

Media Over QUIC  
Internet-Draft  
Intended status: Standards Track  
Expires: 6 March 2026

S. Nandakumar  
Cisco  
V. Vasiliev  
I. Swett, Ed.  
Google  
A. Frindell, Ed.  
Meta  
2 September 2025

Media over QUIC Transport  
draft-ietf-moq-transport-14

## Abstract

This document defines the core behavior for Media over QUIC Transport (MOQT), a media transport protocol designed to operate over QUIC and WebTransport, which have similar functionality. 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.

## About This Document

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

The latest revision of this draft can be found at <https://moq-wg.github.io/moq-transport/draft-ietf-moq-transport.html>. Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-ietf-moq-transport/>.

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

Source for this draft and an issue tracker can be found at <https://github.com/moq-wg/moq-transport>.

## 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 6 March 2026.

## Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

1. Introduction . . . . .	5
1.1. Motivation . . . . .	6
1.1.1. Latency . . . . .	6
1.1.2. Leveraging QUIC . . . . .	6
1.1.3. Convergence . . . . .	6
1.1.4. Relays . . . . .	7
1.2. Terms and Definitions . . . . .	7
1.3. Stream Management Terms . . . . .	8
1.4. Notational Conventions . . . . .	8
1.4.1. Location Structure . . . . .	9
1.4.2. Key-Value-Pair Structure . . . . .	9
1.4.3. Reason Phrase Structure . . . . .	10
2. Object Data Model . . . . .	10
2.1. Objects . . . . .	11
2.2. Subgroups . . . . .	11
2.3. Groups . . . . .	12
2.3.1. Group IDs . . . . .	13
2.4. Track . . . . .	13
2.4.1. Track Naming . . . . .	13
2.5. Malformed Tracks . . . . .	14

2.5.1. Scope . . . . .	15
3. Sessions . . . . .	16
3.1. Session establishment . . . . .	16
3.1.1. WebTransport . . . . .	16
3.1.2. QUIC . . . . .	16
3.1.3. Connection URL . . . . .	17
3.2. Version and Extension Negotiation . . . . .	17
3.3. Session initialization . . . . .	18
3.4. Termination . . . . .	18
3.5. Migration . . . . .	20
3.6. Congestion Control . . . . .	20
3.6.1. Bufferbloat . . . . .	21
3.6.2. Application-Limited . . . . .	21
3.6.3. Consistent Throughput . . . . .	21
4. Modularity . . . . .	21
5. Publishing and Retrieving Tracks . . . . .	22
5.1. Subscriptions . . . . .	22
6. Namespace Discovery . . . . .	23
6.1. Subscribing to Namespaces . . . . .	24
6.2. Publishing Namespaces . . . . .	24
7. Priorities . . . . .	25
7.1. Definitions . . . . .	25
7.2. Scheduling Algorithm . . . . .	26
7.3. Considerations for Setting Priorities . . . . .	27
8. Relays . . . . .	28
8.1. Caching Relays . . . . .	28
8.2. Multiple Publishers . . . . .	29
8.3. Subscriber Interactions . . . . .	29
8.3.1. Graceful Subscriber Relay Switchover . . . . .	30
8.4. Publisher Interactions . . . . .	30
8.4.1. Graceful Publisher Network Switchover . . . . .	32
8.4.2. Graceful Publisher Relay Switchover . . . . .	33
8.5. Relay Object Handling . . . . .	33
9. Control Messages . . . . .	33
9.1. Request ID . . . . .	35
9.2. Parameters . . . . .	36
9.2.1. Version Specific Parameters . . . . .	36
9.3. CLIENT_SETUP and SERVER_SETUP . . . . .	41
9.3.1. Versions . . . . .	42
9.3.2. Setup Parameters . . . . .	42
9.4. GOAWAY . . . . .	44
9.5. MAX_REQUEST_ID . . . . .	45
9.6. REQUESTS_BLOCKED . . . . .	46
9.7. SUBSCRIBE . . . . .	46
9.8. SUBSCRIBE_OK . . . . .	49
9.9. SUBSCRIBE_ERROR . . . . .	51
9.10. SUBSCRIBE_UPDATE . . . . .	52
9.11. UNSUBSCRIBE . . . . .	53

9.12. PUBLISH_DONE . . . . .	54
9.13. PUBLISH . . . . .	56
9.14. PUBLISH_OK . . . . .	57
9.15. PUBLISH_ERROR . . . . .	58
9.16. FETCH . . . . .	59
9.16.1. Standalone Fetch . . . . .	60
9.16.2. Joining Fetches . . . . .	60
9.16.3. Fetch Handling . . . . .	61
9.17. FETCH_OK . . . . .	63
9.18. FETCH_ERROR . . . . .	65
9.19. FETCH_CANCEL . . . . .	66
9.20. TRACK_STATUS . . . . .	66
9.21. TRACK_STATUS_OK . . . . .	67
9.22. TRACK_STATUS_ERROR . . . . .	67
9.23. PUBLISH_NAMESPACE . . . . .	67
9.24. PUBLISH_NAMESPACE_OK . . . . .	68
9.25. PUBLISH_NAMESPACE_ERROR . . . . .	68
9.26. PUBLISH_NAMESPACE_DONE . . . . .	69
9.27. PUBLISH_NAMESPACE_CANCEL . . . . .	70
9.28. SUBSCRIBE_NAMESPACE . . . . .	70
9.29. SUBSCRIBE_NAMESPACE_OK . . . . .	71
9.30. SUBSCRIBE_NAMESPACE_ERROR . . . . .	72
9.31. UNSUBSCRIBE_NAMESPACE . . . . .	73
10. Data Streams and Datagrams . . . . .	73
10.1. Track Alias . . . . .	74
10.2. Objects . . . . .	74
10.2.1. Canonical Object Properties . . . . .	74
10.3. Datagrams . . . . .	77
10.3.1. Object Datagram . . . . .	77
10.4. Streams . . . . .	79
10.4.1. Stream Cancellation . . . . .	79
10.4.2. Subgroup Header . . . . .	79
10.4.3. Closing Subgroup Streams . . . . .	82
10.4.4. Fetch Header . . . . .	85
10.5. Examples . . . . .	86
11. Extension Headers . . . . .	87
11.1. Prior Group ID Gap . . . . .	87
11.2. Immutable Extensions . . . . .	88
11.3. Prior Object ID Gap . . . . .	89
12. Security Considerations . . . . .	89
12.1. Resource Exhaustion . . . . .	90
12.2. Timeouts . . . . .	90
12.3. Relay security considerations . . . . .	90
12.3.1. State maintenance . . . . .	90
12.3.2. SUBSCRIBE_NAMESPACE with short prefixes . . . . .	90
13. IANA Considerations . . . . .	91
13.1. Error Codes . . . . .	91
13.1.1. Session Termination Error Codes . . . . .	91

13.1.2.	SUBSCRIBE_ERROR Codes . . . . .	92
13.1.3.	PUBLISH_DONE Codes . . . . .	93
13.1.4.	PUBLISH_ERROR Codes . . . . .	93
13.1.5.	FETCH_ERROR Codes . . . . .	94
13.1.6.	ANNOUNCE_ERROR Codes . . . . .	94
13.1.7.	SUBSCRIBE_NAMESPACE_ERROR Codes . . . . .	95
13.1.8.	Data Stream Reset Error Codes . . . . .	95
Contributors	. . . . .	96
References	. . . . .	96
Normative References	. . . . .	96
Informative References	. . . . .	97
Appendix A. Change Log	. . . . .	98
A.1. Since draft-ietf-moq-transport-13	. . . . .	98
A.2. Since draft-ietf-moq-transport-12	. . . . .	100
A.3. Since draft-ietf-moq-transport-11	. . . . .	100
A.4. Since draft-ietf-moq-transport-10	. . . . .	101
Authors' Addresses	. . . . .	102

## 1. Introduction

Media Over QUIC Transport (MOQT) is a protocol that is optimized for the QUIC 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 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 data model employed by MOQT.
- \* Section 3 covers aspects of setting up an MOQT session.
- \* Section 7 covers mechanisms for prioritizing subscriptions.
- \* Section 8 covers behavior at the relay entities.
- \* Section 9 covers how control messages are encoded on the wire.
- \* Section 10 covers how data 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 robust feature set of QUIC and relay support.

### 1.1.1. 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 cause queuing along the path from producer to consumer. The speed at which a protocol can detect and respond to congestion determines the overall latency. TCP-based protocols are simple but are slow to detect congestion and suffer from head-of-line blocking. Protocols utilizing UDP directly can avoid queuing, but the application is then 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.

### 1.1.3. Convergence

Some live media architectures today have separate protocols for ingest and distribution, for example RTMP and HTTP based HLS or DASH. Switching protocols necessitates intermediary origins which re-package the media content. While specialization can have its benefits, there are efficiency gains 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.

The following terms are used with the first letter capitalized.

Application: The entity using MOQT to transmit and receive data.

Client: The party initiating a Transport Session.

Server: The party accepting an incoming Transport Session.

Endpoint: A Client or Server.

Peer: The other endpoint than the one being described

Publisher: An endpoint that handles subscriptions by sending requested Objects from the requested track.

Subscriber: An endpoint that subscribes to and receives tracks.

Original Publisher: The initial publisher of a given track.

End Subscriber: A subscriber that initiates a subscription and does not send the data on to other subscribers.

Relay: An entity that is both a Publisher and a Subscriber, is not the Original Publisher or End Subscriber, and conforms to all requirements in Section 8.

Upstream: In the direction of the Original Publisher

Downstream: In the direction of the End Subscriber(s)

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

Stream: A bidirectional or unidirectional bytestream provided by the QUIC transport or WebTransport.

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

Group: A temporal sequence of objects. A group represents a join point in a track. See (Section 2.3).

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

Track: A track is a collection of groups. See (Section 2.4).

### 1.3. Stream Management Terms

This document uses stream management terms described in [RFC9000], Section 1.3 including STOP\_SENDING, RESET\_STREAM and FIN.

### 1.4. Notational Conventions

This document uses the conventions detailed in ([RFC9000], Section 1.3) when describing the binary encoding.

As a quick reference, the following list provides a non normative summary of the parts of RFC9000 field syntax that are used in this specification.

x (L): Indicates that x is L bits long

x (i): Indicates that x holds an integer value using the variable-length encoding as described in ([RFC9000], Section 16)

x (...): Indicates that x can be any length including zero bits long. Values in this format always end on a byte boundary.

[x (L)]: Indicates that x is optional and has a length of L

x (L) ...: Indicates that x is repeated zero or more times and that each instance has a length of L

This document extends the RFC9000 syntax and with the additional field types:

x (b): Indicates that x consists of a variable length integer encoding as described in ([RFC9000], Section 16), followed by that many bytes of binary data

x (tuple): Indicates that x is a tuple, consisting of a variable length integer encoded as described in ([RFC9000], Section 16), followed by that many variable length tuple fields, each of which are encoded as (b) above.

To reduce unnecessary use of bandwidth, variable length integers SHOULD be encoded using the least number of bytes possible to represent the required value.

#### 1.4.1. Location Structure

Location identifies a particular Object in a Group within a Track.

```
Location {  
    Group (i),  
    Object (i)  
}
```

Figure 1: Location structure

In this document, the constituent parts of any Location A can be referred to using A.Group or A.Object.

Location A < Location B if:

A.Group < B.Group || (A.Group == B.Group && A.Object < B.Object)

#### 1.4.2. Key-Value-Pair Structure

Key-Value-Pair is a flexible structure designed to carry key/value pairs in which the key is a variable length integer and the value is either a variable length integer or a byte field of arbitrary length.

Key-Value-Pair is used in both the data plane and control plane, but is optimized for use in the data plane.

```
Key-Value-Pair {  
    Type (i),  
    [Length (i),]  
    Value (...)  
}
```

Figure 2: MOQT Key-Value-Pair

- \* **Type:** an unsigned integer, encoded as a varint, identifying the type of the value and also the subsequent serialization.
- \* **Length:** Only present when Type is odd. Specifies the length of the Value field. The maximum length of a value is  $2^{16}-1$  bytes. If an endpoint receives a length larger than the maximum, it **MUST** close the session with a Protocol Violation.
- \* **Value:** A single varint encoded value when Type is even, otherwise a sequence of Length bytes.

If a receiver understands a Type, and the following Value or Length/Value does not match the serialization defined by that Type, the receiver **MUST** terminate the session with error code `KEY_VALUE_FORMATTING_ERROR`.

#### 1.4.3. Reason Phrase Structure

Reason Phrase provides a way for the sender to encode additional diagnostic information about the error condition, where appropriate.

```
Reason Phrase {  
    Reason Phrase Length (i),  
    Reason Phrase Value (..)  
}
```

- \* **Reason Phrase Length:** A variable-length integer specifying the length of the reason phrase in bytes. The reason phrase length has a maximum length of 1024 bytes. If an endpoint receives a length exceeding the maximum, it **MUST** close the session with a `PROTOCOL_VIOLATION`
- \* **Reason Phrase Value:** Additional diagnostic information about the error condition. The reason phrase value is encoded as UTF-8 string and does not carry information, such as language tags, that would aid comprehension by any entity other than the one that created the text.

## 2. Object Data Model

MOQT has a hierarchical data model, comprised of tracks which contain groups, and groups that contain objects. Inside of a group, the objects can be organized into subgroups.

To give an example of how an application might use this data model, consider an application sending high and low resolution video using a codec with temporal scalability. Each resolution is sent as a separate track to allow the subscriber to pick the appropriate

resolution given the display environment and available bandwidth. Each independently coded sequence of pictures in a resolution is sent as a group as the first picture in the sequence can be used as a random access point. This allows the client to join at the logical points where decoding of the media can start without needing information before the join points. The temporal layers are sent as separate subgroups to allow the priority mechanism to favor lower temporal layers when there is not enough bandwidth to send all temporal layers. Each frame of video is sent as a single object.

## 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 (see Section 2.3). An object is uniquely identified by its track namespace, track name, group ID, and object ID, and must be an identical sequence of bytes regardless of how or where it is retrieved. An Object can become unavailable, but its contents MUST NOT change over time.

Objects are comprised of two parts: metadata and a payload. The metadata is never encrypted and is always visible to relays (see Section 8). The payload portion may be encrypted, in which case it is only visible to the Original Publisher and End Subscribers. The Original Publisher 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.

Objects within a Group are ordered numerically by their Object ID.

## 2.2. Subgroups

A subgroup is a sequence of one or more objects from the same group (Section 2.3) in ascending order by Object ID. Objects in a subgroup have a dependency and priority relationship consistent with sharing a stream and are sent on a single stream whenever possible. A Group is delivered using at least as many streams as there are Subgroups, typically with a one-to-one mapping between Subgroups and streams.

When a Track's forwarding preference (see Section 10.2.1) is "Datagram", Objects are not sent in Subgroups and the description in the remainder of this section does not apply.

Streams offer in-order reliable delivery and the ability to cancel sending and retransmission of data. Furthermore, many implementations offer the ability to control the relative priority of streams, which allows control over the scheduling of sending data on active streams.

Every object within a Group belongs to exactly one Subgroup.

Objects from two subgroups cannot be sent on the same stream. Objects from the same Subgroup MUST NOT be sent on different streams, unless one of the streams was reset prematurely, or upstream conditions have forced objects from a Subgroup to be sent out of Object ID order.

Original publishers assign each Subgroup a Subgroup ID, and do so as they see fit. The scope of a Subgroup ID is a Group, so Subgroups from different Groups MAY share a Subgroup ID without implying any relationship between them. In general, publishers assign objects to subgroups in order to leverage the features of streams as described above.

In general, if Object B is dependent on Object A, then delivery of B can follow A, i.e. A and B can be usefully delivered over a single stream. If an Object is dependent on all previous Objects in a Subgroup, it likely fits best in that Subgroup. If an Object is not dependent on any of the Objects in a Subgroup, it likely belongs in a different Subgroup.

When assigning Objects to different Subgroups, the Original Publisher makes a reasonable tradeoff between having an optimal mapping of Object relationships in a Group and minimizing the number of streams used.

### 2.3. Groups

A group is a collection of Objects and is a sub-unit of a Track (Section 2.4). Groups SHOULD be independently useful, so Objects within a Group SHOULD NOT depend on Objects in other Groups. A Group provides a join point for subscriptions, so a subscriber that does not want to receive the entire Track can opt to receive only Groups starting from a given Group ID. Groups can contain any number of Objects.

### 2.3.1. Group IDs

Within a track, the original publisher SHOULD publish Group IDs which increase with time (where "time" is defined according to the internal clock of the media being sent). In some cases, Groups will be produced in increasing order, but sent to subscribers in a different order, for example when the subscription's Group Order is Descending. Due to network reordering and the partial reliability features of MOQT, Groups can always be received out of order.

As a result, subscribers cannot infer the existence of a Group until an object in the Group is received. This can create gaps in a cache that can be filled by doing a Fetch upstream, if necessary.

Applications that cannot produce Group IDs that increase with time are limited to the subset of MOQT that does not compare group IDs. Subscribers to these Tracks SHOULD NOT use range filters which span multiple Groups in FETCH or SUBSCRIBE. SUBSCRIBE and FETCH delivery use Group Order, so a FETCH cannot deliver Groups out of order and a subscription could have unexpected delivery order if Group IDs do not increase with time.

Note that the increase in time between two groups is not defined by the protocol.

## 2.4. Track

A track is a sequence of groups (Section 2.3). It is the entity against which a subscriber issues a subscription request. A subscriber can request to receive individual tracks starting at a group boundary, including any new objects pushed by the publisher while the track is active.

### 2.4.1. Track Naming

In MOQT, every track is identified by a Full Track Name, consisting of a Track Namespace and a Track Name.

Track Namespace is an ordered N-tuple of bytes where N can be between 1 and 32. The structured nature of Track Namespace allows relays and applications to manipulate prefixes of a namespace. If an endpoint receives a Track Namespace tuple with an N of 0 or more than 32, it MUST close the session with a Protocol Violation.

Track Name is a sequence of bytes that identifies an individual track within the namespace.

The maximum total length of a Full Track Name is 4,096 bytes, computed as the sum of the lengths of each Track Namespace tuple field and the Track Name length field. If an endpoint receives a Full Track Name exceeding this length, it MUST close the session with a `PROTOCOL_VIOLATION`.

In this specification, both the Track Namespace tuple fields and the Track Name are not constrained to a specific encoding. They carry a sequence of bytes and comparison between two Track Namespace tuple fields or Track Names is done by exact comparison of the bytes. Specifications that use MOQT may constrain the information in these fields, for example by restricting them to UTF-8. Any such specification needs to specify the canonicalization into the bytes in the Track Namespace or Track Name such that exact comparison works.

## 2.5. Malformed Tracks

There are multiple ways a publisher can transmit a Track that does not conform to MOQT constraints. Such a Track is considered malformed. Some example conditions that constitute a malformed track when detected by a receiver include:

1. An Object is received in a `FETCH` response with the same Group as the previous Object, but whose Object ID is not strictly larger than the previous object.
2. An Object is received in an Ascending `FETCH` response whose Group ID is smaller than the previous Object in the response.
3. An Object is received in a Descending `FETCH` response whose Group ID is larger than the previous Object in the response.
4. An Object is received whose Object ID is larger than the final Object in the Subgroup. The final Object in a Subgroup is the last Object received on a Subgroup stream before a `FIN`.
5. A Subgroup is received with two or more different final Objects.
6. An Object is received in a Group whose Object ID is larger than the final Object in the Group. The final Object in a Group is the Object with Status `END_OF_GROUP` or the last Object sent in a `FETCH` that requested the entire Group.
7. An Object is received on a Track whose Group and Object ID are larger than the final Object in the Track. The final Object in a Track is the Object with Status `END_OF_TRACK` or the last Object sent in a `FETCH` whose response indicated End of Track.

8. The same Object is received more than once with different Payload or other immutable properties.
9. An Object is received with a different Forwarding Preference than previously observed from the same Track.

The above list of conditions is not considered exhaustive.

When a subscriber detects a Malformed Track, it MUST UNSUBSCRIBE any subscription and FETCH\_CANCEL any fetch for that Track from that publisher, and SHOULD deliver an error to the application. If a relay detects a Malformed Track, it MUST immediately terminate downstream subscriptions with PUBLISH\_DONE and reset any fetch streams with Status Code MALFORMED\_TRACK.

#### 2.5.1. Scope

An MOQT scope is a set of servers (as identified by their connection URIs) for which the tuple of Track Namespace and Track Name are guaranteed to be unique and identify a specific track. It is up to the application using MOQT to define how broad or narrow the scope is. 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.

Because the tuple of Track Namespace and Track Name are unique within an MOQT scope, they can be used as a cache key for the track. If, at a given moment in time, two tracks within the same scope contain different data, they MUST have different names and/or namespaces. MOQT provides subscribers with the ability to alter the specific manner in which tracks are delivered via Parameters, but the actual content of the tracks does not depend on those parameters; this is in contrast to protocols like HTTP, where request headers can alter the server response.

A publisher that loses state (e.g. crashes) and intends to resume publishing on the same Track risks colliding with previously published Objects and violating the above requirements. A publisher can handle this in application specific ways, for example:

1. Select a unique Track Name or Track Namespace whenever it resumes publishing. For example, it can base one of the Namespace tuple fields on the current time, or select a sufficiently large random value.

2. Resume publishing under a previous Track Name and Namespace and set the initial Group ID to a unique value guaranteed to be larger than all previously used groups. This can be done by choosing a Group ID based on the current time.
3. Use TRACK\_STATUS or similar mechanism to query the previous state to determine the largest published Group ID.

### 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. The [QUIC-DATAGRAM] extension MUST be supported and negotiated in the QUIC connection used for MOQT, which is already a requirement for WebTransport over HTTP/3. The RESET\_STREAM\_AT [I-D.draft-ietf-quic-reliable-stream-reset] extension to QUIC can be used by MOQT, but the protocol is also designed to work correctly when the extension is not supported.

There is no definition of the protocol over other transports, such as TCP, and applications using MoQ might need to fallback to another protocol when QUIC or WebTransport aren't available.

##### 3.1.1. WebTransport

An MOQT server that is accessible via WebTransport can be identified using an HTTPS URI ([RFC9110], Section 4.2.2). An 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

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

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

The authority portion MUST NOT contain an empty host portion. The moqt 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 authority, path-abempty and query portions of the URI are also transmitted in SETUP parameters (see Section 9.3.2).

The ALPN value [RFC7301] used by the protocol is moq-00.

### 3.1.3. 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 (see Section 3).

## 3.2. Version and Extension Negotiation

Endpoints use the exchange of Setup messages to negotiate the MOQT version and any extensions to use.

The client indicates the MOQT versions it supports in the CLIENT\_SETUP message (see Section 9.3). It also includes the union of all Setup Parameters (see Section 9.3.2) required for a handshake by any of those versions.

Within any MOQT version, clients request the use of extensions by adding Setup parameters corresponding to that extension. No extensions are defined in this document.

The server replies with a SERVER\_SETUP message that indicates the chosen version, includes all parameters required for a handshake in that version, and parameters for every extension requested by the client that it supports.

New versions of MOQT MUST specify which existing extensions can be used with that version. New extensions MUST specify the existing versions with which they can be used.

If a given parameter carries the same information in multiple versions, but might have different optimal values in those versions, there SHOULD be separate Setup parameters for that information in each version.

### 3.3. Session initialization

The first stream opened is a client-initiated bidirectional control stream where the endpoints exchange Setup messages (Section 9.3), followed by other messages defined in Section 9.

This draft only specifies a single use of bidirectional streams. Objects are sent on unidirectional streams. Because there are no other uses of bidirectional streams, a peer MAY close the session as a PROTOCOL\_VIOLATION if it receives a second bidirectional stream.

The control stream MUST NOT be closed at the underlying transport layer while the session is active. Doing so results in the session being closed as a PROTOCOL\_VIOLATION.

### 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).

When terminating the Session, the application MAY use any error message and SHOULD use a relevant code, as defined below:

NO\_ERROR (0x0): The session is being terminated without an error.

INTERNAL\_ERROR (0x1): An implementation specific error occurred.

UNAUTHORIZED (0x2): The client is not authorized to establish a session.

PROTOCOL\_VIOLATION (0x3): The remote endpoint performed an action that was disallowed by the specification.

INVALID\_REQUEST\_ID (0x4): The session was closed because the endpoint used a Request ID that was smaller than or equal to a previously received request ID, or the least-significant bit of the request ID was incorrect for the endpoint.

DUPLICATE\_TRACK\_ALIAS (0x5): The endpoint attempted to use a Track Alias that was already in use.

KEY\_VALUE\_FORMATTING\_ERROR (0x6): The key-value pair has a formatting error.

TOO\_MANY\_REQUESTS (0x7): The session was closed because the endpoint

used a Request ID equal to or larger than the current Maximum Request ID.

INVALID\_PATH (0x8): The PATH parameter was used by a server, on a WebTransport session, or the server does not support the path.

MALFORMED\_PATH (0x9): The PATH parameter does not conform to the rules in Section 9.3.2.2.

GOAWAY\_TIMEOUT (0x10): The session was closed because the peer took too long to close the session in response to a GOAWAY (Section 9.4) message. See session migration (Section 3.5).

CONTROL\_MESSAGE\_TIMEOUT (0x11): The session was closed because the peer took too long to respond to a control message.

DATA\_STREAM\_TIMEOUT (0x12): The session was closed because the peer took too long to send data expected on an open Data Stream (see Section 10). This includes fields of a stream header or an object header within a data stream. If an endpoint times out waiting for a new object header on an open subgroup stream, it MAY send a STOP\_SENDING on that stream or terminate the subscription.

AUTH\_TOKEN\_CACHE\_OVERFLOW (0x13): The Session limit Section 9.3.2.4 of the size of all registered Authorization tokens has been exceeded.

DUPLICATE\_AUTH\_TOKEN\_ALIAS (0x14): Authorization Token attempted to register an Alias that was in use (see Section 9.2.1.1).

VERSION\_NEGOTIATION\_FAILED (0x15): The client didn't offer a version supported by the server.

MALFORMED\_AUTH\_TOKEN (0x16): Invalid Auth Token serialization during registration (see Section 9.2.1.1).

UNKNOWN\_AUTH\_TOKEN\_ALIAS (0x17): No registered token found for the provided Alias (see Section 9.2.1.1).

EXPIRED\_AUTH\_TOKEN (0x18): Authorization token has expired (Section 9.2.1.1).

INVALID\_AUTHORITY (0x19): The specified AUTHORITY does not correspond to this server or cannot be used in this context.

MALFORMED\_AUTHORITY (0x1A): The AUTHORITY value is syntactically invalid.

An endpoint MAY choose to treat a subscription or request specific error as a session error under certain circumstances, closing the entire session in response to a condition with a single subscription or message. Implementations need to consider the impact on other outstanding subscriptions before making this choice.

### 3.5. Migration

MOQT requires a long-lived and stateful session. However, a service provider needs the ability to shutdown/restart a server without waiting for all sessions to drain naturally, as that can take days for long-form media. MOQT enables proactively draining sessions via the GOAWAY message (Section 9.4).

The server sends a GOAWAY message, signaling the client to establish a new session and migrate any active subscriptions. The GOAWAY message optionally contains a new URI for the new session, otherwise the current URI is reused. The server SHOULD terminate the session with GOAWAY\_TIMEOUT after a sufficient timeout if there are still open subscriptions or fetches on a connection.

When the server is a subscriber, it SHOULD send a GOAWAY message to downstream subscribers prior to any UNSUBSCRIBE messages to upstream publishers.

After the client receives a GOAWAY, it's RECOMMENDED that the client waits until there are no more active subscriptions before closing the session with NO\_ERROR. Ideally this is transparent to the application using MOQT, which involves establishing a new session in the background and migrating active subscriptions and published namespaces. The client can choose to delay closing the session if it expects more OBJECTs to be delivered. The server closes the session with a GOAWAY\_TIMEOUT if the client doesn't close the session quickly enough.

### 3.6. Congestion Control

MOQT does not specify a congestion controller, but there are important attributes to consider when selecting a congestion controller for use with an application built on top of MOQT.

### 3.6.1. Bufferbloat

Traditional AIMD congestion controllers (ex. CUBIC [RFC9438] and Reno [RFC6582]) are prone to Bufferbloat. Bufferbloat occurs when elements along the path build up a substantial queue of packets, commonly more than doubling the round trip time. These queued packets cause head-of-line blocking and latency, even when there is no packet loss.

### 3.6.2. Application-Limited

The average bitrate for latency sensitive content needs to be less than the available bandwidth, otherwise data will be queued and/or dropped. As such, many MOQT applications will typically be limited by the available data to send, and not the congestion controller. Many congestion control algorithms only increase the congestion window or bandwidth estimate if fully utilized. This combination can lead to underestimating the available network bandwidth. As a result, applications might need to periodically ensure the congestion controller is not app-limited for at least a full round trip to ensure the available bandwidth can be measured.

### 3.6.3. Consistent Throughput

Congestion control algorithms are commonly optimized for throughput, not consistency. For example, BBR's PROBE\_RTT state halves the sending rate for more than a round trip in order to obtain an accurate minimum RTT. Similarly, Reno halves its congestion window upon detecting loss. In both cases, the large reduction in sending rate might cause issues with latency sensitive applications.

## 4. Modularity

MOQT defines all messages necessary to implement both simple publishing or subscribing endpoints as well as highly functional Relays. Non-Relay endpoints MAY implement only the subset of functionality required to perform necessary tasks. For example, a limited media player could operate using only SUBSCRIBE related messages. Limited endpoints SHOULD respond to any unsupported messages with the appropriate NOT\_SUPPORTED error code, rather than ignoring them.

Relays MUST implement all MOQT messages defined in this document, as well as processing rules described in Section 8.

## 5. Publishing and Retrieving Tracks

### 5.1. Subscriptions

A subscription can be initiated by either a publisher or a subscriber. A publisher initiates a subscription to a track by sending the PUBLISH message. The subscriber either accepts or rejects the subscription using PUBLISH\_OK or PUBLISH\_ERROR. A subscriber initiates a subscription to a track by sending the SUBSCRIBE message. The publisher either accepts or rejects the subscription using SUBSCRIBE\_OK or SUBSCRIBE\_ERROR. Once either of these sequences is successful, the subscription can be updated by the subscriber using SUBSCRIBE\_UPDATE, terminated by the subscriber using UNSUBSCRIBE, or terminated by the publisher using PUBLISH\_DONE.

All subscriptions have a Forward State which is either 0 or 1. If the Forward State is 0, the publisher does not send objects for the subscription. If the Forward State is 1, the publisher sends objects. The initiator of the subscription sets the initial Forward State in either PUBLISH or SUBSCRIBE. The sender of PUBLISH\_OK can update the Forward State based on its preference. Once the subscription is established, the subscriber can update the Forward State by sending SUBSCRIBE\_UPDATE.

Either endpoint can initiate a subscription to a track without exchanging any prior messages other than SETUP. Relays MUST NOT send any PUBLISH messages without knowing the client is interested in and authorized to receive the content. The communication of intent and authorization can be accomplished by the client sending SUBSCRIBE\_NAMESPACE, or conveyed in other mechanisms out of band.

An endpoint MAY SUBSCRIBE to a Track it is publishing, though only Relays are required to handle such a SUBSCRIBE. Such self-subscriptions are identical to subscriptions initiated by other endpoints, and all published Objects will be forwarded back to the endpoint, subject to priority and congestion response rules.

A publisher MUST send exactly one SUBSCRIBE\_OK or SUBSCRIBE\_ERROR in response to a SUBSCRIBE. It MUST send exactly one FETCH\_OK or FETCH\_ERROR in response to a FETCH. A subscriber MUST send exactly one PUBLISH\_OK or PUBLISH\_ERROR in response to a PUBLISH. The peer SHOULD close the session with a protocol error if it receives more than one.

Publishers MAY start sending Objects on PUBLISH-initiated subscriptions before receiving a PUBLISH\_OK response to reduce latency. Doing so can consume unnecessary resources in cases where the Subscriber rejects the subscription with PUBLISH\_ERROR or sets

Forward State=0 in PUBLISH\_OK. It can also result in the Subscriber dropping Objects if its buffering limits are exceeded (see Section 10.3 and Section 10.4.2).

A subscriber keeps subscription state until it sends UNSUBSCRIBE, or after receipt of a PUBLISH\_DONE or SUBSCRIBE\_ERROR. Note that PUBLISH\_DONE does not usually indicate that state can immediately be destroyed, see Section 9.12.

A subscriber keeps FETCH state until it sends FETCH\_CANCEL, receives FETCH\_ERROR, or receives a FIN or RESET\_STREAM for the FETCH data stream. If the data stream is already open, it MAY send STOP\_SENDING for the data stream along with FETCH\_CANCEL, but MUST send FETCH\_CANCEL.

The Publisher can destroy subscription or fetch state as soon as it has received UNSUBSCRIBE or FETCH\_CANCEL, respectively. It MUST reset any open streams associated with the SUBSCRIBE or FETCH. It can also destroy state after closing the FETCH data stream.

The publisher can immediately delete subscription state after sending PUBLISH\_DONE, but MUST NOT send it until it has closed all related streams. It can destroy all FETCH state after closing the data stream.

A SUBSCRIBE\_ERROR indicates no objects will be delivered, and both endpoints can immediately destroy relevant state. Objects MUST NOT be sent for requests that end with an error.

A FETCH\_ERROR indicates that both endpoints can immediately destroy state. Since a relay can start delivering FETCH Objects from cache before determining the result of the request, some Objects could be received even if the FETCH results in error.

The Parameters in SUBSCRIBE, PUBLISH\_OK and FETCH MUST NOT cause the publisher to alter the payload of the objects it sends, as that would violate the track uniqueness guarantee described in Section 2.5.1.

## 6. Namespace Discovery

Discovery of MOQT servers is always done out-of-band. Namespace discovery can be done in the context of an established MOQT session.

Given sufficient out of band information, it is valid for a subscriber to send a SUBSCRIBE or FETCH message to a publisher (including a relay) without any previous MOQT messages besides SETUP. However, SUBSCRIBE\_NAMESPACE, PUBLISH and PUBLISH\_NAMESPACE messages provide an in-band means of discovery of publishers for a namespace.

The syntax of these messages is described in Section 9.

### 6.1. Subscribing to Namespaces

If the subscriber is aware of a namespace of interest, it can send `SUBSCRIBE_NAMESPACE` to publishers/relays it has established a session with. The recipient of this message will send any relevant `PUBLISH_NAMESPACE`, `PUBLISH_NAMESPACE_DONE` or `PUBLISH` messages for that namespace, or more specific part of that namespace. This includes echoing back `PUBLISH` or `PUBLISH_NAMESPACE` messages to the endpoint that sent them. If an endpoint accepts its own `PUBLISH`, this behaves as self-subscription described in Section 5.1.

A publisher **MUST** send exactly one `SUBSCRIBE_NAMESPACE_OK` or `SUBSCRIBE_NAMESPACE_ERROR` in response to a `SUBSCRIBE_NAMESPACE`. The subscriber **SHOULD** close the session with a protocol error if it detects receiving more than one.

The receiver of a `SUBSCRIBE_NAMESPACE_OK` or `SUBSCRIBE_NAMESPACE_ERROR` ought to forward the result to the application, so the application can decide which other publishers to contact, if any.

An `UNSUBSCRIBE_NAMESPACE` withdraws a previous `SUBSCRIBE_NAMESPACE`. It does not prohibit original publishers from sending further `PUBLISH_NAMESPACE` or `PUBLISH` messages, but relays **MUST NOT** send any further `PUBLISH` messages to a client without knowing the client is interested in and authorized to receive the content.

### 6.2. Publishing Namespaces

A publisher **MAY** send `PUBLISH_NAMESPACE` messages to any subscriber. A `PUBLISH_NAMESPACE` indicates to the subscriber that the publisher has tracks available in that namespace. A subscriber **MAY** send `SUBSCRIBE` or `FETCH` for tracks in a namespace without having received a `PUBLISH_NAMESPACE` for it.

If a publisher is authoritative for a given namespace, or is a relay that has received an authorized `PUBLISH_NAMESPACE` for that namespace from an upstream publisher, it **MUST** send a `PUBLISH_NAMESPACE` to any subscriber that has subscribed via `SUBSCRIBE_NAMESPACE` for that namespace, or a prefix of that namespace. A publisher **MAY** send the `PUBLISH_NAMESPACE` to any other subscriber.

An endpoint **SHOULD** report the reception of a `PUBLISH_NAMESPACE_OK` or `PUBLISH_NAMESPACE_ERROR` to the application to inform the search for additional subscribers for a namespace, or to abandon the attempt to publish under this namespace. This might be especially useful in upload or chat applications. A subscriber **MUST** send exactly one

PUBLISH\_NAMESPACE\_OK or PUBLISH\_NAMESPACE\_ERROR in response to a PUBLISH\_NAMESPACE. The publisher SHOULD close the session with a protocol error if it receives more than one.

A PUBLISH\_NAMESPACE\_DONE message withdraws a previous PUBLISH\_NAMESPACE, although it is not a protocol error for the subscriber to send a SUBSCRIBE or FETCH message for a track in a namespace after receiving an PUBLISH\_NAMESPACE\_DONE.

A subscriber can send PUBLISH\_NAMESPACE\_CANCEL to revoke acceptance of an PUBLISH\_NAMESPACE, for example due to expiration of authorization credentials. The message enables the publisher to PUBLISH\_NAMESPACE again with refreshed authorization, or discard associated state. After receiving an PUBLISH\_NAMESPACE\_CANCEL, the publisher does not send PUBLISH\_NAMESPACE\_DONE.

While PUBLISH\_NAMESPACE indicates to relays how to connect publishers and subscribers, it is not a full-fledged routing protocol and does not protect against loops and other phenomena. In particular, PUBLISH\_NAMESPACE SHOULD NOT be used to find paths through richly connected networks of relays.

A subscriber MAY send a SUBSCRIBE or FETCH for a track to any publisher. If it has accepted a PUBLISH\_NAMESPACE with a namespace that exactly matches the namespace for that track, it SHOULD only request it from the senders of those PUBLISH\_NAMESPACE messages.

## 7. Priorities

MoQ priorities allow a subscriber and original publisher to influence the transmission order of Objects within a session in the presence of congestion.

### 7.1. Definitions

MOQT maintains priorities between different `_schedulable objects_`. A schedulable object in MOQT is either:

1. The first or next Object in a Subgroup that is in response to a subscription.
2. An Object in response to a subscription that belongs to a Track with delivery preference Datagram.
3. An Object in response to a FETCH where that Object is the next Object in the response.

An Object is not schedulable if it is known that no part of it can be written due to underlying transport flow control limits.

A single subgroup or datagram has a single publisher priority. Within a response to SUBSCRIBE, it can be useful to conceptualize this process as scheduling subgroups or datagrams instead of individual objects on them. FETCH responses however can contain objects with different publisher priorities.

A \_priority number\_ is an unsigned integer with a value between 0 and 255. A lower priority number indicates higher priority; the highest priority is 0.

\_Subscriber Priority\_ is a priority number associated with an individual request. It is specified in the SUBSCRIBE or FETCH message, and can be updated via SUBSCRIBE\_UPDATE message. The subscriber priority of an individual schedulable object is the subscriber priority of the request that caused that object to be sent. When subscriber priority is changed, a best effort SHOULD be made to apply the change to all objects that have not been scheduled, but it is implementation dependent what happens to objects that have already been scheduled.

\_Publisher Priority\_ is a priority number associated with an individual schedulable object. It is specified in the header of the respective subgroup or datagram, or in each object in a FETCH response.

\_Group Order\_ is a property of an individual subscription. It can be either 'Ascending' (groups with lower group ID are sent first), or 'Descending' (groups with higher group ID are sent first). The subscriber optionally communicates its group order preference in the SUBSCRIBE message; the publisher's preference is used if the subscriber did not express one (by setting Group Order field to value 0x0). The group order of an existing subscription cannot be changed.

## 7.2. Scheduling Algorithm

When an MOQT publisher has multiple schedulable objects it can choose between, the objects SHOULD be selected as follows:

1. If two objects have different subscriber priorities associated with them, the one with *\*the highest subscriber priority\** is scheduled to be sent first.
2. If two objects have the same subscriber priority, but different publisher priorities, the one with *\*the highest publisher priority\** is scheduled to be sent first.

3. If two objects in response to the same request have the same subscriber and publisher priority, but belong to two different groups of the same track, *\*the group order\** of the associated subscription is used to decide the one that is scheduled to be sent first.
4. If two objects in response to the same request belong to the same group of the same track, the one with *\*the lowest Subgroup ID\** (for tracks with delivery preference Subgroup), or *\*the lowest Object ID\** (for tracks with delivery preference Datagram) is scheduled to be sent first.

The definition of "scheduled to be sent first" in the algorithm is implementation dependent and is constrained by the prioritization interface of the underlying transport. For some implementations, it could mean that the object is serialized and passed to the underlying transport first. Other implementations can control the order packets are initially transmitted.

This algorithm does not provide a well-defined ordering for objects that belong to different subscriptions or FETCH responses, but have the same subscriber and publisher priority. The ordering in those cases is implementation-defined, though the expectation is that all subscriptions will be able to send some data.

Given the critical nature of control messages and their relatively small size, the control stream SHOULD be prioritized higher than all subscribed Objects.

### 7.3. Considerations for Setting Priorities

For downstream subscriptions, relays SHOULD respect the subscriber and original publisher's priorities. Relays can receive subscriptions with conflicting subscriber priorities or Group Order preferences. Relays SHOULD NOT directly use Subscriber Priority or Group Order from incoming subscriptions for upstream subscriptions. Relays' use of these fields for upstream subscriptions can be based on factors specific to it, such as the popularity of the content or policy, or relays can specify the same value for all upstream subscriptions.

MoQ Sessions can span multiple namespaces, and priorities might not be coordinated across namespaces. The subscriber's priority is considered first, so there is a mechanism for a subscriber to fix incompatibilities between different namespaces prioritization schemes. Additionally, it is anticipated that when multiple namespaces are present within a session, the namespaces could be coordinating, possibly part of the same application. In cases when

pooling among namespaces is expected to cause issues, multiple MoQ sessions, either within a single connection or on multiple connections can be used.

Implementations that have a default priority SHOULD set it to a value in the middle of the range (eg: 128) to allow non-default priorities to be set either higher or lower.

## 8. 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.

Relays are endpoints, which means they terminate Transport Sessions in order to have visibility of MoQ Object metadata.

### 8.1. Caching Relays

Relays MAY cache Objects, but are not required to.

A caching relay saves Objects to its cache identified by the Object's Full Track Name, Group ID and Object ID. If multiple objects are received with the same Full Track Name, Group ID and Object ID, Relays MAY ignore subsequently received Objects or MAY use them to update certain cached fields. Implementations that update the cache need to protect against cache poisoning. The only Object fields that can be updated are the following:

1. Object Status can transition from any status to Object Does Not Exist in cases where the object is no longer available. Transitions between Normal, End of Group and End of Track are invalid.
2. Object Header Extensions can be added, removed or updated, subject to the constraints of the specific header extension.

An endpoint that receives a duplicate Object with an invalid Object Status change, or a Forwarding Preference, Subgroup ID, Priority or Payload that differ from a previous version MUST treat the track as Malformed.

Note that due to reordering, an implementation can receive a duplicate Object with a status of Normal, End of Group or End of Track after receiving a previous status of Object Does Not Exist. The endpoint SHOULD NOT cache or forward the duplicate object in this case.

A cache MUST store all properties of an Object defined in Section 10.2.1, with the exception of any extensions (Section 10.2.1.2) that specify otherwise.

## 8.2. Multiple Publishers

A Relay can receive PUBLISH\_NAMESPACE for the same Track Namespace or PUBLISH messages for the same Track from multiple publishers. The following sections explain how Relays maintain subscriptions to all available publishers for a given Track.

There is no specified limit to the number of publishers of a Track Namespace or Track. An implementation can use mechanisms such as PUBLISH\_ERROR, PUBLISH\_NAMESPACE\_ERROR, UNSUBSCRIBE or PUBLISH\_NAMESPACE\_CANCEL if it cannot accept an additional publisher due to implementation constraints. Implementations can consider the establishment or idle time of the session or subscription to determine which publisher to reject or disconnect.

Relays MUST handle Objects for the same Track from multiple publishers and forward them to active matching subscriptions. The Relay SHOULD attempt to deduplicate Objects before forwarding, subject to implementation constraints.

## 8.3. Subscriber Interactions

Subscribers request Tracks by sending a SUBSCRIBE (see Section 9.7) or FETCH (see Section 9.16) control message for each Track of interest. Relays MUST ensure subscribers are authorized to access the content associated with the Track. The authorization information can be part of 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 relay MUST have an established upstream subscription before sending SUBSCRIBE\_OK in response to a downstream SUBSCRIBE. If a relay does not have sufficient information to send a FETCH\_OK immediately in response to a FETCH, it MUST withhold sending FETCH\_OK until it does.

For successful subscriptions, the publisher maintains a list of subscribers for each Track. Relays use the Track Alias (Section 10.1) of an incoming Object to identify its Track and find the active subscribers. Each new Object belonging to the Track is forwarded to each active subscriber, as allowed by the subscription's filter (see Section 9.7), and delivered according to the priority (see Section 7) and delivery timeout (see Section 9.2.1.2).

A relay **MUST NOT** reorder or drop objects received on a multi-object stream when forwarding to subscribers, unless it has application specific information.

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 upstream subscription for the Track. The published content received from the upstream subscription request is cached and shared among the pending subscribers. Because `SUBSCRIBE_UPDATE` only allows narrowing a subscription, relays that aggregate upstream subscriptions can subscribe using the Largest Object filter to avoid churn as downstream subscribers with disparate filters subscribe and unsubscribe from a Track.

A subscriber remains subscribed to a Track at a Relay until it unsubscribes, the upstream publisher terminates the subscription, or the subscription expires (see Section 9.8). A subscription with a filter can reach a state where all possible Objects matching the filter have been delivered to the subscriber. Since tracking this can be prohibitively expensive, Relays are not required or expected to do so.

#### 8.3.1. Graceful Subscriber Relay Switchover

This section describes behavior a subscriber **MAY** implement to allow for a better user experience when a relay sends a `GOAWAY`.

When a subscriber receives the `GOAWAY` message, it starts the process of connecting to a new relay and sending the `SUBSCRIBE` requests for all active subscriptions to the new relay. The new relay will send a response to the subscribes and if they are successful, the subscriptions to the old relay can be stopped with an `UNSUBSCRIBE`.

#### 8.4. Publisher Interactions

There are two ways to publish through a relay:

1. Send a PUBLISH message for a specific Track to the relay. The relay MAY respond with PUBLISH\_OK in Forward State=0 until there are known subscribers for new Tracks.
2. Send a PUBLISH\_NAMESPACE message for a Track Namespace to the relay. This enables the relay to send SUBSCRIBE or FETCH messages to publishers for Tracks in this Namespace in response to requests received from subscribers.

Relays MUST verify that publishers are authorized to publish the set of Tracks whose Track Namespace matches the namespace in a PUBLISH\_NAMESPACE, or the Full Track Name in PUBLISH. The authorization and identification of the publisher depends on the way the relay is managed and is application specific.

When a publisher wants to stop new subscriptions for a published namespace it sends a PUBLISH\_NAMESPACE\_DONE. A subscriber indicates it will no longer subscribe to Tracks in a namespace it previously responded PUBLISH\_NAMESPACE\_OK to by sending a PUBLISH\_NAMESPACE\_CANCEL.

A Relay connects publishers and subscribers by managing sessions based on the Track Namespace or Full Track Name. When a SUBSCRIBE message is sent, its Full Track Name is matched exactly against existing upstream subscriptions.

Namespace Prefix Matching is further used to decide which publishers receive a SUBSCRIBE and which subscribers receive a PUBLISH. In this process, the tuples in the Track Namespace are matched sequentially, requiring an exact match for each field. If the published or subscribed Track Namespace has the same or fewer fields than the Track Namespace in the message, it qualifies as a match.

For example: A SUBSCRIBE message with namespace=(foo, bar) and name=x will match sessions that sent PUBLISH\_NAMESPACE messages with namespace=(foo) or namespace=(foo, bar). It will not match a session with namespace=(foobar).

Relays MUST forward SUBSCRIBE messages to all matching publishers and PUBLISH\_NAMESPACE or PUBLISH messages to all matching subscribers.

When a Relay needs to make an upstream FETCH request, it determines the available publishers using the same matching rules as SUBSCRIBE. When more than one publisher is available, the Relay MAY send the FETCH to any of them.

When a Relay receives an authorized SUBSCRIBE for a Track with one or more active upstream subscriptions, it MUST reply with SUBSCRIBE\_OK. If the SUBSCRIBE has Forward State=1 and the upstream subscriptions are in Forward State=0, the Relay MUST send SUBSCRIBE\_UPDATE with Forward=1 to all publishers. If there are no active upstream subscriptions for the requested Track, the Relay MUST send a SUBSCRIBE request to each publisher that has published the subscription's namespace or prefix thereof. If the SUBSCRIBE has Forward =1, then the Relay MUST use Forward=1 when subscribing upstream.

When a relay receives an incoming PUBLISH message, it MUST send a PUBLISH request to each subscriber that has subscribed (via SUBSCRIBE\_NAMESPACE) to the Track's namespace or prefix thereof.

When a relay receives an authorized PUBLISH\_NAMESPACE for a namespace that matches one or more existing subscriptions to other upstream sessions, it MUST send a SUBSCRIBE to the publisher that sent the PUBLISH\_NAMESPACE for each matching subscription. When it receives an authorized PUBLISH message for a Track that has active subscribers, it MUST respond with PUBLISH\_OK. If at least one downstream subscriber for the Track has Forward State=1, the Relay MUST use Forward State=1 in the reply.

If a Session is closed due to an unknown or invalid control message or Object, the Relay MUST NOT propagate that message or Object to another Session, because it would enable a single Session error to force an unrelated Session, which might be handling other subscriptions, to be closed.

#### 8.4.1. Graceful Publisher Network Switchover

This section describes a behavior that a publisher MAY choose to implement to allow for a better user experience when switching between networks, such as WiFi to Cellular or vice versa.

If the original publisher detects it is likely to need to switch networks, for example because the WiFi signal is getting weaker, and it does not have QUIC connection migration available, it establishes a new session over the new interface and sends PUBLISH\_NAMESPACE and/or PUBLISH messages. The relay will establish subscriptions and the publisher publishes Objects on both sessions. Once the subscriptions have migrated over to the session on the new network, the publisher can stop publishing Objects on the old network. The relay will attempt to deduplicate Objects received on both subscriptions. Ideally, the subscriptions downstream from the relay do not observe this change, and keep receiving the Objects on the same subscription.

#### 8.4.2. Graceful Publisher Relay Switchover

This section describes a behavior that a publisher MAY choose to implement to allow for a better user experience when a relay sends them a GOAWAY.

When a publisher receives a GOAWAY, it starts the process of connecting to a new relay and sends PUBLISH\_NAMESPACE and/or PUBLISH messages, but it does not immediately stop publishing Objects to the old Relay. The new Relay will establish subscriptions and the publisher can start sending new Objects to the new relay instead of the old Relay. Once Objects are going to the new Relay, the published namespaces and subscriptions to the old relay can be withdrawn or terminated.

#### 8.5. Relay Object Handling

MOQT encodes the delivery information via Object headers (Section 10.2). A relay MUST NOT modify Object properties when forwarding, except for Object Extension Headers as specified in Section 10.2.1.2.

A relay MUST treat the object payload as opaque. A relay MUST NOT combine, split, or otherwise modify object payloads. A relay SHOULD prioritize sending Objects based on Section 7.

A publisher SHOULD begin sending incomplete objects when available to avoid incurring additional latency.

### 9. Control Messages

MOQT uses a single bidirectional stream to exchange control messages, as defined in Section 3.3. Every single message on the control stream is formatted as follows:

```
MOQT Control Message {  
    Message Type (i),  
    Message Length (16),  
    Message Payload (...),  
}
```

Figure 3: MOQT Message

The following Message Types are defined:

ID	Messages
0x01	RESERVED (SETUP for version 00)
0x40	RESERVED (CLIENT_SETUP for versions <= 10)
0x41	RESERVED (SERVER_SETUP for versions <= 10)
0x20	CLIENT_SETUP (Section 9.3)
0x21	SERVER_SETUP (Section 9.3)
0x10	GOAWAY (Section 9.4)
0x15	MAX_REQUEST_ID (Section 9.5)
0x1A	REQUESTS_BLOCKED (Section 9.6)
0x3	SUBSCRIBE (Section 9.7)
0x4	SUBSCRIBE_OK (Section 9.8)
0x5	SUBSCRIBE_ERROR (Section 9.9)
0x2	SUBSCRIBE_UPDATE (Section 9.10)
0xA	UNSUBSCRIBE (Section 9.11)
0xB	PUBLISH_DONE (Section 9.12)
0x1D	PUBLISH (Section 9.13)
0x1E	PUBLISH_OK (Section 9.14)
0x1F	PUBLISH_ERROR (Section 9.15)
0x16	FETCH (Section 9.16)
0x18	FETCH_OK (Section 9.17)
0x19	FETCH_ERROR (Section 9.18)
0x17	FETCH_CANCEL (Section 9.19)
0xD	TRACK_STATUS (Section 9.20)
0xE	TRACK_STATUS_OK (Section 9.21)

0xF	TRACK_STATUS_ERROR (Section 9.22)	
0x6	PUBLISH_NAMESPACE (Section 9.23)	
0x7	PUBLISH_NAMESPACE_OK (Section 9.24)	
0x8	PUBLISH_NAMESPACE_ERROR (Section 9.25)	
0x9	PUBLISH_NAMESPACE_DONE (Section 9.26)	
0xC	PUBLISH_NAMESPACE_CANCEL (Section 9.27)	
0x11	SUBSCRIBE_NAMESPACE (Section 9.28)	
0x12	SUBSCRIBE_NAMESPACE_OK (Section 9.29)	
0x13	SUBSCRIBE_NAMESPACE_ERROR (Section 9.30)	
0x14	UNSUBSCRIBE_NAMESPACE (Section 9.31)	

Table 1

An endpoint that receives an unknown message type MUST close the session. Control messages have a length to make parsing easier, but no control messages are intended to be ignored. The length is set to the number of bytes in Message Payload, which is defined by each message type. If the length does not match the length of the Message Payload, the receiver MUST close the session with `PROTOCOL_VIOLATION`.

### 9.1. Request ID

Most MOQT control messages contain a session specific Request ID. The Request ID correlates requests and responses, allows endpoints to update or terminate ongoing requests, and supports the endpoint's ability to limit the concurrency and frequency of requests. There are independent Request IDs for each endpoint. The client's Request ID starts at 0 and are even and the server's Request ID starts at 1 and are odd. The Request ID increments by 2 with each `FETCH`, `SUBSCRIBE`, `SUBSCRIBE_UPDATE`, `SUBSCRIBE_NAMESPACE`, `PUBLISH`, `PUBLISH_NAMESPACE` or `TRACK_STATUS` request. Other messages with a Request ID field reference the Request ID of another message for correlation. If an endpoint receives a Request ID that is not valid for the peer, or a new request with a Request ID that is not expected, it MUST close the session with `INVALID_REQUEST_ID`.

## 9.2. Parameters

Some messages include a Parameters field that encodes optional message elements.

Senders MUST NOT repeat the same parameter type in a message unless the parameter definition explicitly allows multiple instances of that type to be sent in a single message. Receivers SHOULD check that there are no unauthorized duplicate parameters and close the session as a `PROTOCOL_VIOLATION` if found. Receivers MUST allow duplicates of unknown parameters.

Receivers ignore unrecognized parameters.

The number of parameters in a message is not specifically limited, but the total length of a control message is limited to  $2^{16}-1$  bytes.

Parameters are serialized as Key-Value-Pairs Figure 2.

Setup message parameters use a namespace that is constant across all MoQ Transport versions. All other messages use a version-specific namespace. For example, the integer '1' can refer to different parameters for Setup messages and for all other message types. `SETUP` message parameter types are defined in Section 9.3.2. Version-specific parameter types are defined in Section 9.2.1.

### 9.2.1. Version Specific Parameters

Each version-specific parameter definition indicates the message types in which it can appear. If it appears in some other type of message, it MUST be ignored. Note that since Setup parameters use a separate namespace, it is impossible for these parameters to appear in Setup messages.

#### 9.2.1.1. AUTHORIZATION TOKEN

The `AUTHORIZATION TOKEN` parameter (Parameter Type 0x03) MAY appear in a `CLIENT_SETUP`, `SERVER_SETUP`, `PUBLISH`, `SUBSCRIBE`, `SUBSCRIBE_UPDATE`, `SUBSCRIBE_NAMESPACE`, `PUBLISH_NAMESPACE`, `TRACK_STATUS` or `FETCH` message. This parameter conveys information to authorize the sender to perform the operation carrying the parameter.

The `AUTHORIZATION TOKEN` parameter MAY be repeated within a message.

The parameter value is a Token structure containing an optional Session-specific Alias. The Alias allows the sender to reference a previously transmitted Token Type and Token Value in future messages. The Token structure is serialized as follows:

```
Token {  
  Alias Type (i),  
  [Token Alias (i),]  
  [Token Type (i),]  
  [Token Value (..)]  
}
```

Figure 4: Token structure

- \* Alias Type - an integer defining both the serialization and the processing behavior of the receiver. This Alias type has the following code points:

Code	Name	Serialization and behavior
0x0	DELETE	There is an Alias but no Type or Value. This Alias and the Token Value it was previously associated with MUST be retired. Retiring removes them from the pool of actively registered tokens.
0x1	REGISTER	There is an Alias, a Type and a Value. This Alias MUST be associated with the Token Value for the duration of the Session or it is deleted. This action is termed "registering" the Token.
0x2	USE_ALIAS	There is an Alias but no Type or Value. Use the Token Type and Value previously registered with this Alias.
0x3	USE_VALUE	There is no Alias and there is a Type and Value. Use the Token Value as provided. The Token Value may be discarded after processing.

Table 2

- \* Token Alias - a Session-specific integer identifier that references a Token Value. There are separate Alias spaces for the client and server (e.g.: they can each register Alias=1). Once a Token Alias has been registered, it cannot be re-registered by the same sender in the Session without first being deleted. Use of the Token Alias is optional.
- \* Token Type - a numeric identifier for the type of Token payload being transmitted. This type is defined by the IANA table "MOQT Auth Token Type" (see Section 13). Type 0 is reserved to indicate that the type is not defined in the table and is negotiated out-of-band between client and receiver.
- \* Token Value - the payload of the Token. The contents and serialization of this payload are defined by the Token Type.

If the Token structure cannot be decoded, the receiver MUST close the Session with Key-Value Formatting error. The receiver of a message attempting to register an Alias which is already registered MUST close the Session with `DUPLICATE_AUTH_TOKEN_ALIAS`. The receiver of a message referencing an Alias that is not currently registered MUST reject the message with `UNKNOWN_AUTH_TOKEN_ALIAS`.

The receiver of a message containing a well-formed Token structure but otherwise invalid `AUTHORIZATION TOKEN` parameter MUST reject that message with an `MALFORMED_AUTH_TOKEN` error.

The receiver of a message carrying an `AUTHORIZATION TOKEN` with Alias Type `REGISTER` that does not result in a Session error MUST register the Token Alias, in the token cache, even if the message fails for other reasons, including `Unauthorized`. This allows senders to pipeline messages that refer to previously registered tokens without potentially terminating the entire Session. A receiver MAY store an error code (eg: `UNAUTHORIZED` or `MALFORMED_AUTH_TOKEN`) in place of the Token Type and Token Alias if any future message referencing the Token Alias will result in that error. The size of a registered cache entry includes the length of the Token Value, regardless of whether it is stored.

If a receiver detects that an authorization token has expired, it MUST retain the registered Alias until it is deleted by the sender, though it MAY discard other state associated with the token that is no longer needed. Expiration does not affect the size occupied by a token in the token cache. Any message that references the token with Alias Type `USE_ALIAS` fails with `EXPIRED_AUTH_TOKEN`.

Using an Alias to refer to a previously registered Token Type and Value is for efficiency only and has the same effect as if the Token Type and Value was included directly. Retiring an Alias that was previously used to authorize a message has no retroactive effect on the original authorization, nor does it prevent that same Token Type and Value from being re-registered.

Senders of tokens SHOULD only register tokens which they intend to re-use during the Session and SHOULD retire previously registered tokens once their utility has passed.

By registering a Token, the sender is requiring the receiver to store the Token Alias and Token Value until they are deleted, or the Session ends. The receiver can protect its resources by sending a SETUP parameter defining the MAX\_AUTH\_TOKEN\_CACHE\_SIZE limit (see Section 9.3.2.4) it is willing to accept. If a registration is attempted which would cause this limit to be exceeded, the receiver MUST terminate the Session with a AUTH\_TOKEN\_CACHE\_OVERFLOW error.

#### 9.2.1.2. DELIVERY TIMEOUT Parameter

The DELIVERY TIMEOUT parameter (Parameter Type 0x02) MAY appear in a TRACK\_STATUS, TRACK\_STATUS\_OK, PUBLISH, PUBLISH\_OK, SUBSCRIBE, SUBSCRIBE\_OK, or SUBSCRIBE\_UDPATE message. It is the duration in milliseconds the relay SHOULD continue to attempt forwarding Objects after they have been received. The start time for the timeout is based on when the Object Headers are received, and does not depend upon the forwarding preference. There is no explicit signal that an Object was not sent because the delivery timeout was exceeded.

If both the subscriber and publisher specify the parameter, they use the min of the two values for the subscription. The publisher SHOULD always specify the value received from an upstream subscription when there is one, and nothing otherwise.

Publishers can, at their discretion, discontinue forwarding Objects earlier than the negotiated DELIVERY TIMEOUT, subject to stream closure and ordering constraints described in Section 10.4.3. However, if neither the subscriber nor publisher specifies DELIVERY TIMEOUT, all Objects in the track matching the subscription filter are delivered as indicated by their Group Order and Priority. If a subscriber fails to consume Objects at a sufficient rate, causing the publisher to exceed its resource limits, the publisher MAY terminate the subscription with error TOO\_FAR\_BEHIND.

If an object in a subgroup exceeds the delivery timeout, the publisher MUST reset the underlying transport stream (see Section 10.4.3).

When sent by a subscriber, this parameter is intended to be specific to a subscription, so it SHOULD NOT be forwarded upstream by a relay that intends to serve multiple subscriptions for the same track.

Publishers SHOULD consider whether the entire Object can likely be successfully delivered within the timeout period before sending any data for that Object, taking into account priorities, congestion control, and any other relevant information.

#### 9.2.1.3. MAX\_CACHE\_DURATION Parameter

The MAX\_CACHE\_DURATION parameter (Parameter Type 0x04) MAY appear in a PUBLISH, SUBSCRIBE\_OK, FETCH\_OK or TRACK\_STATUS\_OK message. It is an integer expressing the number of milliseconds an object can be served from a cache. If present, the relay MUST NOT start forwarding any individual Object received through this subscription or fetch after the specified number of milliseconds has elapsed since the beginning of the Object was received. This means Objects earlier in a multi-object stream will expire earlier than Objects later in the stream. Once Objects have expired from cache, their state becomes unknown, and a relay that handles a downstream request that includes those Objects re-requests them.

#### 9.3. CLIENT\_SETUP and SERVER\_SETUP

The CLIENT\_SETUP and SERVER\_SETUP messages are the first messages exchanged by the client and the server; they allow the endpoints 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, endpoints MUST ignore unknown setup parameters. TODO: describe GREASE for those.

The wire format of the Setup messages are as follows:

```
CLIENT_SETUP Message {
  Type (i) = 0x20,
  Length (16),
  Number of Supported Versions (i),
  Supported Versions (i) ...,
  Number of Parameters (i),
  Setup Parameters (...) ...,
}

SERVER_SETUP Message {
  Type (i) = 0x21,
  Length (16),
  Selected Version (i),
  Number of Parameters (i),
  Setup Parameters (...) ...,
}
```

Figure 5: MOQT Setup Messages

The available versions and Setup parameters are detailed in the next sections.

#### 9.3.1. Versions

MOQT versions are a 32-bit unsigned integer, encoded as a varint. This version of the specification is identified by the number 0x00000001. Versions with the most significant 16 bits of the version number cleared are reserved for use in future IETF consensus documents.

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 session with VERSION\_NEGOTIATION\_FAILED.

[[RFC editor: please remove the remainder of this section before publication.]]

The version number for the final version of this specification (0x00000001), is reserved for the version of the protocol that is published as an RFC. Version numbers used to identify IETF drafts are created by adding the draft number to 0xff000000. For example, draft-ietf-moq-transport-13 would be identified as 0xff00000D.

#### 9.3.2. Setup Parameters

#### 9.3.2.1. AUTHORITY

The AUTHORITY parameter (Parameter Type 0x05) allows the client to specify the authority component of the MoQ URI when using native QUIC ([QUIC]). It MUST NOT be used by the server, or when WebTransport is used. When an AUTHORITY parameter is received from a server, or when an AUTHORITY parameter is received while WebTransport is used, or when an AUTHORITY parameter is received by a server but the server does not support the specified authority, the session MUST be closed with Invalid Authority.

The AUTHORITY parameter follows the URI formatting rules [RFC3986]. When connecting to a server using a URI with the "moqt" scheme, the client MUST set the AUTHORITY parameter to the authority portion of the URI. If an AUTHORITY parameter does not conform to these rules, the session MUST be closed with Malformed Authority.

#### 9.3.2.2. PATH

The PATH parameter (Parameter Type 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. When a PATH parameter is received from a server, or when a PATH parameter is received while WebTransport is used, or when a PATH parameter is received by a server but the server does not support the specified path, the session MUST be closed with Invalid Path.

The PATH parameter follows the URI formatting rules [RFC3986]. When connecting to a server using a URI with the "moqt" 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. If a PATH does not conform to these rules, the session MUST be closed with Malformed Path.

#### 9.3.2.3. MAX\_REQUEST\_ID

The MAX\_REQUEST\_ID parameter (Parameter Type 0x02) communicates an initial value for the Maximum Request ID to the receiving endpoint. The default value is 0, so if not specified, the peer MUST NOT send requests.

#### 9.3.2.4. MAX\_AUTH\_TOKEN\_CACHE\_SIZE

The MAX\_AUTH\_TOKEN\_CACHE\_SIZE parameter (Parameter Type 0x04) communicates the maximum size in bytes of all actively registered Authorization tokens that the server is willing to store per Session. This parameter is optional. The default value is 0 which prohibits the use of token Aliases.

The token size is calculated as 16 bytes + the size of the Token Value field (see Figure 4). The total size as restricted by the MAX\_AUTH\_TOKEN\_CACHE\_SIZE parameter is calculated as the sum of the token sizes for all registered tokens (Alias Type value of 0x01) minus the sum of the token sizes for all deregistered tokens (Alias Type value of 0x00), since Session initiation.

#### 9.3.2.5. AUTHORIZATION TOKEN

See Section 9.2.1.1. The endpoint can specify one or more tokens in CLIENT\_SETUP or SERVER\_SETUP that the peer can use to authorize MOQT session establishment.

If a server receives an AUTHORIZATION TOKEN parameter in CLIENT\_SETUP with Alias Type REGISTER\_TOKEN that exceeds its MAX\_AUTH\_TOKEN\_CACHE\_SIZE, it MUST NOT fail the session with AUTH\_TOKEN\_CACHE\_OVERFLOW. Instead, it MUST treat the parameter as Alias Type USE\_VALUE. A client MUST handle registration failures of this kind by purging any Token Aliases that failed to register based on the MAX\_AUTH\_TOKEN\_CACHE\_SIZE parameter in SERVER\_SETUP (or the default value of 0).

#### 9.3.2.6. MOQT IMPLEMENTATION

The MOQT\_IMPLEMENTATION parameter (Parameter Type 0x05) identifies the name and version of the sender's MOQT implementation. This SHOULD be a UTF-8 encoded string [RFC3629], though the message does not carry information, such as language tags, that would aid comprehension by any entity other than the one that created the text.

#### 9.4. GOAWAY

An endpoint sends a GOAWAY message to inform the peer it intends to close the session soon. Servers can use GOAWAY to initiate session migration (Section 3.5) with an optional URI.

The GOAWAY message does not impact subscription state. A subscriber SHOULD individually UNSUBSCRIBE for each existing subscription, while a publisher MAY reject new requests after sending a GOAWAY.

Upon receiving a GOAWAY, an endpoint SHOULD NOT initiate new requests to the peer including SUBSCRIBE, PUBLISH, FETCH, PUBLISH\_NAMESPACE, SUBSCRIBE\_NAMESPACE and TRACK\_STATUS.

The endpoint MUST terminate the session with a PROTOCOL\_VIOLATION (Section 3.4) if it receives multiple GOAWAY messages.

```
GOAWAY Message {  
    Type (i) = 0x10,  
    Length (16),  
    New Session URI Length (i),  
    New Session URI (...),  
}
```

Figure 6: MOQT GOAWAY Message

- \* New Session URI: When received by a client, indicates where the client can connect to continue this session. The client MUST use this URI for the new session if provided. If the URI is zero bytes long, the current URI is reused instead. The new session URI SHOULD use the same scheme as the current URI to ensure compatibility. The maximum length of the New Session URI is 8,192 bytes. If an endpoint receives a length exceeding the maximum, it MUST close the session with a PROTOCOL\_VIOLATION.

If a server receives a GOAWAY with a non-zero New Session URI Length it MUST terminate the session with a PROTOCOL\_VIOLATION.

#### 9.5. MAX\_REQUEST\_ID

An endpoint sends a MAX\_REQUEST\_ID message to increase the number of requests the peer can send within a session.

The Maximum Request ID MUST only increase within a session, and receipt of a MAX\_REQUEST\_ID message with an equal or smaller Request ID value is a PROTOCOL\_VIOLATION.

```
MAX_REQUEST_ID Message {  
    Type (i) = 0x15,  
    Length (16),  
    Request ID (i),  
}
```

Figure 7: MOQT MAX\_REQUEST\_ID Message

- \* Request ID: The new Maximum Request ID for the session plus 1. If a Request ID equal to or larger than this is received by the endpoint that sent the MAX\_REQUEST\_ID in any request message

(PUBLISH\_NAMESPACE, FETCH, SUBSCRIBE, SUBSCRIBE\_NAMESPACE, SUBSCRIBE\_UDPATE or TRACK\_STATUS), the endpoint MUST close the session with an error of TOO\_MANY\_REQUESTS.

MAX\_REQUEST\_ID is similar to MAX\_STREAMS in ([RFC9000], Section 4.6), and similar considerations apply when deciding how often to send MAX\_REQUEST\_ID. For example, implementations might choose to increase MAX\_REQUEST\_ID as subscriptions are closed to keep the number of available subscriptions roughly consistent.

#### 9.6. REQUESTS\_BLOCKED

The REQUESTS\_BLOCKED message is sent when an endpoint would like to send a new request, but cannot because the Request ID would exceed the Maximum Request ID value sent by the peer. The endpoint SHOULD send only one REQUESTS\_BLOCKED for a given Maximum Request ID.

An endpoint MAY send a MAX\_REQUEST\_ID upon receipt of REQUESTS\_BLOCKED, but it MUST NOT rely on REQUESTS\_BLOCKED to trigger sending a MAX\_REQUEST\_ID, because sending REQUESTS\_BLOCKED is not required.

```
REQUESTS_BLOCKED Message {  
  Type (i) = 0x1A,  
  Length (16),  
  Maximum Request ID (i),  
}
```

Figure 8: MOQT REQUESTS\_BLOCKED Message

- \* Maximum Request ID: The Maximum Request ID for the session on which the endpoint is blocked. More on Request ID in Section 9.1.

#### 9.7. SUBSCRIBE

A subscription causes the publisher to send newly published objects for a track. A subscriber MUST NOT make multiple active subscriptions for a track within a single session and publishers SHOULD treat this as a protocol violation.

##### \*Filter Types\*

The subscriber specifies a filter on the subscription to allow the publisher to identify which objects need to be delivered.

All filters have a Start Location and an optional End Group. Only objects published or received via a subscription having Locations greater than or equal to Start and strictly less than or equal to the End Group (when present) pass the filter.

The Largest Object is defined to be the object with the largest Location (Section 1.4.1) in the track from the perspective of the endpoint processing the SUBSCRIBE message. Largest Object updates when the first byte of an Object with a larger Location than the previous value is published or received through a subscription.

There are 4 types of filters:

Largest Object (0x2): The filter Start Location is {Largest Object.Group, Largest Object.Object + 1} and Largest Object is communicated in SUBSCRIBE\_OK. If no content has been delivered yet, the filter Start Location is {0, 0}. There is no End Group - the subscription is open ended. Note that due to network reordering or prioritization, relays can receive Objects with Locations smaller than Largest Object after the SUBSCRIBE is processed, but these Objects do not pass the Largest Object filter.

Next Group Start (0x1): The filter Start Location is {Largest Object.Group + 1, 0} and Largest Object is communicated in SUBSCRIBE\_OK. If no content has been delivered yet, the filter Start Location is {0, 0}. There is no End Group - the subscription is open ended. For scenarios where the subscriber intends to start from more than one group in the future, it can use an AbsoluteStart filter instead.

AbsoluteStart (0x3): The filter Start Location is specified explicitly in the SUBSCRIBE message. The Start specified in the SUBSCRIBE message MAY be less than the Largest Object observed at the publisher. There is no End Group - the subscription is open ended. To receive all Objects that are published or are received after this subscription is processed, a subscriber can use an AbsoluteStart filter with Start = {0, 0}.

AbsoluteRange (0x4): The filter Start Location and End Group are specified explicitly in the SUBSCRIBE message. The Start specified in the SUBSCRIBE message MAY be less than the Largest Object observed at the publisher. If the specified End Group is the same group specified in Start, the remainder of that Group passes the filter. End Group MUST specify the same or a larger Group than specified in Start.

An endpoint that receives a filter type other than the above MUST be close the session with `PROTOCOL_VIOLATION`.

Subscribe only delivers newly published or received Objects. Objects from the past are retrieved using FETCH (Section 9.16).

A Subscription can also request a publisher to not forward Objects for a given track by setting the Forward field to 0. This allows the publisher or relay to prepare to serve the subscription in advance, reducing the time to receive objects in the future. Relays SHOULD set the Forward flag to 1 if a new subscription needs to be sent upstream, regardless of the value of the Forward field from the downstream subscription. Subscriptions that are not forwarded consume resources from the publisher, so a publisher might deprioritize, reject, or close those subscriptions to ensure other subscriptions can be delivered. Control messages, such as SUBSCRIBE\_DONE (Section 9.12) are still sent.

The format of SUBSCRIBE is as follows:

```
SUBSCRIBE Message {
  Type (i) = 0x3,
  Length (16),
  Request ID (i),
  Track Namespace (tuple),
  Track Name Length (i),
  Track Name (...),
  Subscriber Priority (8),
  Group Order (8),
  Forward (8),
  Filter Type (i),
  [Start Location (Location)],
  [End Group (i)],
  Number of Parameters (i),
  Parameters (...) ...
}
```

Figure 9: MOQT SUBSCRIBE Message

- \* Request ID: See Section 9.1.
- \* Track Namespace: Identifies the namespace of the track as defined in (Section 2.4.1).
- \* Track Name: Identifies the track name as defined in (Section 2.4.1).
- \* Subscriber Priority: Specifies the priority of a subscription relative to other subscriptions in the same session. Lower numbers get higher priority. See Section 7.

- \* **Group Order:** Allows the subscriber to request Objects be delivered in Ascending (0x1) or Descending (0x2) order by group. See Section 7. A value of 0x0 indicates the original publisher's Group Order SHOULD be used. Values larger than 0x2 are a protocol error.
- \* **Forward:** If 1, Objects matching the subscription are forwarded to the subscriber. If 0, Objects are not forwarded to the subscriber. Any other value is a protocol error and MUST terminate the session with a `PROTOCOL_VIOLATION` (Section 3.4).
- \* **Filter Type:** Identifies the type of filter, which also indicates whether the Start and End Group fields will be present.
- \* **Start Location:** The starting location for this subscriptions. Only present for "AbsoluteStart" and "AbsoluteRange" filter types.
- \* **End Group:** The end Group ID. Only present for the "AbsoluteRange" filter type.
- \* **Parameters:** The parameters are defined in Section 9.2.1.

On successful subscription, the publisher MUST reply with a `SUBSCRIBE_OK`, allowing the subscriber to determine the start group/object when not explicitly specified and the publisher SHOULD start delivering objects.

If a publisher cannot satisfy the requested start or end or if the end has already been published it SHOULD send a `SUBSCRIBE_ERROR` with code `INVALID_RANGE`. A publisher MUST NOT send objects from outside the requested start and end.

#### 9.8. SUBSCRIBE\_OK

A publisher sends a `SUBSCRIBE_OK` control message for successful subscriptions.

```
SUBSCRIBE_OK Message {  
  Type (i) = 0x4,  
  Length (16),  
  Request ID (i),  
  Track Alias (i),  
  Expires (i),  
  Group Order (8),  
  Content Exists (8),  
  [Largest Location (Location)],  
  Number of Parameters (i),  
  Parameters (...) ...  
}
```

Figure 10: MOQT SUBSCRIBE\_OK Message

- \* Request ID: The Request ID of the SUBSCRIBE this message is replying to Section 9.7.
- \* Track Alias: The identifier used for this track in Subgroups or Datagrams (see Section 10.1). The same Track Alias MUST NOT be used to refer to two different Tracks simultaneously. If a subscriber receives a SUBSCRIBE\_OK that uses the same Track Alias as a different track with an active subscription, it MUST close the session with error `DUPLICATE_TRACK_ALIAS`.
- \* Expires: Time in milliseconds after which the subscription is no longer valid. A value of 0 indicates that the subscription does not expire or expires at an unknown time. Expires is advisory and a subscription can end prior to the expiry time or last longer.
- \* Group Order: Indicates the subscription will be delivered in Ascending (0x1) or Descending (0x2) order by group. See Section 7. Values of 0x0 and those larger than 0x2 are a protocol error.
- \* Content Exists: 1 if an object has been published on this track, 0 if not. If 0, then the Largest Group ID and Largest Object ID fields will not be present. Any other value is a protocol error and MUST terminate the session with a `PROTOCOL_VIOLATION` (Section 3.4).
- \* Largest Location: The location of the largest object available for this track. This field is only present if Content Exists has a value of 1.
- \* Parameters: The parameters are defined in Section 9.2.1.

### 9.9. SUBSCRIBE\_ERROR

A publisher sends a SUBSCRIBE\_ERROR control message in response to a failed SUBSCRIBE.

```
SUBSCRIBE_ERROR Message {  
  Type (i) = 0x5,  
  Length (16),  
  Request ID (i),  
  Error Code (i),  
  Error Reason (Reason Phrase),  
}
```

Figure 11: MOQT SUBSCRIBE\_ERROR Message

- \* Request ID: The Request ID of the SUBSCRIBE this message is replying to Section 9.7.
- \* Error Code: Identifies an integer error code for subscription failure.
- \* Error Reason: Provides the reason for subscription error. See Section 1.4.3.

The application SHOULD use a relevant error code in SUBSCRIBE\_ERROR, as defined below:

INTERNAL\_ERROR (0x0): An implementation specific or generic error occurred.

UNAUTHORIZED (0x1): The subscriber is not authorized to subscribe to the given track.

TIMEOUT (0x2): The subscription could not be completed before an implementation specific timeout. For example, a relay could not establish an upstream subscription within the timeout.

NOT\_SUPPORTED (0x3): The endpoint does not support the SUBSCRIBE method.

TRACK\_DOES\_NOT\_EXIST (0x4): The requested track is not available at the publisher.

INVALID\_RANGE (0x5): The end of the SUBSCRIBE range is earlier than the beginning, or the end of the range has already been published.

MALFORMED\_AUTH\_TOKEN (0x10): Invalid Auth Token serialization during registration (see Section 9.2.1.1).

EXPIRED\_AUTH\_TOKEN (0x12): Authorization token has expired  
(Section 9.2.1.1).

#### 9.10. SUBSCRIBE\_UPDATE

A subscriber sends a SUBSCRIBE\_UPDATE to a publisher to modify an existing subscription. Subscriptions can only be narrowed, not widened, as an attempt to widen could fail. If Objects with Locations smaller than the current subscription's Start Location are required, FETCH can be used to retrieve them. The Start Location MUST NOT decrease and the End Group MUST NOT increase. A publisher MUST terminate the session with a PROTOCOL\_VIOLATION if the SUBSCRIBE\_UPDATE violates these rules or if the subscriber specifies a request ID that has not existed within the Session.

When a subscriber narrows their subscription, it might still receive objects outside the new range if the publisher sent them before the update was processed.

There is no control message in response to a SUBSCRIBE\_UPDATE, because it is expected that it will always succeed and the worst outcome is that it is not processed promptly and some extra objects from the existing subscription are delivered.

Unlike a new subscription, SUBSCRIBE\_UPDATE can not cause an Object to be delivered multiple times. Like SUBSCRIBE, End Group MUST be greater than or equal to the Group specified in Start.

If a parameter included in SUBSCRIBE is not present in SUBSCRIBE\_UPDATE, its value remains unchanged. There is no mechanism to remove a parameter from a subscription.

The format of SUBSCRIBE\_UPDATE is as follows:

```
SUBSCRIBE_UPDATE Message {
  Type (i) = 0x2,
  Length (16),
  Request ID (i),
  Subscription Request ID (i),
  Start Location (Location),
  End Group (i),
  Subscriber Priority (8),
  Forward (8),
  Number of Parameters (i),
  Parameters (...) ...
}
```

Figure 12: MOQT SUBSCRIBE\_UPDATE Message

- \* Request ID: See Section 9.1.
- \* Subscription Request ID: The Request ID of the SUBSCRIBE (Section 9.7) this message is updating. This MUST match an existing Request ID.
- \* Start Location : The starting location.
- \* End Group: The end Group ID, plus 1. A value of 0 means the subscription is open-ended.
- \* Subscriber Priority: Specifies the priority of a subscription relative to other subscriptions in the same session. Lower numbers get higher priority. See Section 7.
- \* Forward: If 1, Objects matching the subscription are forwarded to the subscriber. If 0, Objects are not forwarded to the subscriber. Any other value is a protocol error and MUST terminate the session with a PROTOCOL\_VIOLATION (Section 3.4).
- \* Parameters: The parameters are defined in Section 9.2.1.

#### 9.11. UNSUBSCRIBE

A Subscriber issues an UNSUBSCRIBE message to a Publisher indicating it is no longer interested in receiving the specified Track, indicating that the Publisher stop sending Objects as soon as possible.

The format of UNSUBSCRIBE is as follows:

```
UNSUBSCRIBE Message {  
    Type (i) = 0xA,  
    Length (16),  
    Request ID (i)  
}
```

Figure 13: MOQT UNSUBSCRIBE Message

- \* Request ID: The Request ID of the subscription that is being terminated. See Section 9.7.

## 9.12. PUBLISH\_DONE

A publisher sends a PUBLISH\_DONE message to indicate it is done publishing Objects for that subscription. The Status Code indicates why the subscription ended, and whether it was an error. Because PUBLISH\_DONE is sent on the control stream, it is likely to arrive at the receiver before late-arriving objects, and often even late-opening streams. However, the receiver uses it as an indication that it should receive any late-opening streams in a relatively short time.

Note that some objects in the subscribed track might never be delivered, because a stream was reset, or never opened in the first place, due to the delivery timeout.

A sender MUST NOT send PUBLISH\_DONE until it has closed all streams it will ever open, and has no further datagrams to send, for a subscription. After sending PUBLISH\_DONE, the sender can immediately destroy subscription state, although stream state can persist until delivery completes. The sender might persist subscription state to enforce the delivery timeout by resetting streams on which it has already sent FIN, only deleting it when all such streams have received ACK of the FIN.

A sender MUST NOT destroy subscription state until it sends PUBLISH\_DONE, though it can choose to stop sending objects (and thus send PUBLISH\_DONE) for any reason.

A subscriber that receives PUBLISH\_DONE SHOULD set a timer of at least its delivery timeout in case some objects are still inbound due to prioritization or packet loss. The subscriber MAY dispense with a timer if it sent UNSUBSCRIBE or is otherwise no longer interested in objects from the track. Once the timer has expired, the receiver destroys subscription state once all open streams for the subscription have closed. A subscriber MAY discard subscription state earlier, at the cost of potentially not delivering some late objects to the application. The subscriber SHOULD send STOP\_SENDING on all streams related to the subscription when it deletes subscription state.

The format of PUBLISH\_DONE is as follows:

```
PUBLISH_DONE Message {  
    Type (i) = 0xB,  
    Length (16),  
    Request ID (i),  
    Status Code (i),  
    Stream Count (i),  
    Error Reason (Reason Phrase)  
}
```

Figure 14: MOQT PUBLISH\_DONE Message

- \* Request ID: The Request ID of the subscription that is being terminated. See Section 9.7.
- \* Status Code: An integer status code indicating why the subscription ended.
- \* Stream Count: An integer indicating the number of data streams the publisher opened for this subscription. This helps the subscriber know if it has received all of the data published in this subscription by comparing the number of streams received. The subscriber can immediately remove all subscription state once the same number of streams have been processed. If the track had Forwarding Preference = Datagram, the publisher MUST set Stream Count to 0. If the publisher is unable to set Stream Count to the exact number of streams opened for the subscription, it MUST set Stream Count to  $2^{62} - 1$ . Subscribers SHOULD use a timeout or other mechanism to remove subscription state in case the publisher set an incorrect value, reset a stream before the SUBGROUP\_HEADER, or set the maximum value. If a subscriber receives more streams for a subscription than specified in Stream Count, it MAY close the session with a PROTOCOL\_VIOLATION.
- \* Error Reason: Provides the reason for subscription error. See Section 1.4.3.

The application SHOULD use a relevant status code in PUBLISH\_DONE, as defined below:

INTERNAL\_ERROR (0x0): An implementation specific or generic error occurred.

UNAUTHORIZED (0x1): The subscriber is no longer authorized to subscribe to the given track.

TRACK\_ENDED (0x2): The track is no longer being published.

SUBSCRIPTION\_ENDED (0x3): The publisher reached the end of an

associated Subscribe filter.

GOING\_AWAY (0x4): The subscriber or publisher issued a GOAWAY message.

EXPIRED (0x5): The publisher reached the timeout specified in SUBSCRIBE\_OK.

TOO\_FAR\_BEHIND (0x6): The publisher's queue of objects to be sent to the given subscriber exceeds its implementation defined limit.

MALFORMED\_TRACK (0x7): A relay publisher detected the track was malformed (see Section 2.5).

### 9.13. PUBLISH

The publisher sends the PUBLISH control message to initiate a subscription to a track. The receiver verifies the publisher is authorized to publish this track.

```
PUBLISH Message {
  Type (i) = 0x1D,
  Length (i),
  Request ID (i),
  Track Namespace (tuple),
  Track Name Length (i),
  Track Name (...),
  Track Alias (i),
  Group Order (8),
  Content Exists (8),
  [Largest Location (Location),]
  Forward (8),
  Number of Parameters (i),
  Parameters (...) ...,
}
```

Figure 15: MOQT PUBLISH Message

- \* Request ID: See Section 9.1.
- \* Track Namespace: Identifies a track's namespace as defined in (Section 2.4.1)
- \* Track Name: Identifies the track name as defined in (Section 2.4.1).

- \* **Track Alias:** The identifier used for this track in Subgroups or Datagrams (see Section 10.1). The same Track Alias **MUST NOT** be used to refer to two different Tracks simultaneously. If a subscriber receives a PUBLISH that uses the same Track Alias as a different track with an active subscription, it **MUST** close the session with error `DUPLICATE_TRACK_ALIAS`.
- \* **Group Order:** Indicates the subscription will be delivered in Ascending (0x1) or Descending (0x2) order by group. See Section 7. Values of 0x0 and those larger than 0x2 are a protocol error.
- \* **Content Exists:** 1 if an object has been published on this track, 0 if not. If 0, then the Largest Group ID and Largest Object ID fields will not be present. Any other value is a protocol error and **MUST** terminate the session with a `PROTOCOL_VIOLATION` (Section 3.4).
- \* **Largest Location:** The location of the largest object available for this track.
- \* **Forward:** The forward mode for this subscription. Any value other than 0 or 1 is a `PROTOCOL_VIOLATION`. 0 indicates the publisher will not transmit any objects until the subscriber sets the Forward State to 1. 1 indicates the publisher will start transmitting objects immediately, even before `PUBLISH_OK`.
- \* **Parameters:** The parameters are defined in Section 9.2.1.

A subscriber receiving a PUBLISH for a Track it does not wish to receive **SHOULD** send `PUBLISH_ERROR` with error code `UNINTERESTED`, and abandon reading any publisher initiated streams associated with that subscription using a `STOP_SENDING` frame.

#### 9.14. PUBLISH\_OK

The subscriber sends a `PUBLISH_OK` control message to acknowledge the successful authorization and acceptance of a PUBLISH message, and establish a subscription.

```
PUBLISH_OK Message {
  Type (i) = 0x1E,
  Length (i),
  Request ID (i),
  Forward (8),
  Subscriber Priority (8),
  Group Order (8),
  Filter Type (i),
  [Start Location (Location)],
  [End Group (i)],
  Number of Parameters (i),
  Parameters (...) ...,
}
```

Figure 16: MOQT PUBLISH\_OK Message

- \* Request ID: The Request ID of the PUBLISH this message is replying to Section 9.13.
- \* Forward: The Forward State for this subscription, either 0 (don't forward) or 1 (forward).
- \* Subscriber Priority: The Subscriber Priority for this subscription.
- \* Group Order: Indicates the subscription will be delivered in Ascending (0x1) or Descending (0x2) order by group. See Section 7. Values of 0x0 and those larger than 0x2 are a protocol error. This overwrites the GroupOrder specified PUBLISH.
- \* Filter Type, Start Location, End Group: See Section 9.7.
- \* Parameters: Parameters associated with this message.

#### 9.15. PUBLISH\_ERROR

The subscriber sends a PUBLISH\_ERROR control message to reject a subscription initiated by PUBLISH.

```
PUBLISH_ERROR Message {
  Type (i) = 0x1F,
  Length (i),
  Request ID (i),
  Error Code (i),
  Error Reason (Reason Phrase),
}
```

Figure 17: MOQT PUBLISH\_ERROR Message

- \* Request ID: The Request ID of the PUBLISH this message is replying to Section 9.13.
- \* Error Code: Identifies an integer error code for failure.
- \* Error Reason: Provides the reason for subscription error. See Section 1.4.3.

The application SHOULD use a relevant error code in PUBLISH\_ERROR, as defined below:

INTERNAL\_ERROR (0x0): An implementation specific or generic error occurred.

UNAUTHORIZED (0x1): The publisher is not authorized to publish the given namespace or track.

TIMEOUT (0x2): The subscription could not be established before an implementation specific timeout.

NOT\_SUPPORTED (0x3): The endpoint does not support the PUBLISH method.

UNINTERESTED (0x4): The namespace or track is not of interest to the endpoint.

## 9.16. FETCH

A subscriber issues a FETCH to a publisher to request a range of already published objects within a track.

There are three types of Fetch messages.

+=====+	
Code	Fetch Type
+=====+	
0x1	Standalone Fetch
+-----+	
0x2	Relative Joining Fetch
+-----+	
0x3	Absolute Joining Fetch
+-----+	

Table 3

An endpoint that receives a Fetch Type other than 0x1, 0x2 or 0x3 MUST be close the session with a PROTOCOL\_VIOLATION.

### 9.16.1. Standalone Fetch

A Fetch of Objects performed independently of any Subscribe.

A Standalone Fetch includes this structure:

```
Standalone Fetch {  
    Track Namespace (tuple),  
    Track Name Length (i),  
    Track Name (...),  
    Start Location (Location),  
    End Location (Location)  
}
```

- \* Track Namespace: Identifies the namespace of the track as defined in (Section 2.4.1).
- \* Track Name: Identifies the track name as defined in (Section 2.4.1).
- \* Start Location: The start Location.
- \* End Location: The end Location, plus 1. A Location.Object value of 0 means the entire group is requested.

### 9.16.2. Joining Fetches

A Joining Fetch is associated with a Subscribe request by specifying the Request ID of an active subscription. A publisher receiving a Joining Fetch uses properties of the associated Subscribe to determine the Track Namespace, Track Name and End Location such that it is contiguous with the associated Subscribe. The subscriber can set the Start Location to an absolute Location or a Location relative to the current group.

A Subscriber can use a Joining Fetch to, for example, fill a playback buffer with a certain number of groups prior to the live edge of a track.

A Joining Fetch is only permitted when the associated Subscribe has the Filter Type Largest Object; any other value results in closing the session with a `PROTOCOL_VIOLATION`.

If no Objects have been published for the track, and the `SUBSCRIBE_OK` has a Content Exists value of 0, the publisher MUST respond with a `FETCH_ERROR` with error code `INVALID_RANGE`.

A Joining Fetch includes this structure:

```
Joining Fetch {  
  Joining Request ID (i),  
  Joining Start (i)  
}
```

- \* Joining Request ID: The Request ID of the existing subscription to be joined. If a publisher receives a Joining Fetch with a Request ID that does not correspond to an existing Subscribe in the same session, it MUST respond with a Fetch Error with code Invalid Joining Request ID.
- \* Joining Start : A relative or absolute value used to determining the Start Location, described below.

#### 9.16.2.1. Joining Fetch Range Calculation

The Largest Location value from the corresponding subscription is used to calculate the end of a Joining Fetch so the Objects retrieved by the FETCH and SUBSCRIBE are contiguous and non-overlapping.

The publisher receiving a Joining Fetch sets the End Location to {Subscribe Largest Location.Object + 1}.

Note: the last Object included in the Joining FETCH response is Subscribe Largest Location. The + 1 above indicates the equivalent Standalone Fetch encoding.

For a Relative Joining Fetch, the publisher sets the Start Location to {Subscribe Largest Location.Group - Joining Start, 0}.

For an Absolute Joining Fetch, the publisher sets the Start Location to Joining Start.

#### 9.16.3. Fetch Handling

The format of FETCH is as follows:

```
FETCH Message {  
  Type (i) = 0x16,  
  Length (16),  
  Request ID (i),  
  Subscriber Priority (8),  
  Group Order (8),  
  Fetch Type (i),  
  [Standalone (Standalone Fetch)],  
  [Joining (Joining Fetch)],  
  Number of Parameters (i),  
  Parameters (...) ...  
}
```

Figure 18: MOQT FETCH Message

- \* Request ID: See Section 9.1.
- \* Subscriber Priority: Specifies the priority of a fetch request relative to other subscriptions or fetches in the same session. Lower numbers get higher priority. See Section 7.
- \* Group Order: Allows the subscriber to request Objects be delivered in Ascending (0x1) or Descending (0x2) order by group. See Section 7. A value of 0x0 indicates the original publisher's Group Order SHOULD be used. Values larger than 0x2 are a protocol error.
- \* Fetch Type: Identifies the type of Fetch, whether Standalone, Relative Joining or Absolute Joining.
- \* Standalone: Standalone Fetch structure included when Fetch Type is 0x1
- \* Joining: Joining Fetch structure included when Fetch Type is 0x2 or 0x3.
- \* Parameters: The parameters are defined in Section 9.2.1.

A publisher responds to a FETCH request with either a FETCH\_OK or a FETCH\_ERROR message. The publisher creates a new unidirectional stream that is used to send the Objects. The FETCH\_OK or FETCH\_ERROR can come at any time relative to object delivery.

The publisher responding to a FETCH is responsible for delivering all available Objects in the requested range in the requested order. The Objects in the response are delivered on a single unidirectional stream. Any gaps in the Group and Object IDs in the response stream indicate objects that do not exist (eg: they implicitly have status

Object Does Not Exist). For Ascending Group Order this includes ranges between the first requested object and the first object in the stream; between objects in the stream; and between the last object in the stream and the Largest Group/Object indicated in `FETCH_OK`, so long as the fetch stream is terminated by a `FIN`. If no Objects exist in the requested range, the publisher returns `FETCH_ERROR` with code `NO_OBJECTS`.

A relay that has cached objects from the beginning of the range MAY start sending objects immediately in response to a `FETCH`. If it encounters an object in the requested range that is not cached and has unknown status, the relay MUST pause subsequent delivery until it has confirmed the object's status upstream. If the upstream `FETCH` fails, the relay sends a `FETCH_ERROR` and can reset the unidirectional stream. It can choose to do so immediately or wait until the cached objects have been delivered before resetting the stream.

The Object Forwarding Preference does not apply to fetches.

Fetch specifies an inclusive range of Objects starting at Start Location and ending at End Location. End Location MUST specify the same or a larger Location than Start Location for Standalone and Absolute Joining Fetches.

Objects that are not yet published will not be retrieved by a `FETCH`. The Largest available Object in the requested range is indicated in the `FETCH_OK`, and is the last Object a fetch will return if the End Location have not yet been published.

If Start Location is greater than the Largest Object (Section 9.7) the publisher MUST return `FETCH_ERROR` with error code `INVALID_RANGE`.

A publisher MUST send fetched groups in the determined group order, either ascending or descending. Within each group, objects are sent in Object ID order; subgroup ID is not used for ordering.

If an Original Publisher receives a `FETCH` with a range that includes an object with unknown status, it MUST return `FETCH_ERROR` with code `UNKNOWN_STATUS_IN_RANGE`.

#### 9.17. `FETCH_OK`

A publisher sends a `FETCH_OK` control message in response to successful fetches. A publisher MAY send Objects in response to a `FETCH` before the `FETCH_OK` message is sent, but the `FETCH_OK` MUST NOT be sent until the End Location is known.

```
FETCH_OK Message {  
  Type (i) = 0x18,  
  Length (16),  
  Request ID (i),  
  Group Order (8),  
  End Of Track (8),  
  End Location (Location),  
  Number of Parameters (i),  
  Parameters (...) ...  
}
```

Figure 19: MOQT FETCH\_OK Message

- \* Request ID: The Request ID of the FETCH this message is replying to Section 9.7.
- \* Group Order: Indicates the fetch will be delivered in Ascending (0x1) or Descending (0x2) order by group. See Section 7. Values of 0x0 and those larger than 0x2 are a protocol error.
- \* End Of Track: 1 if all Objects have been published on this Track, and the End Location is the final Object in the Track, 0 if not.
- \* End Location: The largest object covered by the FETCH response. The End Location is determined as follows:
  - If the requested FETCH End Location was beyond the Largest known (possibly final) Object, End Location is {Largest.Group, Largest.Object + 1}
  - If End Location.Object in the FETCH request was 0 and the response covers the last Object in the Group, End Location is {Fetch.End Location.Group, 0}
  - Otherwise, End Location is Fetch.End Location Where Fetch.End Location is either Fetch.Standalone.End Location or the computed End Location described in Section 9.16.2.1.

If the relay is subscribed to the track, it uses its knowledge of the largest {Group, Object} to set End Location. If it is not subscribed and the requested End Location exceeds its cached data, the relay makes an upstream request to complete the FETCH, and uses the upstream response to set End Location.

If End Location is smaller than the Start Location in the corresponding FETCH the receiver MUST close the session with PROTOCOL\_VIOLATION.

- \* Parameters: The parameters are defined in Section 9.2.1.

#### 9.18. FETCH\_ERROR

A publisher sends a FETCH\_ERROR control message in response to a failed FETCH.

```
FETCH_ERROR Message {  
    Type (i) = 0x19,  
    Length (16),  
    Request ID (i),  
    Error Code (i),  
    Error Reason (Reason Phrase)  
}
```

Figure 20: MOQT FETCH\_ERROR Message

- \* Request ID: The Request ID of the FETCH this message is replying to Section 9.7.
- \* Error Code: Identifies an integer error code for fetch failure.
- \* Error Reason: Provides the reason for fetch error. See Section 1.4.3.

The application SHOULD use a relevant error code in FETCH\_ERROR, as defined below:

INTERNAL\_ERROR (0x0): An implementation specific or generic error occurred.

UNAUTHORIZED (0x1): The subscriber is not authorized to fetch from the given track.

TIMEOUT (0x2): The fetch could not be completed before an implementation specific timeout. For example, a relay could not FETCH missing objects within the timeout.

NOT\_SUPPORTED (0x3): The endpoint does not support the FETCH method.

TRACK\_DOES\_NOT\_EXIST (0x4): The requested track is not available at the publisher.

INVALID\_RANGE (0x5): The end of the requested range is earlier than the beginning, the start of the requested range is beyond the Largest Location, or the track has not published any Objects yet.

NO\_OBJECTS (0x6): No Objects exist between the requested Start and

End Locations.

INVALID\_JOINING\_REQUEST\_ID (0x7): The joining Fetch referenced a Request ID that did not belong to an active Subscription.

UNKNOWN\_STATUS\_IN\_RANGE (0x8): The requested range contains objects with unknown status.

MALFORMED\_TRACK (0x9): A relay publisher detected the track was malformed (see Section 2.5).

MALFORMED\_AUTH\_TOKEN (0x10): Invalid Auth Token serialization during registration (see Section 9.2.1.1).

EXPIRED\_AUTH\_TOKEN (0x12): Authorization token has expired (Section 9.2.1.1).

#### 9.19. FETCH\_CANCEL

A subscriber sends a FETCH\_CANCEL message to a publisher to indicate it is no longer interested in receiving objects for the fetch identified by the 'Request ID'. The publisher SHOULD promptly close the unidirectional stream, even if it is in the middle of delivering an object.

The format of FETCH\_CANCEL is as follows:

```
FETCH_CANCEL Message {  
  Type (i) = 0x17,  
  Length (16),  
  Request ID (i)  
}
```

Figure 21: MOQT FETCH\_CANCEL Message

- \* Request ID: The Request ID of the FETCH (Section 9.16) this message is cancelling.

#### 9.20. TRACK\_STATUS

A potential subscriber sends a TRACK\_STATUS message on the control stream to obtain information about the current status of a given track.

The TRACK\_STATUS message format is identical to the SUBSCRIBE message (Section 9.7).

The receiver of a TRACK\_STATUS message treats it identically as if it had received a SUBSCRIBE message, except it does not create downstream subscription state or send any Objects. Relays without an active subscription MAY forward TRACK\_STATUS to one or more publishers, or MAY initiate a subscription (subject to authorization) as described in Section 8.4 to determine the response. The publisher does not send PUBLISH\_DONE for this request, and the subscriber cannot send SUBSCRIBE\_UPDATE or UNSUBSCRIBE.

#### 9.21. TRACK\_STATUS\_OK

The publisher sends a TRACK\_STATUS\_OK control message in response to a successful TRACK\_STATUS message.

The TRACK\_STATUS\_OK message format is identical to the SUBSCRIBE\_OK message (Section 9.8).

The publisher populates the fields of TRACK\_STATUS\_OK exactly as it would have populated a SUBSCRIBE\_OK, setting Track Alias to 0. It is not considered an error if Track Alias 0 is already in use by an active subscription.

#### 9.22. TRACK\_STATUS\_ERROR

The publisher sends a TRACK\_STATUS\_ERROR control message in response to a failed TRACK\_STATUS message.

The TRACK\_STATUS\_ERROR message format is identical to the SUBSCRIBE\_ERROR message (Section 9.9).

The publisher populates the fields of TRACK\_STATUS\_ERROR exactly as it would have populated a SUBSCRIBE\_ERROR.

#### 9.23. PUBLISH\_NAMESPACE

The publisher sends the PUBLISH\_NAMESPACE control message to advertise that it has tracks available within a Track Namespace. The receiver verifies the publisher is authorized to publish tracks under this namespace.

```
PUBLISH_NAMESPACE Message {  
  Type (i) = 0x6,  
  Length (16),  
  Request ID (i),  
  Track Namespace (tuple),  
  Number of Parameters (i),  
  Parameters (...) ...,  
}
```

Figure 22: MOQT PUBLISH\_NAMESPACE Message

- \* Request ID: See Section 9.1.
- \* Track Namespace: Identifies a track's namespace as defined in (Section 2.4.1)
- \* Parameters: The parameters are defined in Section 9.2.1.

#### 9.24. PUBLISH\_NAMESPACE\_OK

The subscriber sends a PUBLISH\_NAMESPACE\_OK control message to acknowledge the successful authorization and acceptance of a PUBLISH\_NAMESPACE message.

```
PUBLISH_NAMESPACE_OK Message {  
  Type (i) = 0x7,  
  Length (16),  
  Request ID (i)  
}
```

Figure 23: MOQT PUBLISH\_NAMESPACE\_OK Message

- \* Request ID: The Request ID of the PUBLISH\_NAMESPACE this message is replying to Section 9.23.

#### 9.25. PUBLISH\_NAMESPACE\_ERROR

The subscriber sends a PUBLISH\_NAMESPACE\_ERROR control message for tracks that failed authorization.

```
PUBLISH_NAMESPACE_ERROR Message {  
  Type (i) = 0x8,  
  Length (16),  
  Request ID (i),  
  Error Code (i),  
  Error Reason (Reason Phrase)  
}
```

Figure 24: MOQT PUBLISH\_NAMESPACE\_ERROR Message

- \* Request ID: The Request ID of the PUBLISH\_NAMESPACE this message is replying to Section 9.23.
- \* Error Code: Identifies an integer error code for publish namespace failure.

- \* Error Reason: Provides the reason for publish namespace error.  
See Section 1.4.3.

The application SHOULD use a relevant error code in PUBLISH\_NAMESPACE\_ERROR, as defined below:

INTERNAL\_ERROR (0x0): An implementation specific or generic error occurred.

UNAUTHORIZED (0x1): The subscriber is not authorized to announce the given namespace.

TIMEOUT (0x2): The announce could not be completed before an implementation specific timeout.

NOT\_SUPPORTED (0x3): The endpoint does not support the PUBLISH\_NAMESPACE method.

UNINTERESTED (0x4): The namespace is not of interest to the endpoint.

MALFORMED\_AUTH\_TOKEN (0x10): Invalid Auth Token serialization during registration (see Section 9.2.1.1).

EXPIRED\_AUTH\_TOKEN (0x12): Authorization token has expired (Section 9.2.1.1).

#### 9.26. PUBLISH\_NAMESPACE\_DONE

The publisher sends the PUBLISH\_NAMESPACE\_DONE control message to indicate its intent to stop serving new subscriptions for tracks within the provided Track Namespace.

```
PUBLISH_NAMESPACE_DONE Message {  
  Type (i) = 0x9,  
  Length (16),  
  Track Namespace (tuple),  
}
```

Figure 25: MOQT PUBLISH\_NAMESPACE\_DONE Message

- \* Track Namespace: Identifies a track's namespace as defined in (Section 2.4.1).

## 9.27. PUBLISH\_NAMESPACE\_CANCEL

The subscriber sends an PUBLISH\_NAMESPACE\_CANCEL control message to indicate it will stop sending new subscriptions for tracks within the provided Track Namespace.

```
PUBLISH_NAMESPACE_CANCEL Message {  
    Type (i) = 0xC,  
    Length (16),  
    Track Namespace (tuple),  
    Error Code (i),  
    Error Reason (Reason Phrase),  
}
```

Figure 26: MOQT PUBLISH\_NAMESPACE\_CANCEL Message

- \* Track Namespace: Identifies a track's namespace as defined in (Section 2.4.1).
- \* Error Code: Identifies an integer error code for canceling the publish. PUBLISH\_NAMESPACE\_CANCEL uses the same error codes as PUBLISH\_NAMESPACE\_ERROR (Section 9.25).
- \* Error Reason: Provides the reason for publish cancelation. See Section 1.4.3.

## 9.28. SUBSCRIBE\_NAMESPACE

The subscriber sends the SUBSCRIBE\_NAMESPACE control message to a publisher to request the current set of matching published namespaces and established subscriptions, as well as future updates to the set.

```
SUBSCRIBE_NAMESPACE Message {  
    Type (i) = 0x11,  
    Length (16),  
    Request ID (i),  
    Track Namespace Prefix (tuple),  
    Number of Parameters (i),  
    Parameters (...) ...,  
}
```

Figure 27: MOQT SUBSCRIBE\_NAMESPACE Message

- \* Request ID: See Section 9.1.
- \* Track Namespace Prefix: An ordered N-Tuple of byte fields which are matched against track namespaces known to the publisher. For example, if the publisher is a relay that has received

PUBLISH\_NAMESPACE messages for namespaces ("example.com", "meeting=123", "participant=100") and ("example.com", "meeting=123", "participant=200"), a SUBSCRIBE\_NAMESPACE for ("example.com", "meeting=123") would match both. If an endpoint receives a Track Namespace Prefix tuple with an N of 0 or more than 32, it MUST close the session with a Protocol Violation.

\* Parameters: The parameters are defined in Section 9.2.1.

The publisher will respond with SUBSCRIBE\_NAMESPACE\_OK or SUBSCRIBE\_NAMESPACE\_ERROR. If the SUBSCRIBE\_NAMESPACE is successful, the publisher will immediately forward existing PUBLISH\_NAMESPACE and PUBLISH messages that match the Track Namespace Prefix that have not already been sent to this subscriber. If the set of matching PUBLISH\_NAMESPACE messages changes, the publisher sends the corresponding PUBLISH\_NAMESPACE or PUBLISH\_NAMESPACE\_DONE message.

A subscriber cannot make overlapping namespace subscriptions on a single session. Within a session, if a publisher receives a SUBSCRIBE\_NAMESPACE with a Track Namespace Prefix that is a prefix of, suffix of, or equal to an active SUBSCRIBE\_NAMESPACE, it MUST respond with SUBSCRIBE\_NAMESPACE\_ERROR, with error code NAMESPACE\_PREFIX\_OVERLAP.

The publisher MUST ensure the subscriber is authorized to perform this namespace subscription.

SUBSCRIBE\_NAMESPACE is not required for a publisher to send PUBLISH\_NAMESPACE, PUBLISH\_NAMESPACE\_DONE or PUBLISH messages to a subscriber. It is useful in applications or relays where subscribers are only interested in or authorized to access a subset of available namespaces and tracks.

#### 9.29. SUBSCRIBE\_NAMESPACE\_OK

A publisher sends a SUBSCRIBE\_NAMESPACE\_OK control message for successful namespace subscriptions.

```
SUBSCRIBE_NAMESPACE_OK Message {
  Type (i) = 0x12,
  Length (16),
  Request ID (i),
}
```

Figure 28: MOQT SUBSCRIBE\_NAMESPACE\_OK Message

\* Request ID: The Request ID of the SUBSCRIBE\_NAMESPACE this message is replying to Section 9.28.

## 9.30. SUBSCRIBE\_NAMESPACE\_ERROR

A publisher sends a SUBSCRIBE\_NAMESPACE\_ERROR control message in response to a failed SUBSCRIBE\_NAMESPACE.

```
SUBSCRIBE_NAMESPACE_ERROR Message {  
  Type (i) = 0x13,  
  Length (16),  
  Request ID (i),  
  Error Code (i),  
  Error Reason (Reason Phrase)  
}
```

Figure 29: MOQT SUBSCRIBE\_NAMESPACE\_ERROR Message

- \* Request ID: The Request ID of the SUBSCRIBE\_NAMESPACE this message is replying to Section 9.28.
- \* Error Code: Identifies an integer error code for the namespace subscription failure.
- \* Error Reason: Provides the reason for the namespace subscription error. See Section 1.4.3.

The application SHOULD use a relevant error code in SUBSCRIBE\_NAMESPACE\_ERROR, as defined below:

INTERNAL\_ERROR (0x0): An implementation specific or generic error occurred.

UNAUTHORIZED (0x1): The subscriber is not authorized to subscribe to the given namespace prefix.

TIMEOUT (0x2): The operation could not be completed before an implementation specific timeout.

NOT\_SUPPORTED (0x3): The endpoint does not support the SUBSCRIBE\_NAMESPACE method.

NAMESPACE\_PREFIX\_UNKNOWN (0x4): The namespace prefix is not available for subscription.

NAMESPACE\_PREFIX\_OVERLAP (0x5): The namespace prefix overlaps with another SUBSCRIBE\_NAMESPACE in the same session.

MALFORMED\_AUTH\_TOKEN (0x10): Invalid Auth Token serialization during registration (see Section 9.2.1.1).

EXPIRED\_AUTH\_TOKEN (0x12): Authorization token has expired  
(Section 9.2.1.1).

### 9.31. UNSUBSCRIBE\_NAMESPACE

A subscriber issues a UNSUBSCRIBE\_NAMESPACE message to a publisher indicating it is no longer interested in PUBLISH\_NAMESPACE, PUBLISH\_NAMESPACE\_DONE and PUBLISH messages for the specified track namespace prefix.

The format of UNSUBSCRIBE\_NAMESPACE is as follows:

```
UNSUBSCRIBE_NAMESPACE Message {
  Type (i) = 0x14,
  Length (16),
  Track Namespace Prefix (tuple)
}
```

Figure 30: MOQT UNSUBSCRIBE\_NAMESPACE Message

\* Track Namespace Prefix: As defined in Section 9.28.

## 10. Data Streams and Datagrams

A publisher sends Objects matching a subscription on Data Streams or Datagrams.

All unidirectional MOQT streams start with a variable-length integer indicating the type of the stream in question.

ID	Type
0x10-0x1D	SUBGROUP_HEADER (Section 10.4.2)
0x05	FETCH_HEADER (Section 10.4.4)

Table 4

All MOQT datagrams start with a variable-length integer indicating the type of the datagram.

ID	Type
0x00-0x07,0x20-21	OBJECT_DATAGRAM (Section 10.3.1)

Table 5

An endpoint that receives an unknown stream or datagram type MUST close the session.

Every Track has a single 'Object Forwarding Preference' and the Original Publisher MUST NOT mix different forwarding preferences within a single track (see Section 2.5).

### 10.1. Track Alias

To optimize wire efficiency, Subgroups and Datagrams refer to a track by a numeric identifier, rather than the Full Track Name. Track Alias is chosen by the publisher and included in SUBSCRIBE\_OK (Section 9.8) or PUBLISH (Section 9.13).

Objects can arrive after a subscription has been cancelled. Subscribers SHOULD retain sufficient state to quickly discard these unwanted Objects, rather than treating them as belonging to an unknown Track Alias.

### 10.2. Objects

An Object contains a range of contiguous bytes from the specified track, as well as associated metadata required to deliver, cache, and forward it. Objects are sent by publishers.

#### 10.2.1. Canonical Object Properties

A canonical MoQ Object has the following information:

- \* Track Namespace and Track Name: The track this object belongs to.
- \* Group ID: The object is a member of the indicated group ID Section 2.3 within the track.
- \* Object ID: The order of the object within the group.
- \* Publisher Priority: An 8 bit integer indicating the publisher's priority for the Object Section 7.

- \* Object Forwarding Preference: An enumeration indicating how a publisher sends an object. The preferences are Subgroup and Datagram. When in response to a SUBSCRIBE, an Object MUST be sent according to its Object Forwarding Preference, described below.
- \* Subgroup ID: The object is a member of the indicated subgroup ID (Section 2.2) within the group. This field is omitted if the Object Forwarding Preference is Datagram.
- \* Object Status: As enumeration used to indicate missing objects or mark the end of a group or track. See Section 10.2.1.1 below.
- \* Object Extension Length: The total length of the Object Extension Headers block, in bytes.
- \* Object Extensions : A sequence of Object Extension Headers. See Section 10.2.1.2 below.
- \* Object Payload: An opaque payload intended for an End Subscriber and SHOULD NOT be processed by a relay. Only present when 'Object Status' is Normal (0x0).

#### 10.2.1.1. Object Status

The Object Status informs subscribers what objects will not be received because they were never produced, are no longer available, or because they are beyond the end of a group or track.

Status can have following values:

- \* 0x0 := Normal object. This status is implicit for any non-zero length object. Zero-length objects explicitly encode the Normal status.
- \* 0x1 := Indicates Object Does Not Exist. Indicates that this Object does not exist at any publisher and it will not be published in the future. This SHOULD be cached.
- \* 0x3 := Indicates End of Group. Object ID is one greater than the largest Object produced in the Group identified by the Group ID. If the Object ID is 0, it indicates there are no Objects in this Group. This SHOULD be cached. A publisher MAY use an end of Group object to signal the end of all open Subgroups in a Group. A non-zero-length Object can be the End of Group, as signaled in the DATAGRAM or SUBGROUP\_HEADER Type field (see Section 10.3.1 and Section 10.4.2).

- \* 0x4 := Indicates End of Track. Group ID is either the largest Group produced in this Track with Object ID one greater than the largest Object produced in that Group, or Group ID is one greater than the largest Group produced in this Track with Object ID zero. This status also indicates the specified Group has ended. Publishers MUST NOT publish an Object with a Location larger than this Location (see Section 2.5). This SHOULD be cached.

Any other value SHOULD be treated as a protocol error and the session SHOULD be terminated with a PROTOCOL\_VIOLATION (Section 3.4). Any object with a status code other than zero MUST have an empty payload.

#### 10.2.1.2. Object Extension Header

Any Object may have extension headers except those with Object Status 'Object Does Not Exist'. If an endpoint receives a non-existent Object containing extension headers it MUST close the session with a PROTOCOL\_VIOLATION.

Object Extension Headers are visible to relays and allow the transmission of future metadata relevant to MOQT Object distribution. Any Object metadata never accessed by the transport or relays SHOULD be serialized as part of the Object payload and not as an extension header.

Extension Headers are defined in external specifications and registered in an IANA table Section 13. These specifications define the type and value of the header, along with any rules concerning processing, modification, caching and forwarding. A relay which is coded to implement these rules is said to "support" the extension.

If unsupported by the relay, Extension Headers MUST NOT be modified, MUST be cached as part of the Object and MUST be forwarded by relays.

If supported by the relay and subject to the processing rules specified in the definition of the extension, Extension Headers MAY be modified, added, removed, and/or cached by relays.

Object Extension Headers are serialized as Key-Value-Pairs (see Figure 2).

Header types are registered in the IANA table 'MOQ Extension Headers'. See Section 13.

### 10.3. Datagrams

A single object can be conveyed in a datagram. The Track Alias field Section 10.1 indicates the track this Datagram belongs to. If an endpoint receives a datagram with an unknown Track Alias, it MAY drop the datagram or choose to buffer it for a brief period to handle reordering with the control message that establishes the Track Alias.

An Object received in an OBJECT\_DATAGRAM or OBJECT\_DATAGRAM\_STATUS message has an Object Forwarding Preference = Datagram.

To send an Object with Object Forwarding Preference = Datagram, determine the length of the header and payload and send the Object as datagram. When the total size is larger than maximum datagram size for the session, the Object will be dropped without any explicit notification.

Each session along the path between the Original Publisher and End Subscriber might have different maximum datagram sizes. Additionally, Object Extension Headers (Section 10.2.1.2) can be added to Objects as they pass through the MOQT network, increasing the size of the Object and the chances it will exceed the maximum datagram size of a downstream session and be dropped.

#### 10.3.1. Object Datagram

An OBJECT\_DATAGRAM carries a single object in a datagram.

```
OBJECT_DATAGRAM {
  Type (i) = 0x0-0x7,0x20-21
  Track Alias (i),
  Group ID (i),
  [Object ID (i),]
  Publisher Priority (8),
  [Extension Headers Length (i),
  Extension headers (...)],
  [Object Status (i),]
  [Object Payload (...),]
}
```

Figure 31: MOQT OBJECT\_DATAGRAM

The Type value determines which fields are present in the OBJECT\_DATAGRAM. There are 10 defined Type values for OBJECT\_DATAGRAM.

Type	End Of Group	Extensions	Object ID	Status / Payload
		Present	Present	
0x00	No	No	Yes	Payload
0x01	No	Yes	Yes	Payload
0x02	Yes	No	Yes	Payload
0x03	Yes	Yes	Yes	Payload
0x04	No	No	No	Payload
0x05	No	Yes	No	Payload
0x06	Yes	No	No	Payload
0x07	Yes	Yes	No	Payload
0x20	No	No	Yes	Status
0x21	No	Yes	Yes	Status

Table 6

- \* End of Group: For Type values where End of Group is "Yes" the Object is the last Object in the Group.
- \* Extensions Present: If Extensions Present is "Yes" the Extension Headers Length and Extension headers fields are included. If an endpoint receives a datagram with Extensions Present as "Yes" and a Extension Headers Length of 0, it MUST close the session with PROTOCOL\_VIOLATION.
- \* Object ID Present: If Object ID Present is No, the Object ID field is omitted and the Object ID is 0. When Object ID Present is Yes, the Object ID field is present and encodes the Object ID.
- \* Payload and Status: The Object Status field and Object Payload are mutually exclusive.
  - For Type values 0x00 through 0x07, the Object Payload is present and the Object Status field is omitted.

There is no explicit length field for the Object Payload. The entirety of the transport datagram following the Object header fields contains the payload.

- For Type values 0x20 and 0x21, the Object Status field is present and there is no Object Payload.

#### 10.4. Streams

When Objects are sent on streams, the stream begins with a Subgroup or Fetch Header and is followed by one or more sets of serialized Object fields. If a stream ends gracefully (i.e., the stream terminates with a FIN) in the middle of a serialized Object, the session SHOULD be terminated with a `PROTOCOL_VIOLATION`.

A publisher SHOULD NOT open more than one stream at a time with the same Subgroup Header field values.

##### 10.4.1. Stream Cancellation

Streams aside from the control stream MAY be canceled due to congestion or other reasons by either the publisher or subscriber. Early termination of a stream does not affect the MoQ application state, and therefore has no effect on outstanding subscriptions.

##### 10.4.2. Subgroup Header

All Objects on a Subgroup stream belong to the track identified by Track Alias (see Section 10.1) and the Subgroup indicated by 'Group ID' and Subgroup ID in the `SUBGROUP_HEADER`.

If an endpoint receives a subgroup with an unknown Track Alias, it MAY abandon the stream, or choose to buffer it for a brief period to handle reordering with the control message that establishes the Track Alias. The endpoint MAY withhold stream flow control beyond the `SUBGROUP_HEADER` until the Track Alias has been established. To prevent deadlocks, the publisher MUST allocate connection flow control to the control stream before allocating it any data streams. Otherwise, a receiver might wait for a control message containing a Track Alias to release flow control, while the sender waits for flow control to send the message.

```
SUBGROUP_HEADER {  
    Type (i) = 0x10..0x1D,  
    Track Alias (i),  
    Group ID (i),  
    [Subgroup ID (i),]  
    Publisher Priority (8),  
}
```

Figure 32: MOQT SUBGROUP\_HEADER

All Objects received on a stream opened with SUBGROUP\_HEADER have an Object Forwarding Preference = Subgroup.

There are 12 defined Type values for SUBGROUP\_HEADER:

Type	Subgroup ID	Subgroup ID	Extensions	Contains End
	Field Present	Value	Present	of Group
0x10	No	0	No	No
0x11	No	0	Yes	No
0x12	No	First Object ID	No	No
0x13	No	First Object ID	Yes	No
0x14	Yes	N/A	No	No
0x15	Yes	N/A	Yes	No
0x18	No	0	No	Yes
0x19	No	0	Yes	Yes
0x1A	No	First Object ID	No	Yes
0x1B	No	First Object ID	Yes	Yes
0x1C	Yes	N/A	No	Yes
0x1D	Yes	N/A	Yes	Yes

Table 7

For Type values where Contains End of Group is Yes, the last Object in this Subgroup stream before a FIN is the last Object in the Group. If the Subgroup stream is terminated with a RESET\_STREAM or RESET\_STREAM\_AT, the receiver cannot determine the End of Group Object ID.

For Type values where Subgroup ID Field Present is No, there is no explicit Subgroup ID field in the header and the Subgroup ID is either 0 (for Types 0x10-11 and 0x18-19) or the Object ID of the first object transmitted in this subgroup (for Types 0x12-13 and 0x1A-1B).

For Type values where Extensions Present is No, Extensions Headers Length is not present and all Objects have no extensions. When Extensions Present is Yes, Extension Headers Length is present in all Objects in this subgroup. Objects with no extensions set Extension Headers Length to 0.

To send an Object with Object Forwarding Preference = Subgroup, find the open stream that is associated with the subscription, Group ID and Subgroup ID, or open a new one and send the SUBGROUP\_HEADER. Then serialize the following fields.

The Object Status field is only sent if the Object Payload Length is zero.

The Object ID Delta + 1 is added to the previous Object ID in the Subgroup stream if there was one. The Object ID is the Object ID Delta if it's the first Object in the Subgroup stream. For example, a Subgroup of sequential Object IDs starting at 0 will have 0 for all Object ID Delta values. A consumer cannot infer information about the existence of Objects between the current and previous Object ID in the Subgroup (e.g. when Object ID Delta is non-zero) unless there is an Prior Object ID Gap extension header (see Section 11.3).

```
{
  Object ID Delta (i),
  [Extension Headers Length (i),
  Extension headers (...)],
  Object Payload Length (i),
  [Object Status (i)],
  Object Payload (...),
}
```

Figure 33: MOQT Subgroup Object Fields

#### 10.4.4.3. Closing Subgroup Streams

Subscribers will often need to know if they have received all objects in a Subgroup, particularly if they serve as a relay or cache. QUIC and Webtransport streams provide signals that can be used for this purpose. Closing Subgroups promptly frees system resources and often unlocks flow control credit to open more streams.

If a sender has delivered all objects in a Subgroup to the QUIC stream, except any Objects with Locations smaller than the subscription's Start Location, it MUST close the stream with a FIN.

If a sender closes the stream before delivering all such objects to the QUIC stream, it MUST use a RESET\_STREAM or RESET\_STREAM\_AT [I-D.draft-ietf-quic-reliable-stream-reset] frame. This includes an open Subgroup exceeding its Delivery Timeout, early termination of subscription due to an UNSUBSCRIBE message, a publisher's decision to end the subscription early, or a SUBSCRIBE\_UPDATE moving the subscription's End Group to a smaller Group or the Start Location to a larger Location. When RESET\_STREAM\_AT is used, the reliable\_size SHOULD include the stream header so the receiver can identify the corresponding subscription and accurately account for reset data streams when handling PUBLISH\_DONE (see Section 9.12). Publishers that reset data streams without using RESET\_STREAM\_AT with an appropriate reliable\_size can cause subscribers to hold on to subscription state until a timeout expires.

A sender might send all objects in a Subgroup and the FIN on a QUIC stream, and then reset the stream. In this case, the receiving application would receive the FIN if and only if all objects were received. If the application receives all data on the stream and the FIN, it can ignore any RESET\_STREAM it receives.

If a sender will not deliver any objects from a Subgroup, it MAY send a SUBGROUP\_HEADER on a new stream, with no objects, and then send RESET\_STREAM\_AT with a reliable\_size equal to the length of the stream header. This explicitly tells the receiver there is an unsent Subgroup.

A relay MUST NOT forward an Object on an existing Subgroup stream unless it is the next Object in that Subgroup. A relay knows that an Object is the next Object in the Subgroup if at least one of the following is true:

- \* the Object ID is one greater than the previous Object sent on this Subgroup stream.
- \* the Object was received on the same upstream Subgroup stream as the previously sent Object on the downstream Subgroup stream, with no other Objects in between.
- \* it knows all Object IDs between the current and previous Object IDs on the Subgroup stream belong to different Subgroups or do not exist.

If the relay does not know if an Object is the next Object, it MUST reset the Subgroup stream and open a new one to forward it.

Since SUBSCRIBES always end on a group boundary, an ending subscription can always cleanly close all its subgroups. A sender that terminates a stream early for any other reason (e.g., to handoff to a different sender) MUST use RESET\_STREAM or RESET\_STREAM\_AT. Senders SHOULD terminate a stream on Group boundaries to avoid doing so.

An MOQT implementation that processes a stream FIN is assured it has received all objects in a subgroup from the start of the subscription. If a relay, it can forward stream FINs to its own subscribers once those objects have been sent. A relay MAY treat receipt of EndOfGroup, GroupDoesNotExist, or EndOfTrack objects as a signal to close corresponding streams even if the FIN has not arrived, as further objects on the stream would be a protocol violation.

Similarly, an EndOfGroup message indicates the maximum Object ID in the Group, so if all Objects in the Group have been received, a FIN can be sent on any stream where the entire subgroup has been sent. This might be complex to implement.

Processing a RESET\_STREAM or RESET\_STREAM\_AT means that there might be other objects in the Subgroup beyond the last one received. A relay might immediately reset the corresponding downstream stream, or it might attempt to recover the missing Objects in an effort to send all the Objects in the subgroups and the FIN. It also might send RESET\_STREAM\_AT with reliable\_size set to the last Object it has, so as to reliably deliver the Objects it has while signaling that other Objects might exist.

A subscriber MAY send a QUIC STOP\_SENDING frame for a subgroup stream if the Group or Subgroup is no longer of interest to it. The publisher SHOULD respond with RESET\_STREAM or RESET\_STREAM\_AT. If RESET\_STREAM\_AT is sent, note that the receiver has indicated no interest in the objects, so setting a reliable\_size beyond the stream header is of questionable utility.

RESET\_STREAM and STOP\_SENDING on SUBSCRIBE data streams have no impact on other Subgroups in the Group or the subscription, although applications might cancel all Subgroups in a Group at once.

The application SHOULD use a relevant error code in RESET\_STREAM or RESET\_STREAM\_AT, as defined below:

INTERNAL\_ERROR (0x0): An implementation specific error.

CANCELLED (0x1): The subscriber requested cancellation via

UNSUBSCRIBE, FETCH\_CANCEL or STOP\_SENDING, or the publisher ended the subscription, in which case PUBLISH\_DONE (Section 9.12) will have a more detailed status code.

DELIVERY\_TIMEOUT (0x2): The DELIVERY\_TIMEOUT Section 9.2.1.2 was exceeded for this stream.

SESSION\_CLOSED (0x3): The publisher session is being closed.

#### 10.4.4. Fetch Header

When a stream begins with FETCH\_HEADER, all objects on the stream belong to the track requested in the Fetch message identified by Request ID.

```
FETCH_HEADER {  
  Type (i) = 0x5,  
  Request ID (i),  
}
```

Figure 34: MOQT FETCH\_HEADER

Each object sent on a fetch stream after the FETCH\_HEADER has the following format:

```
{  
  Group ID (i),  
  Subgroup ID (i),  
  Object ID (i),  
  Publisher Priority (8),  
  Extension Headers Length (i),  
  [Extension headers (...)],  
  Object Payload Length (i),  
  [Object Status (i)],  
  Object Payload (...),  
}
```

Figure 35: MOQT Fetch Object Fields

The Object Status field is only sent if the Object Payload Length is zero.

The Subgroup ID field of an object with a Forwarding Preference of "Datagram" (see Section 10.2.1) is set to the Object ID.

### 10.5. Examples

Sending a subgroup on one stream:

Stream = 2

```
SUBGROUP_HEADER {
  Type = 0x14
  Track Alias = 2
  Group ID = 0
  Subgroup ID = 0
  Publisher Priority = 0
}
{
  Object ID = 0
  Object Payload Length = 4
  Payload = "abcd"
}
{
  Object ID = 1
  Object Payload Length = 4
  Payload = "efgh"
}
```

Sending a group on one stream, with the first object containing two Extension Headers.

```
Stream = 2

SUBGROUP_HEADER {
  Type = 0x15
  Track Alias = 2
  Group ID = 0
  Subgroup ID = 0
  Publisher Priority = 0
}
{
  Object ID Delta = 0 (Object ID is 0)
  Extension Headers Length = 33
  { Type = 4
    Value = 2186796243
  },
  { Type = 77
    Length = 21
    Value = "traceID:123456"
  }
  Object Payload Length = 4
  Payload = "abcd"
}
{
  Object ID Delta = 0 (Object ID is 1)
  Extension Headers Length = 0
  Object Payload Length = 4
  Payload = "efgh"
}
```

## 11. Extension Headers

The following Object Extension Headers are defined in MOQT.

### 11.1. Prior Group ID Gap

Prior Group ID Gap (Extension Header Type 0x3C) is a variable length integer containing the number of Groups prior to the current Group that do not and will never exist. This is equivalent to receiving an End of Group status with Object ID 0 for each skipped Group. For example, if the Original Publisher is publishing an Object in Group 7 and knows it will never publish any Objects in Group 8 or Group 9, it can include Prior Group ID Gap = 2 in any number of Objects in Group 10, as it sees fit. A Track is considered malformed (see Section 2.5) if any of the following conditions are detected:

- \* An Object contains more than one instance of Prior Group ID Gap

- \* A Group contains more than one Object with different values for Prior Group ID Gap
- \* An Object has a Prior Group ID Gap larger than the Group ID
- \* An endpoint receives an Object with a Prior Group ID Gap covering an Object it previously received
- \* An endpoint receives an Object with a Group ID within a previously communicated gap

This extension is optional, as publishers might not know the prior gap gize, or there may not be a gap. If Prior Group ID Gap is not present, the receiver cannot infer any information about the existence of prior groups (see Section 2.3.1).

This extension can be added by the Original Publisher, but MUST NOT be added by relays. This extension MUST NOT be modified or removed.

## 11.2. Immutable Extensions

The Immutable Extensions (Extension Header Type 0xB) contains a sequence of Key-Value-Pairs (see Figure 2) which are also Object Extension Headers of the Object.

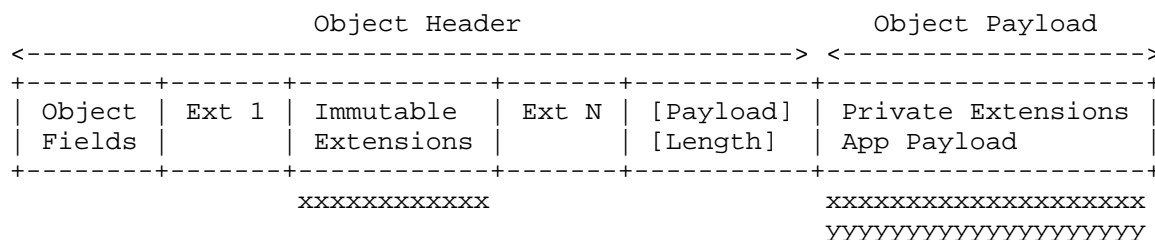
```
Immutable Extensions {  
    Type (0xB),  
    Length (i),  
    Key-Value-Pair (...) ...  
}
```

This extension can be added by the Original Publisher, but MUST NOT be added by Relays. This extension MUST NOT be modified or removed. Relays MUST cache this extension if the Object is cached and MUST forward this extension if the enclosing Object is forwarded. Relays MAY decode and view these extensions.

A Track is considered malformed (see Section 2.5) if any of the following conditions are detected:

- \* An Object contains an Immutable Extensions header that contains another Immutable Extensions key
- \* A Key-Value-Pair cannot be parsed

The following figure shows an example Object structure with a combination of mutable and immutable extensions and end to end encrypted metadata in the Object payload.



x = e2e Authenticated Data

y = e2e Encrypted Data

EXT 1 and EXT N can be modified or removed by Relays

### 11.3. Prior Object ID Gap

Prior Object ID Gap (Extension Header Type 0x3E) is a variable length integer containing the number of Objects prior to the current Object that do not and will never exist. This is equivalent to receiving an Object Does Not Exist status for each skipped Object ID. For example, if the Original Publisher is publishing Object 10 in Group 3 and knows it will never publish Objects 8 or 9 in this Group, it can include Prior Object ID Gap = 2. A Track is considered malformed (see Section 2.5) if any of the following conditions are detected:

- \* An Object contains more than one instance of Prior Object ID Gap
- \* An Object has a Prior Object ID Gap larger than the Object ID
- \* An endpoint receives an Object with a Prior Object ID Gap covering an Object it previously received
- \* An endpoint receives an Object with an Object ID within a previously communicated gap

This extension is optional, as publishers might not know the prior gap gize, or there may not be a gap. If Prior Object ID Gap is not present, the receiver cannot infer any information about the existence of prior objects (see Section 2.1).

This extension can be added by the Original Publisher, but MUST NOT be added by relays. This extension MUST NOT be modified or removed.

## 12. Security Considerations

TODO: Expand this section, including subscriptions.

TODO: Describe Cache Poisoning attacks

### 12.1. Resource Exhaustion

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

MOQT uses stream limits and 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 an application could have in flight.

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

### 12.2. Timeouts

Implementations are advised to use timeouts to prevent resource exhaustion attacks by a peer that does not send expected data within an expected time. Each implementation is expected to set its own limits.

### 12.3. Relay security considerations

#### 12.3.1. State maintenance

A Relay SHOULD have mechanisms to prevent malicious endpoints from flooding it with PUBLISH\_NAMESPACE or SUBSCRIBE\_NAMESPACE requests that could bloat data structures. It could use the advertised MAX\_REQUEST\_ID to limit the number of such requests, or could have application-specific policies that can reject incoming PUBLISH\_NAMESPACE or SUBSCRIBE\_NAMESPACE requests that cause the state maintenance for the session to be excessive.

#### 12.3.2. SUBSCRIBE\_NAMESPACE with short prefixes

A Relay can use authorization rules in order to prevent subscriptions closer to the root of a large prefix tree. Otherwise, if an entity sends a relay a SUBSCRIBE\_NAMESPACE message with a short prefix, it can cause the relay to send a large volume of PUBLISH\_NAMESPACE messages. As churn continues in the tree of prefixes, the relay would have to continue to send PUBLISH\_NAMESPACE/PUBLISH\_NAMESPACE\_DONE messages to the entity that had sent the SUBSCRIBE\_NAMESPACE.

TODO: Security/Privacy Considerations of MOQT\_IMPLEMENTATION parameter

### 13. IANA Considerations

TODO: fill out currently missing registries:

- \* MOQT version numbers
- \* Setup parameters
- \* Non-setup Parameters - List which params can be repeated in the table.
- \* Message types
- \* MOQ Extension headers - we wish to reserve extension types 0-63 for standards utilization where space is a premium, 64 - 16383 for standards utilization where space is less of a concern, and 16384 and above for first-come-first-served non-standardization usage. List which headers can be repeated in the table.
- \* MOQT Auth Token Type

TODO: register the URI scheme and the ALPN and grease the Extension types

#### 13.1. Error Codes

##### 13.1.1. Session Termination Error Codes

Name	Code	Specification
NO_ERROR	0x0	Section 3.4
INTERNAL_ERROR	0x1	Section 3.4
UNAUTHORIZED	0x2	Section 3.4
PROTOCOL_VIOLATION	0x3	Section 3.4
INVALID_REQUEST_ID	0x4	Section 3.4
DUPLICATE_TRACK_ALIAS	0x5	Section 3.4
KEY_VALUE_FORMATTING_ERROR	0x6	Section 3.4

TOO_MANY_REQUESTS	0x7	Section 3.4	
+-----+-----+-----+			
INVALID_PATH	0x8	Section 3.4	
+-----+-----+-----+			
MALFORMED_PATH	0x9	Section 3.4	
+-----+-----+-----+			
GOAWAY_TIMEOUT	0x10	Section 3.4	
+-----+-----+-----+			
CONTROL_MESSAGE_TIMEOUT	0x11	Section 3.4	
+-----+-----+-----+			
DATA_STREAM_TIMEOUT	0x12	Section 3.4	
+-----+-----+-----+			
AUTH_TOKEN_CACHE_OVERFLOW	0x13	Section 3.4	
+-----+-----+-----+			
DUPLICATE_AUTH_TOKEN_ALIAS	0x14	Section 3.4	
+-----+-----+-----+			
VERSION_NEGOTIATION_FAILED	0x15	Section 3.4	
+-----+-----+-----+			
MALFORMED_AUTH_TOKEN	0x16	Section 3.4	
+-----+-----+-----+			
UNKNOWN_AUTH_TOKEN_ALIAS	0x17	Section 3.4	
+-----+-----+-----+			
EXPIRED_AUTH_TOKEN	0x18	Section 3.4	
+-----+-----+-----+			
INVALID_AUTHORITY	0x19	Section 3.4	
+-----+-----+-----+			
MALFORMED_AUTHORITY	0x1A	Section 3.4	
+-----+-----+-----+			

Table 8

## 13.1.2. SUBSCRIBE\_ERROR Codes

+=====+	+=====+	+=====+	
Name	Code	Specification	
+=====+	+=====+	+=====+	
INTERNAL_ERROR	0x0	Section 9.9	
+-----+-----+	+-----+-----+	+-----+-----+	
UNAUTHORIZED	0x1	Section 9.9	
+-----+-----+	+-----+-----+	+-----+-----+	
TIMEOUT	0x2	Section 9.9	
+-----+-----+	+-----+-----+	+-----+-----+	
NOT_SUPPORTED	0x3	Section 9.9	
+-----+-----+	+-----+-----+	+-----+-----+	
TRACK_DOES_NOT_EXIST	0x4	Section 9.9	
+-----+-----+	+-----+-----+	+-----+-----+	
INVALID_RANGE	0x5	Section 9.9	
+-----+-----+	+-----+-----+	+-----+-----+	

MALFORMED_AUTH_TOKEN	0x10	Section 9.9	
+-----+	+-----+	+-----+	+-----+
EXPIRED_AUTH_TOKEN	0x12	Section 9.9	
+-----+	+-----+	+-----+	+-----+

Table 9

## 13.1.3. PUBLISH\_DONE Codes

+-----+	+-----+	+-----+	+-----+
Name	Code	Specification	
+-----+	+-----+	+-----+	+-----+
INTERNAL_ERROR	0x0	Section 9.12	
+-----+	+-----+	+-----+	+-----+
UNAUTHORIZED	0x1	Section 9.12	
+-----+	+-----+	+-----+	+-----+
TRACK_ENDED	0x2	Section 9.12	
+-----+	+-----+	+-----+	+-----+
SUBSCRIPTION_ENDED	0x3	Section 9.12	
+-----+	+-----+	+-----+	+-----+
GOING_AWAY	0x4	Section 9.12	
+-----+	+-----+	+-----+	+-----+
EXPIRED	0x5	Section 9.12	
+-----+	+-----+	+-----+	+-----+
TOO_FAR_BEHIND	0x6	Section 9.12	
+-----+	+-----+	+-----+	+-----+
MALFORMED_TRACK	0x7	Section 9.12	
+-----+	+-----+	+-----+	+-----+

Table 10

## 13.1.4. PUBLISH\_ERROR Codes

+-----+	+-----+	+-----+	+-----+
Name	Code	Specification	
+-----+	+-----+	+-----+	+-----+
INTERNAL_ERROR	0x0	Section 9.15	
+-----+	+-----+	+-----+	+-----+
UNAUTHORIZED	0x1	Section 9.15	
+-----+	+-----+	+-----+	+-----+
TIMEOUT	0x2	Section 9.15	
+-----+	+-----+	+-----+	+-----+
NOT_SUPPORTED	0x3	Section 9.15	
+-----+	+-----+	+-----+	+-----+
UNINTERESTED	0x4	Section 9.15	
+-----+	+-----+	+-----+	+-----+

Table 11

## 13.1.5. FETCH\_ERROR Codes

Name	Code	Specification
INTERNAL_ERROR	0x0	Section 9.18
UNAUTHORIZED	0x1	Section 9.18
TIMEOUT	0x2	Section 9.18
NOT_SUPPORTED	0x3	Section 9.18
TRACK_DOES_NOT_EXIST	0x4	Section 9.18
INVALID_RANGE	0x5	Section 9.18
NO_OBJECTS	0x6	Section 9.18
INVALID_JOINING_REQUEST_ID	0x7	Section 9.18
UNKNOWN_STATUS_IN_RANGE	0x8	Section 9.18
MALFORMED_TRACK	0x9	Section 9.18
MALFORMED_AUTH_TOKEN	0x10	Section 9.18
EXPIRED_AUTH_TOKEN	0x12	Section 9.18

Table 12

## 13.1.6. ANNOUNCE\_ERROR Codes

Name	Code	Specification
INTERNAL_ERROR	0x0	Section 9.25
UNAUTHORIZED	0x1	Section 9.25
TIMEOUT	0x2	Section 9.25
NOT_SUPPORTED	0x3	Section 9.25
UNINTERESTED	0x4	Section 9.25
MALFORMED_AUTH_TOKEN	0x10	Section 9.25

EXPIRED_AUTH_TOKEN	0x12	Section 9.25
--------------------	------	--------------

Table 13

## 13.1.7. SUBSCRIBE\_NAMESPACE\_ERROR Codes

Name	Code	Specification
INTERNAL_ERROR	0x0	Section 9.30
UNAUTHORIZED	0x1	Section 9.30
TIMEOUT	0x2	Section 9.30
NOT_SUPPORTED	0x3	Section 9.30
NAMESPACE_PREFIX_UNKNOWN	0x4	Section 9.30
NAMESPACE_PREFIX_OVERLAP	0x5	Section 9.30
MALFORMED_AUTH_TOKEN	0x10	Section 9.30
EXPIRED_AUTH_TOKEN	0x12	Section 9.30

Table 14

## 13.1.8. Data Stream Reset Error Codes

Name	Code	Specification
INTERNAL_ERROR	0x0	Section 10.4.3
CANCELLED	0x1	Section 10.4.3
DELIVERY_TIMEOUT	0x2	Section 10.4.3
SESSION_CLOSED	0x3	Section 10.4.3

Table 15

## Contributors

The original design behind this protocol was inspired by three independent proposals: WARP [I-D.draft-lcurley-warp] by Luke Curley, RUSH [I-D.draft-kpugin-rush] by Kirill Pugin, Nitin Garg, Alan Frindell, Jordi Cenzano and Jake Weissman, and QUICR [I-D.draft-jennings-moq-quicr-proto] by Cullen Jennings, Suhas Nandakumar and Christian Huitema. The authors of those documents merged their proposals to create the first draft of moq-transport. The IETF MoQ Working Group received an enormous amount of support from many people. The following people provided substantive contributions to this document:

- \* Ali Begen
- \* Charles Krasic
- \* Christian Huitema
- \* Cullen Jennings
- \* James Hurley
- \* Jordi Cenzano
- \* Kirill Pugin
- \* Luke Curley
- \* Martin Duke
- \* Mike English
- \* Mo Zanaty
- \* Will Law

## References

### Normative References

- [I-D.draft-ietf-quic-reliable-stream-reset]  
Seemann, M. and K. Oku, "QUIC Stream Resets with Partial Delivery", Work in Progress, Internet-Draft, draft-ietf-quic-reliable-stream-reset-07, 14 June 2025,  
<<https://datatracker.ietf.org/doc/html/draft-ietf-quic-reliable-stream-reset-07>>.

- [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-http3-13, 7 July 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-http3-13>>.

#### Informative References

- [I-D.draft-jennings-moq-quicr-proto] Jennings, C. F., Nandakumar, S., and C. Huitema, "QuicR - Media Delivery Protocol over QUIC", Work in Progress, Internet-Draft, draft-jennings-moq-quicr-proto-01, 11 July 2022, <<https://datatracker.ietf.org/doc/html/draft-jennings-moq-quicr-proto-01>>.

[I-D.draft-kpugin-rush]

Pugin, K., Garg, N., Frindell, A., Ferret, J. C., and J. Weissman, "RUSH - Reliable (unreliable) streaming protocol", Work in Progress, Internet-Draft, draft-kpugin-rush-03, 21 April 2025, <<https://datatracker.ietf.org/doc/html/draft-kpugin-rush-03>>.

[I-D.draft-lcurley-warp]

Curley, L., Pugin, K., Nandakumar, S., and V. Vasiliev, "Warp - Live Media Transport over QUIC", Work in Progress, Internet-Draft, draft-lcurley-warp-04, 13 March 2023, <<https://datatracker.ietf.org/doc/html/draft-lcurley-warp-04>>.

[I-D.ietf-webtrans-overview]

Kinnear, E. and V. Vasiliev, "The WebTransport Protocol Framework", Work in Progress, Internet-Draft, draft-ietf-webtrans-overview-10, 7 July 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-overview-10>>.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <<https://www.rfc-editor.org/rfc/rfc6582>>.

[RFC9000] 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>>.

[RFC9438] Xu, L., Ha, S., Rhee, I., Goel, V., and L. Eggert, Ed., "CUBIC for Fast and Long-Distance Networks", RFC 9438, DOI 10.17487/RFC9438, August 2023, <<https://www.rfc-editor.org/rfc/rfc9438>>.

## Appendix A. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

### A.1. Since draft-ietf-moq-transport-13

\*Setup and Control Plane\*

- \* Add an AUTHORITY parameter (#1058)
- \* Add a free-form SETUP parameter identifying the implementation (#1114)
- \* Add a Request ID to SUBSCRIBE\_UDPATE (#1106)
- \* Indicate which params can appear PUBLISH\* messages (#1071)
- \* Add TRACK\_STATUS to the list of request types affected by GOAWAY (#1105)

\*Data Plane Wire Format and Handling\*

- \* Delta encode Object IDs within Subgroups (#1042)
- \* Use a bit in Datagram Type to convey Object ID = 0 (#1055)
- \* Corrected missed code point updates to Object Datagram Status (#1082)
- \* Merge OBJECT\_DATAGRAM and OBJECT\_DATAGRAM\_STATUS description (#1179)
- \* Objects are not schedulable if flow-control blocked (#1054)
- \* Clarify DELIVERY\_TIMEOUT reordering computation (#1120)
- \* Receiving unrequested Objects (#1112)
- \* Clarify End of Track (#1111)
- \* Malformed tracks apply to FETCH (#1083)
- \* Remove early FIN from the definition of malformed tracks (#1096)
- \* Prior Object ID Gap Extension header (#939)
- \* Add Extension containing immutable extensions (#1025)

\*Relay Handling\*

- \* Explain FETCH routing for relays (#1165)
- \* MUST for multi-publisher relay handling (#1115)
- \* Filters don't (usually) determine the end of subscription (#1113)

- \* Allow self-subscriptions (#1110)
- \* Explain Namespace Prefix Matching in more detail (#1116)
- \*Explanatory\*
- \* Explain Modularity of MOQT (#1107)
- \* Explain how to resume publishing after losing state (#1087)
- \*Major Editorial Changes\*
- \* Rename ANNOUNCE to PUBLISH\_NAMESPACE (#1104)
- \* Rename SUBSCRIBE\_DONE to PUBLISH\_DONE (#1108)
- \* Major FETCH Reorganization (#1173)
- \* Reformat Error Codes (#1091)

#### A.2. Since draft-ietf-moq-transport-12

- \* TRACK\_STATUS\_REQUEST and TRACK\_STATUS have changed to directly mirror SUBSCRIBE/OK/ERROR (#1015)
- \* SUBSCRIBE\_ANNOUNCES was renamed back to SUBSCRIBE\_NAMESPACE (#1049)

#### A.3. Since draft-ietf-moq-transport-11

- \* Move Track Alias from SUBSCRIBE to SUBSCRIBE\_OK (#977)
- \* Expand cases FETCH\_OK returns Invalid Range (#946) and clarify fields (#936)
- \* Add an error code to FETCH\_ERROR when an Object status is unknown (#825)
- \* Rename Latest Object to Largest Object (#1024) and clarify what to do when it's incomplete (#937)
- \* Explain Malformed Tracks and what to do with them (#938)
- \* Allow End of Group to be indicated in a normal Object (#1011)
- \* Relays MUST have an upstream subscription to send SUBSCRIBE\_OK (#1017)

- \* Allow AUTHORIZATION TOKEN in CLIENT\_SETUP, SERVER\_SETUP and other fixes (#1013)
- \* Add PUBLISH for publisher initiated subscriptions (#995) and fix the PUBLISH codepoints (#1048, #1051)

#### A.4. Since draft-ietf-moq-transport-10

- \* Added Common Structure definitions - Location, Key-Value-Pair and Reason Phrase
- \* Limit lengths of all variable length fields, including Track Namespace and Name
- \* Control Message length is now 16 bits instead of variable length
- \* Subscribe ID became Request ID, and was added to most control messages. Request ID is used to correlate OK/ERROR responses for ANNOUNCE, SUBSCRIBE\_NAMESPACE, and TRACK\_STATUS. Like Subscribe ID, Request IDs are flow controlled.
- \* Explain rules for caching in more detail
- \* Changed the SETUP parameter format for even number parameters to match the Object Header Extension format
- \* Rotated SETUP code points
- \* Added Parameters to TRACK\_STATUS and TRACK\_STATUS\_REQUEST
- \* Clarified how subscribe filters work
- \* Added Next Group Filter to SUBSCRIBE
- \* Added Forward flag to SUBSCRIBE
- \* Renamed FETCH\_OK field to End and clarified how to set it
- \* Added Absolute Joining Fetch
- \* Clarified No Error vs Invalid Range FETCH\_ERROR cases
- \* Use bits in SUBGROUP\_HEADER and DATAGRAM\* types to compress subgroup ID and extensions
- \* Coalesced END\_OF\_GROUP and END\_OF\_TRACK\_AND\_GROUP status

- \* Objects that Do Not Exist cannot have extensions when sent on the wire
- \* Specified error codes for resetting data streams
- \* Defined an Object Header Extension for communicating a known Group ID gap
- \* Replaced AUTHORIZATION\_INFO with AUTHORIZATION\_TOKEN, which has more structure, compression, and additional Auth related error codes (#760)

#### Authors' Addresses

Suhas Nandakumar  
Cisco  
Email: snandaku@cisco.com

Victor Vasiliev  
Google  
Email: vasilvv@google.com

Ian Swett (editor)  
Google  
Email: ianswett@google.com

Alan Frindell (editor)  
Meta  
Email: afrind@meta.com