY** impose a MAX STREAM count limit which would restrict the number of concurrent streams which a MOQT Streaming Format could have in flight.

The producer prioritizes and transmits streams out of order. Streams might be starved indefinitely during congestion. The producer and consumer **MUST** cancel a stream, preferably the lowest priority, after reaching a resource limit.

8. IANA Considerations

TODO: fill out currently missing registries: * MOQT version numbers * SETUP parameters * Track Request parameters * Subscribe Error codes * Announce Error codes * Track format numbers * Message types * Object headers

TODO: register the URI scheme and the ALPN

TODO: the MOQT spec should establish the IANA registration table for MoQ Streaming Formats. Each MoQ streaming format can then register its type in that table. The MoQ Streaming Format type **MUST** be carried as the leading varint in catalog track objects.

Contributors

- *Alan Frindell
- *Ali Begen
- *Charles Krasic
- *Christian Huitema
- *Cullen Jennings
- *James Hurley
- *Jordi Cenzano
- *Mike English
- *Mo Zanaty
- *Will Law

References

Normative References

[QUIC] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[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>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/

RFC7301, July 2014, <https://www.rfc-editor.org/rfc/rfc7301>.

[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>.

[RFC8615] Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", RFC 8615, DOI 10.17487/RFC8615, May 2019, <https://www.rfc-editor.org/rfc/rfc8615>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[WebTransport] Frindell, A., Kinnear, E., and V. Vasiliev, "WebTransport over HTTP/3", Work in Progress, Internet-Draft, draft-ietf-webtrans-http3-06, 25 May 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-http3-06>.

Informative References

[I-D.ietf-webtrans-overview]
Vasiliev, V., "The WebTransport Protocol Framework", Work in Progress, Internet-Draft, draft-ietf-webtrans-overview-05, 24 January 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-overview-05>.

Authors' Addresses

Luke Curley
Twitch

Email: kixelated@gmail.com

Kirill Pugin
Meta

Email: ikir@meta.com

Suhas Nandakumar
Cisco

Email: snandaku@cisco.com

Victor Vasiliev
Google

Email: vasilvv@google.com