Media Over QUIC
Internet-Draft
Intended status: Standards Track
Expires: 13 November 2026

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

12 May 2026

Media over QUIC Transport
draft-ietf-moq-transport-18

Abstract

This document defines Media over QUIC Transport (MOQT), a publish/subscribe protocol that runs over QUIC and WebTransport. MOQT leverages the features of these transports, such as streams, datagrams, priorities, and partial reliability. MOQT operates both point-to-point and through intermediate relays, enabling scalable low-latency delivery. Despite its name, MOQT is media agnostic and can be used for a wide range of use cases.

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.

This Internet-Draft will expire on 13 November 2026.

Copyright Notice

Table of Contents

1. Introduction

Media Over QUIC Transport (MOQT) is a publish/subscribe protocol that runs over QUIC [QUIC] or WebTransport [WebTransport]. Publishers produce data that is delivered to subscribers either point-to-point or through intermediate relays. MOQT leverages transport features such as streams, datagrams, priorities, and partial reliability to support a wide range of use cases with different resiliency and latency needs, from live to interactive, without compromising scalability.

Despite its name, MOQT is content agnostic. MoQ Streaming Formats define how specific content types are 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 9 covers behavior at the relay entities.

* Section 10 covers how control messages are encoded on the wire.

* Section 11 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 as 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 9.

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. It also uses RESET_STREAM_AT from [I-D.draft-ietf-quic-reliable-stream-reset]. RESET_STREAM_AT can be used by MOQT, but the protocol is also designed to work correctly when the extension is not supported.

When this document says an endpoint "resets" a stream, it means the endpoint sends a RESET_STREAM or RESET_STREAM_AT frame on that stream (see Section 11.4.3 for considerations on choosing between them).

1.4. Notational Conventions

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

1.4.1. Variable-Length Integers

MOQT requires a variable-length integer encoding with the following properties:

1. The encoded length can be determined from the first encoded byte.

2. The range of 1 byte values is as large as possible.

3. All 64 bit numbers can be encoded.

The variable-length integer encoding uses the number of leading 1 bits of the first byte to indicate the length of the encoding in bytes. The remaining bits after the first 0 and subsequent bytes, if any, represent the integer value, encoded in network byte order.

Integers are encoded in 1 to 9 bytes and can encode up to 64 bit unsigned integers. The following table summarizes the encoding properties.

Leading Bits	Length (bytes)	Usable Bits	Range
0	1	7	0-127
10	2	14	0-16383
110	3	21	0-2097151
1110	4	28	0-268435455
11110	5	35	0-34359738367
111110	6	42	0-4398046511103
1111110	7	49	0-562949953421311
11111110	8	56	0-72057594037927935
11111111	9	64	0-18446744073709551615

Table 1: Summary of Integer Encodings

The following table contains some example encodings:

Byte Sequence	Decimal Value
0x25	37
0x8025	37
0xbbbd	15,293
0xed7f3e7d	226,442,877
0xfaa1a0e403d8	2,893,212,287,960
0xfc8998abc66bc0	151,288,809,941,952
0xfefa318fa8e3ca11	70,423,237,261,249,041
0xffffffffffffffff	18,446,744,073,709,551,615

Table 2: Example Integer Encodings

Variable length integers do not need to be encoded using the minimum number of bytes; any encoding length that can represent the value is valid.

x (vi64): Indicates that x holds an integer value using the variable-length encoding as described above.

1.4.2. Location Structure

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

```
Location {
  Group (vi64),
  Object (vi64)
}
```

Figure 1: Location structure

In this document, a Location can be expressed in the form of {GroupID, ObjectID}, where GroupID and ObjectID indicate the Group ID and Object ID of the Location, respectively. 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)

1.4.3. 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-Pairs encode a Type value as a delta from the previous Type value, or from 0 if there is no previous Type value. This is efficient on the wire and makes it easy to ensure there is only one instance of a type when needed. The previous Type value plus the Delta Type MUST NOT be greater than 2^64 - 1. If a Delta Type is received that would be too large, the Session MUST be closed with a PROTOCOL_VIOLATION.

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 {
  Delta Type (vi64),
  [Length (vi64),]
  Value (..)
}
```

Figure 2: MOQT Key-Value-Pair

* Delta Type: an unsigned integer, encoded as a varint, identifying the Type as a delta encoded value from the previous Type, if any. The Type identifies the type of value and also the subsequent serialization.

* Length: Only present when Type is odd. Specifies the length of the Value field in bytes. 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 close the session with error code KEY_VALUE_FORMATTING_ERROR.

Key-Value-Pairs are always parsed with a known byte length, which bounds the sequence. The source of this length varies by context.

1.4.4. 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 (vi64),
  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 value 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.

1.5. Representing Namespace and Track Names

There is often a need to render namespace tuples and track names for purposes such as logging, representing track filenames, or use in certain authorization verification schemes. The namespace and track name are binary, so they need to be converted to a safe form.

The following format is RECOMMENDED:

* Each of the namespace tuples are rendered in order with a hyphen (-) between them followed by the track name with a double hyphen (--) between the last namespace and track name.

* Bytes in the range a-z, A-Z, 0-9 as well as _ (0x5f) are output as is, while all other bytes are encoded as a period (.) symbol followed by exactly two lower case hex digits.

The goal of this format is to have a format that is both filename and URL safe. It allows many common names to be rendered in an easily human readable form while still supporting binary values.

1.5.1. Parsing Serialized Names

When parsing a serialized namespace or track name back to its binary form, implementations MUST apply the following rules to ensure a canonical encoding:

* The hex digits following a period (.) MUST be lowercase (a-f). Uppercase hex digits (A-F) are invalid and MUST cause parsing to fail.

* Bytes that can be represented literally (a-z, A-Z, 0-9, _) MUST NOT appear in their hex-encoded form. For example, .61 is invalid because a must be represented as the literal character a. A parser MUST reject such redundant encodings.

* A period (.) MUST be followed by exactly two hex digits. A trailing period or a period followed by fewer than two hex digits is invalid.

These rules ensure that the encoding is bijective: every binary value has exactly one valid serialized representation, and every valid serialized string maps to exactly one binary value. This property simplifies comparison of serialized names without requiring full deserialization.

Implementations that receive an invalid serialized name SHOULD treat it as an error. The specific error handling behavior is application-defined.

Example:

```
example.2enet-team2-project_x--report
  Namespace tuples: (example.net, team2, project_x)
  Track name: report
```

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 9). 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 in ascending order by Object ID.

From the perspective of a subscriber or a cache, an Object can be in three possible states:

1. The Object is known to not exist. This state is permanent. All signals that an Object does not exist are authoritative.

2. The Object is known to exist. From this state, it can transition to not existing, but not vice versa.

3. The state of the Object is unknown, either because it has not yet been received, or it has not been produced yet.

Since Objects can be delivered out of order, an endpoint can receive an Object after it has already recorded that the Object does not exist (e.g., via a FETCH gap from one source and later delivery via a subscription). This is not a protocol error and the Track is not malformed.

Whenever the publisher communicates that certain objects do not exist, this fact is expressed as a contiguous range of non-existent objects and by including Properties indicating the group/object gaps; MOQT implementers should take that into account when selecting appropriate data structures.

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 an Object's forwarding preference (see Section 11.2.1.2) is "Datagram", it is not sent in Subgroups, does not belong to a Subgroup in any way, 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 QUIC and WebTransport implementations offer the ability to control the relative scheduling priority of pending stream data.

Every Object within a Group belongs to exactly one Subgroup or Datagram.

When Objects are sent in a subscription (see Section 5.1), Objects from two subgroups MUST NOT be sent on the same stream, and 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.

When the Original Publisher opens a new subgroup, it MUST set the FIRST_OBJECT bit (Section 11.4.2) to indicate that the first object in the subgroup stream is the first object ever published in that subgroup. A relay forwarding a subgroup that begins with the first object ever published in that subgroup MUST set the FIRST_OBJECT bit.

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 they could have an unexpected delivery order if Group IDs do not increase with time.

The amount of time elapsed between publishing an Object in Group ID N and in a Group ID > N, or even which will be published first, is not defined by this specification and is defined by the applications using MOQT.

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 set of between 0 and 32 Track Namespace Fields, encoded as follows:

```
Track Namespace {
  Number of Track Namespace Fields (vi64),
  Track Namespace Field (..) ...
}
```

* Number of Track Namespace Fields: A variable-length integer specifying the number of Track Namespace Fields in the Track Namespace.

Each Track Namespace Field is encoded as follows:

```
Track Namespace Field {
  Track Namespace Field Length (vi64),
  Track Namespace Field Value (..)
}
```

* Track Namespace Field Length: A variable-length integer specifying the length of the Track Namespace Field in bytes.

* Track Namespace Field Value: A sequence of bytes that forms a Track Namespace Field.

Each Track Namespace Field Value MUST contain at least one byte. If an endpoint receives a Track Namespace Field with a Track Namespace Field Length of 0, it MUST close the session with a PROTOCOL_VIOLATION.

The structured nature of Track Namespace allows relays and applications to manipulate prefixes of a namespace. If an endpoint receives a Track Namespace consisting of greater than 32 Track Namespace Fields, it MUST close the session with a PROTOCOL_VIOLATION.

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

The maximum total length of a Full Track Name is 4,096 bytes. The length of a Full Track Name is computed as the sum of the Track Namespace Field Length fields and the Track Name Length field. The length of a Track Namespace is the sum of the Track Namespace Field Length fields. If an endpoint receives a Track Namespace or a Full Track Name exceeding 4,096 bytes, it MUST close the session with a PROTOCOL_VIOLATION.

In this specification, both the Track Namespace Fields and the Track Name are not constrained to a specific encoding. They carry a sequence of bytes and comparison between two Track Namespace 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 Fields or Track Name such that exact comparison works.

2.4.2. 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 with a particular Subgroup ID is received, but its Publisher Priority is different from that of the previous Object with the same Subgroup ID.

2. 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.

3. A Subgroup is received over multiple transport streams terminated by FIN with different final Objects.

4. 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 before a FIN in a Subgroup which has the END_OF_GROUP bit set. If the end of a Group is implicitly determined via a gap in a FETCH response, the final Object in the Group remains unknown.

5. An Object is received 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.

6. The same Object is received more than once with different Payload or other immutable properties.

7. An Object is received with a different Forwarding Preference than previously observed.

The above list of conditions is not considered exhaustive.

When a subscriber detects a Malformed Track, it MUST cancel any corresponding subscription or fetches for that Track from that publisher (see Section 3.3.2), 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. Object(s) triggering Malformed Track status MUST NOT be cached.

2.4.3. Scope

An MOQT scope is a set of servers (as identified by their connection URIs) for which a Full Track Name is 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.

A single MOQT transport session is tied to the scope that is negotiated in the beginning of the session. Unless the application has additional information, two tracks are assumed to belong to the same scope if and only if the authority and the path values are equal. The authority and the path values are communicated through the CLIENT_SETUP message in case of raw QUIC, and through HTTP request header fields in case of WebTransport.

Because each Full Track Name is 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 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.

2.5. Properties

Tracks and Objects can have additional relay-visible fields, known as Properties, which do not require negotiation, and can be used to alter MOQT Object distribution.

Properties are defined in Section 12 as well as external specifications and are registered in an IANA table Section 15. These specifications define the type and value of the property, along with any rules concerning processing, modification, caching and forwarding.

If a Relay does not support a Property, it MUST NOT be modified, MUST be forwarded, and MUST be cached with the Track or Object, unless it is a Mandatory Track Property as described in Section 2.5.1. If a Track or Object arrives with a different set of unknown properties than previously cached, the most recent set SHOULD replace any cached values, removing any unknown values not present in the new set. Relays MUST NOT attempt to merge sets of unknown properties received in different messages.

If a Relay supports a Property, it MAY be modified, added, removed, and/or cached, subject to the processing rules specified in the definition.

Properties are serialized as Key-Value-Pairs (see Figure 2). Track Properties always appear as the final field in the messages that carry them; their length is the remaining bytes of the message after all preceding fields have been consumed. Object Properties (Section 11.2.1.2) are preceded by an explicit length field.

Property types are registered in the IANA table 'MOQ Properties'. See Section 15.

Certain Property type ranges are reserved for application-specific use and will never be allocated by IANA in future MOQT specifications:

* 0x38 to 0x3F (1-byte encoding): 8 code points for applications with tight space constraints

* 0x3800 to 0x3FFF (2-byte encoding): 2048 code points (including grease Section 14) for applications with moderate space constraints

Applications MAY use code points in these ranges without registration for format-specific metadata or other application-defined purposes. Relays that do not understand the application format MUST forward these properties unchanged but MUST NOT attempt to interpret their semantic meaning. Different applications using the same code point in these ranges may assign different meanings; the interpretation depends on the track or application context known to the publisher and subscriber.

2.5.1. Mandatory Track Properties

Property types in the range 0x4000-0x7FFF are designated as Mandatory Track Properties. These properties MUST have Track scope. Mandatory Track Properties have special handling rules that prevent tracks with required extensions from being forwarded to or processed by endpoints that do not understand them.

An Object received with a Mandatory Track Property as an Object Property is malformed (see Section 2.4.2).

When an endpoint receives Track Properties (in PUBLISH, SUBSCRIBE_OK, or FETCH_OK messages) containing a Mandatory Track Property type that it does not understand, it MUST NOT process or forward that track:

* For PUBLISH messages: the subscriber MUST respond with REQUEST_ERROR with error code UNSUPPORTED_EXTENSION.

* For SUBSCRIBE_OK messages: the subscriber MUST cancel the subscription (see Section 3.3.2). If the subscriber is a relay with pending downstream subscribers, it MUST send REQUEST_ERROR with error code UNSUPPORTED_EXTENSION to the downstream subscribers.

* For FETCH_OK messages: the subscriber MUST cancel the fetch (see Section 3.3.2). If the subscriber is a relay and has not yet sent a FETCH_OK or REQUEST_ERROR downstream, it MUST send REQUEST_ERROR with error code UNSUPPORTED_EXTENSION to the downstream fetch requester. If the relay has already forwarded data on a fetch stream, it MUST reset the stream.

A publisher that knows a subscriber does not support a Mandatory Track Property SHOULD take the following action:

* For SUBSCRIBE: respond with REQUEST_ERROR with error code UNSUPPORTED_EXTENSION.

* For FETCH: respond with REQUEST_ERROR with error code UNSUPPORTED_EXTENSION.

* For PUBLISH: do not publish the track to that subscriber.

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 ([RFC9221]) MUST be supported and negotiated in the QUIC connection used for MOQT, which is already a requirement for WebTransport over HTTP/3.

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

MOQT uses ALPN in QUIC and "WT-Available-Protocols" in WebTransport ([WebTransport], Section 3.3) to perform version negotiation.

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

The ALPN value [RFC7301] for the final version of this specification is moqt. ALPNs used to identify IETF drafts are created by appending the draft number to "moqt-". For example, draft-ietf-moq-transport-13 would be identified as "moqt-13".

Note: Draft versions prior to -15 all used moq-00 ALPN, followed by version negotiation in the SETUP messages.

3.1.1. MOQT URI Scheme

An MOQT server is identified using a URI with the "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].

The moqt URI scheme follows the generic URI syntax of [RFC3986] for the authority, path-abempty, and query components, including the use of reserved characters and percent-encoding defined therein. A moqt URI can be converted to an https URI by replacing the scheme (see Section 3.1.3), so the path-abempty and query components use the same syntax as https URIs.

3.1.2. Fragment Identifiers

The media type for resources identified by moqt URIs is application/moqt (see Section 15.2).

Fragment identifiers MAY be used with moqt URIs. The fragment is not transmitted to the server; it is processed locally by the client after establishing the MOQT session.

A moqt URI fragment MUST begin with a registered fragment type identifier, followed by a colon (:), followed by a type-specific value:

```
moqt://example.com/app#<type>:<value>
```

Fragment type identifiers MUST consist of ASCII lowercase letters, digits, and hyphens (a-z, 0-9, -). The semantics of the value after the colon are defined by the specification that registers the fragment type.

Fragment type identifiers are registered in the "MOQT URI Fragment Types" registry (Section 15.3).

The default operation for dereferencing a moqt URI is to establish a MOQT session to the identified server.

TODO: Add URI scheme security considerations per RFC 7595 Section 3.7 (e.g., authority in SNI, path/query exposure).

TODO: Add internationalization statement per RFC 7595 Section 3.6.

If the port is omitted in the URI, a default port of 443 is used.

The client MAY use either native QUIC or WebTransport. On a QUIC connection, the client offers any combination of MOQT ALPNs (e.g. moqt/1, moqt/2) and h3 that it supports in its TLS ClientHello, in preference order. If the server selects an MOQT ALPN, the session proceeds as described in Section 3.1.4. If the server selects h3, the client establishes a WebTransport session as described in Section 3.1.3. On a TCP+TLS connection, the client offers h2 in its TLS ClientHello and establishes a WebTransport session as described in Section 3.1.3.

3.1.3. WebTransport

When the client uses WebTransport, it constructs an https URI from the moqt URI by replacing the scheme with https. For example, moqt://example.com/path becomes https://example.com/path. The client sends an extended CONNECT request to this URI to establish a WebTransport session, as described in ([WebTransport], Section 3). The client includes MOQT protocol identifiers in the WT-Available-Protocols header ([WebTransport], Section 3.3).

3.1.4. Native QUIC

The client establishes a QUIC connection to the host and port identified by the authority section of the URI. When the client uses native QUIC, the authority, path-abempty and query portions of the URI are transmitted in Setup Options (see Section 10.3.1).

3.1.5. 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. Extension Negotiation

Endpoints use the exchange of Setup messages to negotiate MOQT extensions. Extensions can define new Message types, new Parameters, new Properties, or new framing for Streams and Datagrams.

The client and server MUST include all Setup Options Section 10.3.1 required for the negotiated MOQT version in SETUP.

Each endpoint declares the extensions it supports and provides any initial values required by those extensions as Setup Options in SETUP. Once an endpoint has both sent and received SETUP messages, it determines the set of negotiated extensions.

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.

3.2.1. Reserved Namespaces

MOQT reserves all Track Namespace values whose first tuple field begins with a period (0x2e, .). These namespaces MUST NOT be used unless their meaning is defined through IANA registration. Unless otherwise specified, an endpoint that receives a request for an unrecognized reserved namespace MUST pass it to the Application, so that future extensions can define new reserved namespaces without breaking older implementations.

A Track Namespace whose first field is exactly . (a single period, 0x2e) is reserved and MUST NOT be used for any purpose; endpoints MUST NOT publish tracks or namespaces under it and MUST reject requests referencing it with DOES_NOT_EXIST.

3.2.2. Session-Level Tracks and Namespaces

MOQT defines the .session namespace (the bytes 0x2e, 0x73, 0x65, 0x73, 0x73, 0x69, 0x6f, 0x6e) in the first position of the Track Namespace for session-level tracks and namespaces. Session-level tracks and namespaces are managed by the MOQT implementation, not the Application. They provide a mechanism for extending MOQT transport functionality using existing subscription and object delivery machinery, without defining new control messages or stream types.

The Application MUST NOT publish tracks or namespaces whose first field is .session. Relays MUST NOT forward requests for session-level tracks and namespaces to other sessions.

The empty track name in the .session namespace is defined to not exist. A request with a Track Namespace whose first field is .session and an empty Track Name MUST be rejected with DOES_NOT_EXIST.

An endpoint that receives a request for an unrecognized session-level track or namespace MUST reject it with REQUEST_ERROR using error code DOES_NOT_EXIST rather than passing it to the Application.

The track names and namespaces available under the .session namespace are defined by extensions to this specification and registered with IANA (see Section 15.9).

3.3. Session initialization

MOQT uses a pair of unidirectional streams for creating the session and exchanging control messages. Each peer opens one control stream beginning with a SETUP message. Using a pair of unidirectional streams rather than a single bidirectional stream allows either peer to send data as soon as it is able. Depending on whether 0-RTT is available on the QUIC connection, either client or server might be able to send stream data first.

In addition to the control streams, this specification uses bidirectional streams to carry requests. A request stream begins with one of these seven message types: TRACK_STATUS, SUBSCRIBE, PUBLISH, FETCH, PUBLISH_NAMESPACE, SUBSCRIBE_NAMESPACE, and SUBSCRIBE_TRACKS. Bidirectional streams MUST NOT begin with any other message type unless negotiated. If they do, the peer MUST close the Session with a PROTOCOL_VIOLATION. Objects are sent on unidirectional streams.

As such, a client can initiate a MOQT session, subscribe, and start publishing Objects all in parallel.

Unidirectional streams containing Objects or bidirectional stream(s) beginning with a request message could arrive prior to the control streams, in which case the data SHOULD be buffered until both control streams arrive and setup is complete. If an implementation does not want to buffer or if the message type is not supported, it MAY reset such bidirectional streams before the session and control streams are established.

A control stream MUST NOT be closed at the underlying transport layer during the session’s lifetime. Doing so results in the session being closed as a PROTOCOL_VIOLATION.

Prior to receiving the peer’s SETUP message, it’s unknown what extensions a peer will support. Message Parameters requiring negotiation SHOULD NOT be used prior to receiving the peer’s SETUP message unless the application requires the extension or the endpoint knows the peer supports the extension. If an unsupported Message Parameter is used, the peer will be unable to process it and the session will be terminated. See Section 10.2.

3.3.1. 0-RTT

QUIC supports 0-RTT (Section 2.3 of [RFC8446]), but WebTransport over QUIC is not expected to use 0-RTT, because initializing a WebTransport session uses CONNECT, which is not a safe method. [RFC8470] describes the use of 0-RTT with HTTP in more detail. If 0-RTT is used with an existing or future version of WebTransport, the following would apply to it as well as QUIC.

MOQT Messages and Objects as defined in this draft are safe to replay in most circumstances.

* TRACK_STATUS gets the Largest Object and Track Properties, but does not change the state of a Track or any Object in the Track.

* SUBSCRIBE requests Objects be delivered, but does not change the Objects being requested.

* PUBLISH initiates a Subscription. Objects can be immediately sent to the Subscriber. Processing the same Objects multiple times is idempotent, as the subscriber or relay can identify and discard duplicates based on the Group ID and Object ID.

* SUBSCRIBE_NAMESPACE requests a list of namespaces and the establishment of new subscriptions, but does not change the available Namespaces, Tracks, or Objects contained within a Track.

* PUBLISH_NAMESPACE requests that Subscriptions under the namespace be sent to that Publisher. If a Subscription was sent to the replaying endpoint, it would fail because the endpoint cannot complete the handshake.

Some potential side effects of replay are:

* Publishing Objects that were previously published could cause those Objects to be distributed to active Subscriptions if the relays do not identify them as already having been published. This re-distribution could also make them available in cache again after they previously expired.

Replays could increase load on the MOQT network. For relay to client traffic, this is no worse than 0-RTT in HTTP/3, since the server is limited by the amplification factor until address validation. However, it could cause the relay to initiate new upstream Subscriptions. For a SUBSCRIBE_NAMESPACE that requested Subscriptions in the Namespace, sending that upstream could cause the Relay to receive a number of new Subscriptions on the replaying client’s behalf.

Relays MAY defer initiating upstream subscriptions until the handshake is complete or reject 0-RTT entirely to mitigate resource exhaustion from replayed packets.

3.3.2. Request Cancellation and Rejection

Once a request stream has been opened, the request MAY be cancelled by either endpoint. Senders cancel requests if the response is no longer of interest; Receivers cancel requests if they are unable to or choose not to respond.

Implementations SHOULD cancel requests by abruptly terminating any directions of a stream that are still open by resetting or sending STOP_SENDING.

When an endpoint rejects a request without performing any application processing, it SHOULD send a REQUEST_ERROR and FIN the stream.

The application SHOULD use a relevant error code when resetting or sending STOP_SENDING on a request stream, as defined in Section 3.3.3.

3.3.3. Stream Reset Error Codes

The application SHOULD use a relevant error code when resetting or sending STOP_SENDING on any stream.

INTERNAL_ERROR (0x0): An implementation specific error.

CANCELLED (0x1): The stream was cancelled by either endpoint. For Subscriptions, PUBLISH_DONE (Section 10.11) may have a more detailed status code.

DELIVERY_TIMEOUT (0x2): A delivery timeout (Section 8) was exceeded for this stream.

SESSION_CLOSED (0x3): The session is being closed.

GOING_AWAY (0x4): The endpoint is rejecting this request because it has sent or received a GOAWAY.

TOO_FAR_BEHIND (0x5): The corresponding subscription has exceeded the publisher’s resource limits and is being terminated (see Section 8).

UNKNOWN_OBJECT_STATUS (0x6): In response to a FETCH, the publisher is unable to determine the status of the next Object in the requested range.

EXPIRED_AUTH_TOKEN (0x7): The authorization token for the request has expired.

EXCESSIVE_LOAD (0x9): The endpoint is overloaded and is resetting this stream.

MALFORMED_TRACK (0x12): A relay publisher detected that the track was malformed (see Section 2.4.2).

3.4. Unidirectional Stream Types

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

ID	Type
0x05	FETCH_HEADER (Section 11.4.4)
0b0XX1XXXX	SUBGROUP_HEADER (Section 11.4.2)
0x2F00	SETUP (Section 10.3)
0x132B3E28	PADDING (Section 11.5.1)

Table 3

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

Control streams (SETUP) are described in Section 3.3. Data streams (FETCH_HEADER, SUBGROUP_HEADER) are described in Section 11. Padding streams are described in Section 11.5.

3.5. 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 6).

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 endpoint received a Request ID with an incorrect least significant bit for the sender, or a duplicate Request ID. See Section 10.1.

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.

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 10.3.1.2.

GOAWAY_TIMEOUT (0x10): The session was closed because the peer took too long to close the session in response to a GOAWAY (Section 10.4) message. See session migration (Section 3.6).

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 11). 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 10.3.1.3 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 10.2.2).

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 10.2.2).

UNKNOWN_AUTH_TOKEN_ALIAS (0x17): No registered token found for the provided Alias (see Section 10.2.2).

EXPIRED_AUTH_TOKEN (0x18): Authorization token has expired (Section 10.2.2).

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.

3.6. 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 10.4).

The server sends a GOAWAY message, signaling the client to establish a new session and migrate any Established subscriptions. The GOAWAY message optionally contains a new URI for the new session, otherwise the current URI is reused. The GOAWAY message contains a Timeout indicating how long, in milliseconds, the sender intends to wait before closing the session. The sender SHOULD close the session with GOAWAY_TIMEOUT after the indicated 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 unsubscribing from upstream publishers.

After the client receives a GOAWAY, it’s RECOMMENDED that the client waits until there are no more Established 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 Established subscriptions and published namespaces. The client can choose to delay closing the

session if it expects more OBJECTs to be delivered. The sender closes the session with a GOAWAY_TIMEOUT if the peer doesn't close the session within the indicated Timeout.

3.7. 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.

3.7.1. 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.

3.7.2. 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.

Some applications might have APIs to allow sending duplicate data or forward error correction to probe for more bandwidth while also limiting the impact of probing in case it causes packet loss. Subscribers wanting to switch to an alternate representation of a Track can subscribe to it at a lower priority, or subscribe to additional Tracks at the lowest (255) priority to fill the congestion window during probing intervals while minimizing the impact on higher priority media. Publishers can send padding (Section 11.5) to probe for additional bandwidth without requiring additional subscriptions. Network-assisted bandwidth estimation mechanisms such as SCONE [I-D.ietf-scone-protocol] can provide receivers with sustainable bandwidth hints, which subscribers can use to inform track selection decisions and potentially avoid unnecessary probing.

3.7.3. 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 it's congestion window upon detecting loss. In both cases, the large reduction in sending rate might cause issues with latency sensitive applications.

4. Extensibility

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.

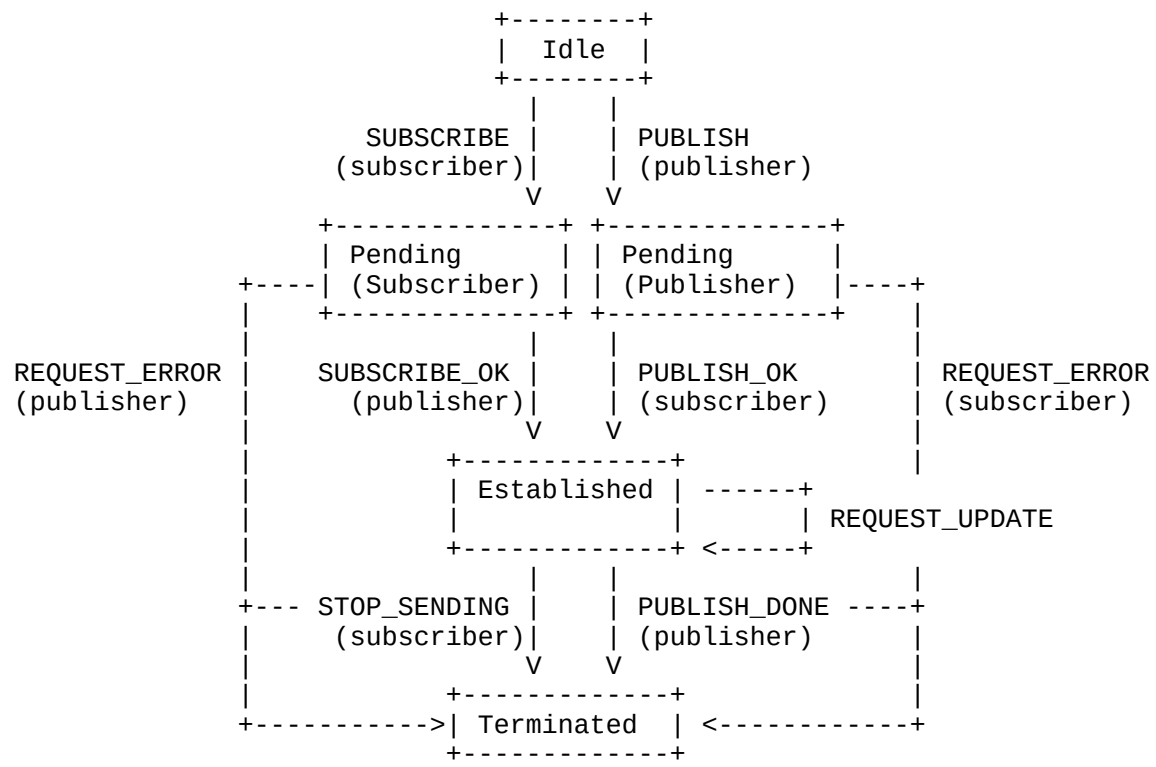
Relays MUST implement all MOQT messages defined in this document, as well as processing rules described in Section 9.

5. Publishing and Retrieving Tracks

5.1. Subscriptions

All subscriptions begin in the Idle state. A subscription can be initiated and moved to the Pending state 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 (Section 10.5) or REQUEST_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 REQUEST_ERROR. Once either of these sequences is successful, the subscription moves to the Established state and can be updated by the subscriber using REQUEST_UPDATE. Either endpoint can terminate an Established subscription, moving it to the Terminated state. The subscriber terminates a subscription in the Pending (Subscriber) or Established states by sending STOP_SENDING. The publisher terminates a subscription in the Pending (Publisher) or Established states by sending PUBLISH_DONE and closing the stream.

This diagram shows the subscription state machine:



A publisher MUST send exactly one SUBSCRIBE_OK or REQUEST_ERROR in response to a SUBSCRIBE. A subscriber MUST send exactly one PUBLISH_OK (Section 10.5) or REQUEST_ERROR in response to a PUBLISH. The peer SHOULD close the session with a protocol error if it receives more than one.

All Established subscriptions have a Forward State which is either 0 or 1. The publisher does not send Objects if the Forward State is 0, and does send them if the Forward State is 1. The initiator of the subscription sets the initial Forward State in either PUBLISH or SUBSCRIBE. The subscriber can send PUBLISH_OK or REQUEST_UPDATE to update the Forward State. Control messages, such as PUBLISH_DONE (Section 10.11) are sent regardless of the forward state.

A publisher MUST save the Largest Location communicated in SUBSCRIBE_OK, PUBLISH or REQUEST_UPDATE_OK that changes the Forward State from 0 to 1. This value is called the Joining Location and can be used in a Joining FETCH (see Section 10.12.2) while the subscription is in the Established state.

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.

For a given Track, an endpoint can have at most one subscription to a Track acting as the publisher and at most one acting as a subscriber. If an endpoint receives a message attempting to establish a second subscription to a Track with the same role, it MUST fail that request with a DUPLICATE_SUBSCRIPTION error.

If a publisher receives a SUBSCRIBE request for a Track with an existing subscription in Pending (publisher) state, it MUST fail that request with a DUPLICATE_SUBSCRIPTION error. If a subscriber receives a PUBLISH for a Track with a subscription in the Pending (Subscriber) state, it MUST ensure the subscription it initiated transitions to the Terminated state before sending PUBLISH_OK.

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

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 REQUEST_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 11.3 and Section 11.4.2).

5.1.1. Subscription State Management

A subscriber keeps subscription state until it cancels the request (see Section 3.3.2), or after receipt of a PUBLISH_DONE or REQUEST_ERROR. Note that PUBLISH_DONE does not usually indicate that state can immediately be destroyed, see Section 10.11.

The Publisher can destroy subscription state as soon as it has received STOP_SENDING. It MUST reset any open streams associated with the SUBSCRIBE.

The Publisher can also immediately delete subscription state after sending PUBLISH_DONE, but MUST NOT send it until it has closed all related streams.

A REQUEST_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.

5.1.2. Subscription Filters

Subscribers can specify a filter on a subscription indicating to the publisher which Objects to send. Subscriptions without a filter pass all Objects published or received via upstream subscriptions.

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

Some filters are defined to be relative to the Largest Object. The Largest Object is the Object with the largest Location (Section 1.4.2) in the Track from the perspective of the publisher processing the message. Largest Object updates when the first byte of an Object with a Location larger than the previous value is published or received through a subscription.

A Subscription Filter has the following structure:

```
Subscription Filter {
  Filter Type (vi64),
  [Start Location (Location),]
  [End Group Delta (vi64),]
}
```

Filter Type can have one of the following values:

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. The specified Start Location MAY be less than the Largest Object observed at the publisher. There is no End Group - the subscription is open ended. An AbsoluteStart filter with Start = {0, 0} is equivalent to an unfiltered subscription.

AbsoluteRange (0x4): The filter Start Location and End Group are specified explicitly. The specified Start Location MAY be less than the Largest Object observed at the publisher. If the specified End Group Delta is zero, the remainder of that Group passes the filter. Otherwise, the last Group ID to be delivered will be the Group ID in Start Location plus the End Group Delta. If the resulting Group ID would be greater than 2^64 - 1, the endpoint MUST close the session with a PROTOCOL_VIOLATION.

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

If the publisher cannot satisfy the requested Subscription Filter (see Section 10.2.9) or if the entire End Group has already been published it SHOULD send a REQUEST_ERROR with code INVALID_RANGE. A publisher MUST NOT send objects from outside the requested range.

5.1.3. Joining an Ongoing Track

The MOQT Object model is designed with the concept that the beginning of a Group is a join point, so in order for a subscriber to join a Track, it needs to request an existing Group or wait for a future Group. Different applications will have different approaches for when to begin a new Group.

To join a Track at a past Group, the subscriber sends a SUBSCRIBE, PUBLISH_OK or REQUEST_UPDATE with Forward State 1 followed by a Joining FETCH (see Section 10.12.2) for the intended start Group, which can be relative. To join a Track at the next Group, the subscriber sends a SUBSCRIBE with Filter Type Next Group Start.

5.1.3.1. Dynamically Starting New Groups

While some publishers will deterministically create new Groups, other applications might want to only begin a new Group when needed. A subscriber joining a Track might detect that it is more efficient to request the Original Publisher create a new group than issue a Joining FETCH. Publishers indicate a Track supports dynamic group creation using the DYNAMIC_GROUPS parameter (Section 12.6).

One possible subscriber pattern is to SUBSCRIBE to a Track using Filter Type Largest Object and observe the Largest Location in the response. If the Object ID is below the application's threshold, the subscriber sends a FETCH for the beginning of the Group. If the Object ID is above the threshold and the Track supports dynamic groups, the subscriber sends a REQUEST_UPDATE message with the NEW_GROUP_REQUEST parameter equal to the Largest Location's Group, plus one (see Section 10.2.13).

Another possible subscriber pattern is to send a SUBSCRIBE with Filter Type Next Group Start and NEW_GROUP_REQUEST equal to 0. The value of DYNAMIC_GROUPS in SUBSCRIBE_OK will indicate if the publisher supports dynamic groups. A publisher that does will begin the next group as soon as practical.

5.2. Fetch State Management

The publisher MUST send exactly one FETCH_OK or REQUEST_ERROR in response to a FETCH.

A subscriber keeps FETCH state until it cancels the request (see Section 3.3.2), receives REQUEST_ERROR, or the FETCH data stream receives a FIN or is reset. If the data stream is already open, the subscriber wishing to cancel the FETCH MAY send STOP_SENDING for the data stream as well as the bidi request stream. It MUST send STOP_SENDING for the bidi request stream.

The Publisher can destroy fetch state as soon as it has received a STOP_SENDING. It MUST reset the bidi request stream and unidirectional data stream associated with the FETCH. It can also destroy state after closing the FETCH data stream.

It can destroy all FETCH state after closing the data stream with a FIN.

A REQUEST_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.

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, SUBSCRIBE_TRACKS, 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 10.

6.1. Subscribing to Namespaces

If the subscriber is aware of a namespace of interest, it can send SUBSCRIBE_NAMESPACE or SUBSCRIBE_TRACKS to publishers/relays it has established a session with.

SUBSCRIBE_NAMESPACE requests namespace discovery: the publisher sends relevant NAMESPACE and NAMESPACE_DONE messages for namespaces matching the prefix, including echoing back Track Namespaces under the prefix that have been published to it.

SUBSCRIBE_TRACKS requests track subscriptions: the publisher sends PUBLISH messages for tracks within matching namespaces, excluding tracks published by the subscriber.

Either message with zero Track Namespace fields indicates the sender is interested in all namespaces or all tracks from the receiver, respectively.

The subscriber sends SUBSCRIBE_NAMESPACE or SUBSCRIBE_TRACKS on a new bidirectional stream and the publisher MUST send a single REQUEST_OK or REQUEST_ERROR as the first message on the bidirectional stream in response.

If a Subscription cannot be created because there are no available bidirectional streams, the Publisher sends a PUBLISH_BLOCKED message on the SUBSCRIBE_TRACKS response stream to indicate the Full Track Name of the Subscription that could not be established. The Publisher MUST NOT send a PUBLISH for a Track after PUBLISH_BLOCKED has been sent. The subscriber can instead issue a SUBSCRIBE to establish a subscription to that track.

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

A SUBSCRIBE_NAMESPACE or SUBSCRIBE_TRACKS can be cancelled by closing the stream with either a FIN or RESET_STREAM. Cancelling SUBSCRIBE_TRACKS does not prohibit original publishers from sending further 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 NAMESPACE message to any subscriber that has sent 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 REQUEST_OK or REQUEST_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 REQUEST_OK or REQUEST_ERROR as the first message on the bidi stream 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 is withdrawn by cancelling the request (see Section 3.3.2), although it is not a protocol error for the subscriber to send a SUBSCRIBE or FETCH message for a track in a namespace after the namespace is withdrawn.

A subscriber can cancel the request (see Section 3.3.2) to revoke acceptance of a PUBLISH_NAMESPACE. If the reason for cancellation is expiration of authorization credentials, the publisher can send PUBLISH_NAMESPACE again on a new bidi stream with refreshed authorization, or close the stream and discard associated state.

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

MOQT 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 with forwarding 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 numberis 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 REQUEST_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. A default can be specified in the parameters of PUBLISH, or SUBSCRIBE_OK. Publisher priority can also be specified in a subgroup header or datagram (see Section 11).

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 omitting the Group Order parameter). 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 have the same subscriber and publisher priority and belong to the same group of the same track, the one with *the lowest Subgroup ID* (for objects with forwarding preference Subgroup), or *the lowest Object ID* (for objects with forwarding preference Datagram) is scheduled to be sent first. If the two objects have different Forwarding Preferences the order is implementation dependent.

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.

A publisher might not utilize the entire available congestion window, session flow control, or all available streams for lower priority Objects if it expects higher priority Objects will be available to send in the near future or it wants to reserve some bandwidth for control messages.

Given the critical nature of control messages and their relatively small size, the control streams SHOULD be prioritized highest, followed by the bidi request streams and then all subscribed Objects. Bidi request streams MAY be prioritized within themselves by Subscriber Priority if specified.

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.

MOQT 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 MOQT 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. Delivery Timeouts and Data Reliability

Each MOQT subscription has two timeout values associated with it: a SUBGROUP_DELIVERY_TIMEOUT and an OBJECT_DELIVERY_TIMEOUT. Both of those values are expressed in milliseconds; both are optional; a value of 0 means that there is no timeout set.

The publisher communicates both timeout values as a Track Property; the subscriber communicates them as Message Parameters. For each type of timeout, if both the publisher and the subscriber have a non-zero value, the smaller of the two is used.

If the OBJECT_DELIVERY_TIMEOUT is not zero, the MOQT implementation MUST retain the time at which the first payload byte of every object has been either received from the upstream subscription, or provided by the original publisher application. The actual mechanism by which the timeout works depends on the Object Forwarding Preference:

* For subgroups, the implementation MUST check the time elapsed since the first byte of the object before attempting to pass it to the underlying transport for transmission; if the time elapsed exceeds OBJECT_DELIVERY_TIMEOUT, it MUST reset the underlying transport stream with the reset stream code DELIVERY_TIMEOUT (see Section 11.4.3) and SHOULD NOT attempt to open a new stream to deliver additional Objects in that Subgroup. The implementation SHOULD check object delivery timeouts before retransmitting object data if the underlying transport implementation allows that. The implementations SHOULD minimize the amount of data buffered at the underlying transport layer, as any data buffered at this layer can no longer be timed out, potentially leading to transmission of expired data.

* For datagrams, the implementation MUST drop the datagrams if the time elapsed since the first byte exceeds OBJECT_DELIVERY_TIMEOUT. Similar to subgroups, implementations SHOULD either minimize datagram queueing, or use datagram queueing mechanisms that support time bounds (such as the outgoingMaxAge parameter in the W3C WebTransport API).

If the Object Forwarding Preference is Subgroup and the value of SUBGROUP_DELIVERY_TIMEOUT is not zero, the MOQT implementation MUST start a timer of SUBGROUP_DELIVERY_TIMEOUT duration once it becomes aware that all of the objects on the subgroup have been published (either by receiving a FIN from the upstream subscription, or, in

case of the original publisher, through being notified of this fact by the application). If the timer expires before the underlying transport stream reaches "all data committed" state ([I-D.ietf-webtrans-overview], Section 4.3), the implementation MUST reset the stream. This ensures that MOQT can time out subgroups where all of the data has been sent but not yet fully delivered due to packet loss.

For objects with Object Forwarding Preference set to Datagram, the SUBGROUP_DELIVERY_TIMEOUT acts the same way as OBJECT_DELIVERY_TIMEOUT; if both are non-zero, the smaller of the two is used.

	SUBGROUP_DELIVERY_TIMEOUT	OBJECT_DELIVERY_TIMEOUT
Timeout starts	When the FIN for the subgroup is received	When the first byte of the object is received
Timeout checked at	Via a timer until all data is acknowledged	When the object is sent to the underlying transport
Action upon timeout	Reset for subgroups, drop for datagrams	Reset for subgroups, drop for datagrams

Table 4: Comparison of the delivery timeout mechanisms

Publishers can, at their discretion, discontinue forwarding Objects before either of the timeouts occurs, subject to stream closure and ordering constraints described in Section 11.4.3. However, if none of the timeouts are set to a non-zero value, 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 using PUBLISH_DONE with error TOO_FAR_BEHIND.

9. Relays

Relays are leveraged to enable distribution scale in the MOQT 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 MOQT Object metadata.

9.1. 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 can transition from existing to not existing in cases where the object is no longer available.

2. Object Properties can be added, removed or updated, subject to the constraints of the specific property.

An endpoint that receives a duplicate Object with a different Forwarding Preference, Subgroup ID, Priority or Payload MUST treat the track as Malformed.

For ranges of objects that do not exist, relays MAY change the representation of a missing range to a semantically equivalent one. For instance, a relay may change an End-of-Group="Y" Subgroup Header to an equivalent object with an End of Group status, or a Prior Group ID Gap property could be removed in FETCH, where it's redundant.

As described in Section 2.1, an endpoint can receive an Object after it has already recorded that the Object does not exist. A caching relay SHOULD NOT cache or forward the Object in this case.

A cache MUST store all fields of an Object defined in Section 11.2.1, with the exception of any Object Properties (Section 11.2.1.2) that specify otherwise.

9.2. Forward Handling

Relays SHOULD set the Forward flag to 1 when 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.

9.3. 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 REQUEST_ERROR or unsubscribing (see Section 3.3.2) 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 matching Established subscriptions. The Relay SHOULD attempt to deduplicate Objects before forwarding, subject to implementation constraints.

9.4. Subscriber Interactions

Subscribers request Tracks by sending a SUBSCRIBE (see Section 10.7) or FETCH (see Section 10.12) 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. Relays MUST follow the constraints on LARGEST_OBJECT defined in Section 10.2.11.

Publishers maintain a list of Established downstream subscriptions for each Track. Relays use the Track Alias (Section 11.1) of an incoming Object to identify its Track and find the current subscribers. Each new Object belonging to the Track is forwarded to each subscriber, as allowed by the subscription’s filter (see Section 10.7), and delivered according to the priority (see Section 7) and delivery timeout (see Section 8).

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 MOQT restricts widening 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 10.8). 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.

9.4.1. Graceful Subscriber Relay Switchover

This section describes a behavior that a Subscriber MAY implement to improve user experience when a relay sends a GOAWAY or the Subscriber switches between networks, such as WiFi to Cellular, and QUIC Connection Migration is not possible.

When a subscriber receives the GOAWAY message, it starts the process of connecting to a new relay and sending the SUBSCRIBE requests for all Established 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 cancelled (see Section 3.3.2).

9.5. 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. Relays MUST NOT assume that an authorized publisher of a single Track is

implicitly authorized to publish any other Tracks or Track Namespaces. If a Publisher would like Subscriptions in a Namespace routed to it, it MUST send an explicit PUBLISH_NAMESPACE. 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 cancels the request (see Section 3.3.2) to withdraw the PUBLISH_NAMESPACE. A subscriber indicates it will no longer subscribe to Tracks in a namespace it previously responded PUBLISH_NAMESPACE_OK to by cancelling the PUBLISH_NAMESPACE request.

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 fields 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 send SUBSCRIBE messages to all matching publishers. This includes matching both Established subscriptions on the Full Track Name and Namespace Prefix Matching against published Namespaces. Relays MUST forward 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 Established 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 REQUEST_UPDATE with Forward=1 to all publishers. If there are no Established 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 sent SUBSCRIBE_TRACKS for the Track's namespace or a prefix thereof. However, if the relay is holding a downstream SUBSCRIBE awaiting a publisher for this Track (see Section 10.2.6), it MUST proceed with the SUBSCRIBE and MUST NOT also forward the PUBLISH to that subscriber.

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 Established downstream subscriptions, 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.

9.5.1. Graceful Publisher Relay Switchover

This section describes a behavior that a publisher MAY implement to improve user experience when a relay sends a GOAWAY or the publisher switches between networks, such as WiFi to Cellular, and QUIC Connection Migration is not possible.

A new Session is established, to a new URI if specified in a GOAWAY. The publisher sends PUBLISH_NAMESPACE and/or PUBLISH messages to begin publishing on the new Session, but it does not immediately stop publishing Objects on the old Session.

Once the subscriptions have migrated over to the new session, the publisher can stop publishing Objects on the old session. 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.

9.6. Relay Track Handling

A relay MUST include all Properties associated with a Track when sending any PUBLISH, SUBSCRIBE_OK, TRACK_STATUS_OK, or FETCH_OK, unless allowed by the property's specification (see Section 2.5).

9.7. Relay Object Handling

MOQT encodes the delivery information in the Object header (Section 11.2.1). A relay MUST NOT modify Object fields when forwarding, except for Object Properties as specified in Section 2.5.

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.

10. Control Messages

MOQT uses a pair of unidirectional streams to exchange control messages, as defined in Section 3.3. Every message on a control or request stream is formatted as follows:

```
MOQT Control Message {
  Message Type (vi64),
  Message Length (16),
  Message Payload (..),
}
```

Figure 3: MOQT Control Message

The following Message Types are defined. The Stream column indicates which stream type each message is sent on: Control indicates the control stream (Section 3.3), and Request indicates a bidirectional request stream. Messages marked "First" MUST be the first message on a new request stream.

ID	Messages	Stream
0x01	RESERVED (SETUP for version 00)	
0x40	RESERVED (CLIENT_SETUP for <= 10)	

0x41	RESERVED (SERVER_SETUP for <= 10)	
0x20	RESERVED (CLIENT_SETUP in <= 16)	
0x21	RESERVED (SERVER_SETUP in <= 16)	
0x2F00	SETUP (Section 10.3)	Control
0x10	GOAWAY (Section 10.4)	Control, Request
0x3	SUBSCRIBE (Section 10.7)	Request, First
0x4	SUBSCRIBE_OK (Section 10.8)	Request
0x1D	PUBLISH (Section 10.10)	Request, First
0x1E	PUBLISH_OK (Section 10.5)	Request
0xB	PUBLISH_DONE (Section 10.11)	Request
0x16	FETCH (Section 10.12)	Request, First
0x18	FETCH_OK (Section 10.13)	Request
0xD	TRACK_STATUS (Section 10.14)	Request, First
0x6	PUBLISH_NAMESPACE (Section 10.15)	Request, First
0x50	SUBSCRIBE_NAMESPACE (Section 10.18)	Request, First
0x51	SUBSCRIBE_TRACKS (Section 10.19)	Request, First
0x8	NAMESPACE (Section 10.16)	Request
0xE	NAMESPACE_DONE (Section 10.17)	Request
0xF	PUBLISH_BLOCKED (Section 10.20)	Request
0x2	REQUEST_UPDATE (Section 10.9)	Request
0x7	REQUEST_OK (Section 10.5)	Request
0x5	REQUEST_ERROR (Section 10.6)	Request

Table 5

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 a PROTOCOL_VIOLATION.

10.1. Request ID

Request ID is included in request messages and is used to identify requests across messages. For example, Joining Fetch references the Request ID of a SUBSCRIBE.

The client generates even numbered Request IDs, starting at 0, and the server generates odd numbered Request IDs, starting at 1. Each endpoint increments its Request ID by 2 for each new request.

Each SUBSCRIBE, PUBLISH, FETCH, SUBSCRIBE_NAMESPACE, SUBSCRIBE_TRACKS, PUBLISH_NAMESPACE, REQUEST_UPDATE, and TRACK_STATUS message consumes a Request ID. Only request messages include a Request ID; response messages do not, since they are sent on the same bidirectional stream as the request.

If an endpoint receives a Request ID where the least significant bit is incorrect for the sender, or a duplicate Request ID, it MUST close the session with INVALID_REQUEST_ID.

10.2. Message Parameters

Some control messages include a field that encodes optional Message Parameters. Message Parameters are serialized as follows:

```
Message Parameter {
  Type Delta (vi64),
  Value (..)
}
```

Figure 4: Message Parameter

Type Delta: The difference between this Parameter Type and the previous Parameter Type in the message, or the Parameter Type itself for the first parameter. Parameters MUST be serialized in ascending order by Type. If the resulting Type would be greater than 2^64 - 1, the endpoint MUST close the session with a PROTOCOL_VIOLATION.

* Value: The encoding is specified by each parameter definition. The encodings defined in this draft are:

- uint8: A single-byte unsigned integer (0-255)
- varint: A variable-length integer
- Location: Two consecutive varints (Group, Object)
- Length-prefixed: A varint length followed by that many bytes

Message Parameters are intended for the peer only and are not forwarded by Relays, though relays can consider received parameter values when making a request.

All Message Parameters MUST be defined in the negotiated version of MOQT or negotiated via Setup Options. An endpoint that receives an unknown Message Parameter MUST close the session with PROTOCOL_VIOLATION. Because the receiver has to understand every Message Parameter, there is no need for a mechanism to skip unknown parameters. Because unknown parameters cannot be skipped, the block is bounded by a parameter count rather than a length.

The Message Parameter types defined in this version of MOQT are listed below.

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 unexpected duplicate parameters and close the session with PROTOCOL_VIOLATION if found.

The number of Message Parameters is not specifically limited, but the total length of a control message is limited to 2^16-1 bytes.

Message 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 2.4.3.

10.2.1. Parameter Scope

Message Parameters are always intended for the peer endpoint only and are not forwarded by Relays, though relays can consider received parameter values when making a request. Track information not specific to the Message or Session is encoded in Track Properties. See Section 2.5.

Each Message Parameter definition indicates the message types in which it can appear. If it appears in some other type of message, the receiving endpoint MUST close the connection with a

PROTOCOL_VIOLATION. Note that since Setup Options use a separate namespace, it is impossible for Message Parameters to appear in Setup messages.

10.2.2. AUTHORIZATION TOKEN Parameter

The AUTHORIZATION TOKEN parameter (Parameter Type 0x03) uses Length-prefixed encoding. It MAY appear in a PUBLISH, SUBSCRIBE, REQUEST_UPDATE, SUBSCRIBE_NAMESPACE, SUBSCRIBE_TRACKS, PUBLISH_NAMESPACE, TRACK_STATUS or FETCH message. This parameter conveys information to authorize the sender to perform the operation carrying the parameter.

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 (vi64),
  [Token Alias (vi64),]
  [Token Type (vi64),]
  [Token Value (..)]
}
```

Figure 5: 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:

DELETE (0x0): 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.

REGISTER (0x1): 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.

USE_ALIAS (0x2): There is an Alias but no Type or Value. Use the Token Type and Value previously registered with this Alias.

USE_VALUE (0x3): 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.

If a server receives Alias Type DELETE (0x0) or USE_ALIAS (0x2) in a SETUP message, it MUST close the session with a PROTOCOL_VIOLATION.

* 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 endpoint 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 15). 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. However, it is important to not store an error code for a token that might be valid in the future or due to some other property becoming fulfilled which currently isn’t. 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 Option defining the MAX_AUTH_TOKEN_CACHE_SIZE limit (see Section 10.3.1.3) 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.

The AUTHORIZATION TOKEN parameter MAY be repeated within a message as long as the combination of Token Type and Token Value are unique after resolving any aliases.

Messages carrying the AUTHORIZATION TOKEN parameter can appear on different control streams. Because stream processing order can be different than send order, the receiver and sender can have inconsistent views of the token cache state.

Senders MUST NOT send USE_ALIAS on one control stream for an alias registered on a different stream until the sender has received a response to the message containing the REGISTER. Senders MAY use USE_ALIAS on the same control stream as the REGISTER without waiting for a response.

Senders MUST NOT send DELETE for an alias while any message using USE_ALIAS with that alias has not received a response.

10.2.3. SUBGROUP_DELIVERY_TIMEOUT Parameter

The SUBGROUP_DELIVERY_TIMEOUT parameter (Parameter Type 0x06) is a varint. It MAY appear in a PUBLISH_OK, SUBSCRIBE, or REQUEST_UPDATE message. Its semantics are defined in Section 8.

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.

10.2.4. OBJECT_DELIVERY_TIMEOUT Parameter

The OBJECT_DELIVERY_TIMEOUT parameter (Parameter Type 0x02) is a varint. It MAY appear in a PUBLISH_OK, SUBSCRIBE, or REQUEST_UPDATE message. Its semantics are defined in Section 8.

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.

10.2.5. FILL TIMEOUT Parameter

The FILL_TIMEOUT parameter (Parameter Type 0x0A) MAY appear in a FETCH message.

It is the maximum total duration in milliseconds a relay SHOULD spend waiting for upstream sources to provide Objects that are not immediately available before reporting them as Unknown gaps in the FETCH response. When a relay encounters Objects within the requested range that are not immediately available and have unknown status, it issues upstream FETCHes to retrieve them. The Fill Timeout represents a total budget for all such upstream FETCHes generated by this request. If the budget is exhausted, the relay reports any remaining unavailable Objects as Unknown gaps and continues delivering available Objects in the range.

A value of 0 indicates the subscriber only wants Objects that are immediately available; the relay MUST NOT wait for upstream delivery and MUST report any unavailable Objects as Unknown gaps.

If the Fill Timeout parameter is absent, the relay waits for an implementation specific duration before reporting Unknown gaps. If the subscriber specifies a Fill Timeout larger than the relay is willing to wait, the relay MAY use a shorter timeout without informing the subscriber.

10.2.6. RENDEZVOUS TIMEOUT Parameter

The RENDEZVOUS_TIMEOUT parameter (Parameter Type 0x04) MAY appear in a SUBSCRIBE message.

It is the duration in milliseconds the subscriber is willing to wait for a publisher to become available. This applies when a relay receives a SUBSCRIBE for a Track that has no current publisher.

If the RENDEZVOUS_TIMEOUT is present, the relay SHOULD hold the subscription and wait for a publisher to appear, up to the specified duration. The relay does not send SUBSCRIBE_OK until a publisher becomes available. If a publisher becomes available within this time, the relay proceeds with the subscription normally. If the timeout expires without a publisher, the relay SHOULD respond with REQUEST_ERROR with error code TIMEOUT.

The relay MAY use a shorter timeout than requested by the subscriber. For example, a relay might limit the maximum rendezvous timeout to protect its resources.

A value of 0 indicates the subscriber does not want to wait and expects an immediate response. The relay MUST immediately return REQUEST_ERROR with error code DOES_NOT_EXIST if no publisher is available

If RENDEZVOUS_TIMEOUT is absent, the default is 0.

10.2.7. SUBSCRIBER PRIORITY Parameter

The SUBSCRIBER_PRIORITY parameter (Parameter Type 0x20) is a uint8. It MAY appear in a SUBSCRIBE, FETCH, REQUEST_UPDATE (for a subscription or FETCH), or PUBLISH_OK message. It is an integer expressing the priority of a subscription relative to other subscriptions and fetch responses in the same session. Lower numbers get higher priority. See Section 7.

If omitted from SUBSCRIBE, PUBLISH_OK or FETCH, the publisher uses the value 128.

10.2.8. GROUP ORDER Parameter

The GROUP_ORDER parameter (Parameter Type 0x22) is a uint8. It MAY appear in a SUBSCRIBE, PUBLISH_OK, or FETCH.

Its value indicates how to prioritize Objects from different groups within the same subscription (see Section 7), or how to order Groups in a Fetch response (see Section 10.12.3). The allowed values are

Ascending (0x1) or Descending (0x2). If an endpoint receives a value outside this range, it MUST close the session with PROTOCOL_VIOLATION.

If omitted from SUBSCRIBE, the publisher's preference from the Track is used. If omitted from FETCH, the receiver uses Ascending (0x1).

10.2.9. SUBSCRIPTION FILTER Parameter

The SUBSCRIPTION_FILTER parameter (Parameter Type 0x21) uses length-prefixed encoding. It MAY appear in a SUBSCRIBE, PUBLISH_OK or REQUEST_UPDATE (for a subscription) message. It is a Subscription Filter (see Section 5.1.2).

If omitted from SUBSCRIBE or PUBLISH_OK, the subscription is unfiltered. If omitted from REQUEST_UPDATE, the value is unchanged.

10.2.10. EXPIRES Parameter

The EXPIRES parameter (Parameter Type 0x8) is a varint. It MAY appear in SUBSCRIBE_OK, PUBLISH, PUBLISH_OK, or REQUEST_UPDATE_OK. It encodes the time in milliseconds after which the sender of the parameter will terminate the subscription. The sender will terminate the subscription using PUBLISH_DONE or by cancelling the request (see Section 3.3.2). This value is advisory and the sender can terminate the subscription prior to or after the expiry time.

The receiver of the parameter can attempt to extend the subscription by sending a REQUEST_UPDATE with 0 or more updated parameters. If the receiver has one or more updated AUTHORIZATION_TOKENs, it SHOULD include those in the REQUEST_UPDATE. If the extension is granted, the sender includes a new EXPIRES value in REQUEST_UPDATE_OK. Relays that send this parameter and applications that receive it MAY introduce jitter to prevent many endpoints from updating simultaneously.

If the EXPIRES parameter is 0 or is not present in a message, the subscription does not expire or expires at an unknown time.

10.2.11. LARGEST OBJECT Parameter

The LARGEST_OBJECT parameter (Parameter Type 0x9) is a Location. It MAY appear in SUBSCRIBE_OK, PUBLISH, REQUEST_UPDATE_OK, or TRACK_STATUS_OK. It contains the largest Location (see Section 1.4.2) in the Track observed by the sending endpoint (see Section 5.1.2). If Objects have been published on this Track the Publisher MUST include this parameter.

If omitted from a message, the sending endpoint has not published or received any Objects in the Track.

A relay MUST set LARGEST_OBJECT to the largest of the following:

1. Any LARGEST_OBJECT value received from the upstream publisher in SUBSCRIBE_OK, PUBLISH, or REQUEST_UPDATE_OK

2. The largest Location of an Object received on an upstream subscription

10.2.12. FORWARD Parameter

The FORWARD parameter (Parameter Type 0x10) is a uint8. It MAY appear in SUBSCRIBE, REQUEST_UPDATE (for a subscription), PUBLISH, PUBLISH_OK and SUBSCRIBE_TRACKS. It specifies the Forwarding State on affected subscriptions (see Section 5.1). The allowed values are 0 (don't forward) or 1 (forward). If an endpoint receives a value outside this range, it MUST close the session with PROTOCOL_VIOLATION.

If the parameter is omitted from REQUEST_UPDATE, the value for the subscription remains unchanged. If the parameter is omitted from any other message, the default value is 1.

10.2.13. NEW GROUP REQUEST Parameter

The NEW_GROUP_REQUEST parameter (Parameter Type 0x32) is a varint. It MAY appear in PUBLISH_OK, SUBSCRIBE or REQUEST_UPDATE for a subscription. It represents the largest Group ID in the Track known by the subscriber, plus 1. A value of 0 indicates that the subscriber has no Group information for the Track. A subscriber MUST NOT send this parameter in PUBLISH_OK or REQUEST_UPDATE if the Track did not include the DYNAMIC_GROUPS Property with value 1. A subscriber MAY include this parameter in SUBSCRIBE without foreknowledge of support. If the original publisher does not support dynamic Groups, it ignores the parameter in that case.

When an Original Publisher that supports dynamic Groups receives a NEW_GROUP_REQUEST with a value of 0 or a value larger than the current Group, it SHOULD end the current Group and begin a new Group as soon as practical. The Original Publisher MAY delay the NEW_GROUP_REQUEST subject to implementation specific concerns, for example, achieving a minimum duration for each Group. The Original Publisher chooses the next Group ID; there are no requirements that it be equal to the NEW_GROUP_REQUEST parameter value.

Relay Handling:

A relay that receives a NEW_GROUP_REQUEST for a Track without an Established subscription MUST include the NEW_GROUP_REQUEST when subscribing upstream.

A relay that receives a NEW_GROUP_REQUEST for an Established subscription with a value of 0 or a value larger than the Largest Group MUST send a REQUEST_UPDATE including the NEW_GROUP_REQUEST to the publisher unless:

1. The Track does not support dynamic Groups

2. There is already an outstanding NEW_GROUP_REQUEST from this Relay with a greater or equal value

If a relay receives a NEW_GROUP_REQUEST with a non-zero value less than or equal to the Largest Group, it does not send a NEW_GROUP_REQUEST upstream.

After sending a NEW_GROUP_REQUEST upstream, the request is considered outstanding until the Largest Group increases.

10.2.14. TRACK_NAMESPACE_PREFIX Parameter

The TRACK_NAMESPACE_PREFIX parameter (Parameter Type 0x34) uses the Track Namespace encoding described in Section 2.4.1. It MAY appear in REQUEST_UPDATE for a SUBSCRIBE_NAMESPACE or SUBSCRIBE_TRACKS request. It updates the Track Namespace Prefix for that subscription. If the new prefix would share a common prefix with another active subscription of the same type in the same session, the receiver MUST respond with REQUEST_ERROR with error code PREFIX_OVERLAP.

10.3. SETUP

The SETUP message is the first message each endpoint sends on its control stream (see Section 3.3); it allows the endpoints to agree on the initial configuration before any other control messages are exchanged. An endpoint that is not offering extensions which modify control message semantics MAY pipeline other control messages after SETUP without waiting for the peer’s SETUP.

The messages contain a sequence of key-value pairs called Setup Options; 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 Options.

The wire format of the Setup message is as follows:

```
SETUP Message {
  Type (vi64) = 0x2F00,
  Length (16),
  Setup Options (..) ...,
}
```

Figure 6: MOQT SETUP Message

Setup Options are serialized as Key-Value-Pairs Figure 2, spanning the entire message payload, bounded by the message Length field. Setup Options use a namespace that is constant across all MOQT versions, separate from Message Parameters. Receivers MUST ignore unrecognized Setup Options. Senders MUST NOT repeat the same Option Type in a message unless the option definition explicitly allows multiple instances. Receivers MUST allow duplicates of unknown Setup Options.

The available Setup Options are detailed in the next sections.

10.3.1. Setup Options

10.3.1.1. AUTHORITY

The AUTHORITY option (Option Type 0x05) allows the client to specify the authority component of the MoQ URI when using native QUIC (Section 3.1.4). It MUST NOT be used by the server, or when WebTransport is used. When an AUTHORITY option is received from a server, or when an AUTHORITY option is received while WebTransport is used, or when an AUTHORITY option is received by a server but the server does not support the specified authority, the session MUST be closed with INVALID_AUTHORITY.

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

10.3.1.2. PATH

The PATH option (Option Type 0x01) allows the client to specify the path of the MoQ URI when using native QUIC (Section 3.1.4). 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 option follows the URI formatting rules [RFC3986]. When connecting to a server using a URI with the "moqt" scheme, the client MUST set the PATH option 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 option. If a PATH does not conform to these rules, the session MUST be closed with MALFORMED_PATH.

10.3.1.3. MAX_AUTH_TOKEN_CACHE_SIZE

The MAX_AUTH_TOKEN_CACHE_SIZE option (Option Type 0x04) communicates the maximum size in bytes of all actively registered Authorization tokens that the endpoint is willing to store per Session. This option 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 5). The total size as restricted by the MAX_AUTH_TOKEN_CACHE_SIZE option 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.

10.3.1.4. AUTHORIZATION TOKEN

The AUTHORIZATION TOKEN Setup Option (Option Type 0x03) is functionally equivalent to the AUTHORIZATION TOKEN message parameter, see Section 10.2.2. The endpoint can specify one or more tokens in SETUP that the peer can use to authorize MOQT session establishment.

If an endpoint receives an AUTHORIZATION TOKEN option in SETUP with Alias Type REGISTER that exceeds its MAX_AUTH_TOKEN_CACHE_SIZE, it MUST NOT fail the session with AUTH_TOKEN_CACHE_OVERFLOW. Instead, it MUST treat the option as Alias Type USE_VALUE. Since each endpoint’s SETUP may be sent before the peer’s SETUP is received, the sender MUST handle registration failures of this kind by purging any Token Aliases that failed to register based on the peer’s MAX_AUTH_TOKEN_CACHE_SIZE option in SETUP (or the default value of 0).

10.3.1.5. MOQT IMPLEMENTATION

The MOQT_IMPLEMENTATION option (Option Type 0x07) 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.

An endpoint SHOULD send a MOQT_IMPLEMENTATION option unless specifically configured not to do so. This option helps identify the scope of interoperability problems and work around implementation-specific limitations.

Senders SHOULD limit the value to the implementation name and version, avoiding advertising or other nonessential information. Implementations SHOULD NOT use the identifiers of other implementations to declare compatibility, as this undermines the usefulness of implementation identification for debugging.

10.4. GOAWAY

An endpoint sends a GOAWAY message on its control stream to inform the peer it intends to close the session soon. When sent by a server, it can initiate session migration (Section 3.6) with an optional URI. A client MUST send a zero-length New Session URI in any GOAWAY, as clients cannot instruct servers to initiate connections.

A GOAWAY MAY also be sent on a request stream to initiate migration of that individual request. Upon receiving a GOAWAY on a request stream, the endpoint SHOULD re-issue that specific request on a session at the specified URI (or the current session if no URI is provided), and close the old request stream using the appropriate mechanism (e.g. FIN, stream reset, or PUBLISH_DONE).

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 on the control stream, an endpoint SHOULD NOT initiate new requests to the peer including SUBSCRIBE, PUBLISH, FETCH, PUBLISH_NAMESPACE, SUBSCRIBE_NAMESPACE, SUBSCRIBE_TRACKS and TRACK_STATUS.

Sending a GOAWAY does not prevent the sender from initiating new requests, though the sender SHOULD avoid initiating requests unless required by migration (see (Section 9.4.1 and Section 9.5.1). An endpoint that receives a GOAWAY MAY reject new requests with an appropriate error code (e.g., REQUEST_ERROR with error code GOING_AWAY).

The endpoint MUST close the session with a PROTOCOL_VIOLATION (Section 3.5) if it receives more than one GOAWAY on the control stream or on a single request stream.

```
GOAWAY Message {
  Type (vi64) = 0x10,
  Length (16),
  New Session URI Length (vi64),
  New Session URI (..),
  Timeout (vi64),
  [Request ID (vi64)],
}
```

Figure 7: MOQT GOAWAY Message

* New Session URI: When received by a client, indicates where the client can connect to continue this session or re-issue this request. 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 close the session with a PROTOCOL_VIOLATION.

* Timeout: The time in milliseconds the sender will wait for graceful closure. When sent on the control stream, the sender closes the session with GOAWAY_TIMEOUT after the indicated timeout if there are still open requests. When sent on a request stream, the sender SHOULD reset the stream with GOING_AWAY after the indicated timeout. A value of 0 indicates the sender has no specific timeout, but the recipient SHOULD migrate as quickly as possible. This is a hint; the sender of the GOAWAY MAY close the session or reset the request stream before the indicated timeout has elapsed.

* Request ID: Present only when sent on the control stream. The smallest peer Request ID that was not or might not have been processed prior to sending the GOAWAY. If no requests have been processed, this is 0 (at a server) or 1 (at a client). If the parity of the Request ID does not match the receiver’s parity, the endpoint MUST close the session with INVALID_REQUEST_ID. Requests with a Request ID equal to or greater than the indicated value, as well as any requests that arrive after the GOAWAY, MUST be rejected with REQUEST_ERROR using error code GOING_AWAY. Requests with a Request ID less than the indicated value were or might have been processed; their status can be determined from the response on each request stream.

10.5. REQUEST_OK

The REQUEST_OK message is sent in response to PUBLISH, REQUEST_UPDATE, TRACK_STATUS, SUBSCRIBE_NAMESPACE, SUBSCRIBE_TRACKS and PUBLISH_NAMESPACE requests.

This document uses the shorthand PUBLISH_OK, REQUEST_UPDATE_OK, TRACK_STATUS_OK, SUBSCRIBE_NAMESPACE_OK, and PUBLISH_NAMESPACE_OK to refer to a REQUEST_OK sent in response to the corresponding request type.

```
REQUEST_OK Message {
  Type (vi64) = 0x7,
  Length (16),
  Number of Parameters (vi64),
  Parameters (..) ...,
  Track Properties (..),
}
```

Figure 8: MOQT REQUEST_OK Message

* Parameters: The parameters are defined in Section 10.2.

* Track Properties : A sequence of Properties. See Section 2.5. The length of Track Properties is the remaining length of the message after parsing all previous fields. Track Properties are populated in TRACK_STATUS_OK; they are empty in PUBLISH_OK, REQUEST_UPDATE_OK, SUBSCRIBE_NAMESPACE_OK and PUBLISH_NAMESPACE_OK. If an endpoint receives Track Properties in one of these messages it MUST close the session with a PROTOCOL_VIOLATION.

10.6. REQUEST_ERROR

The REQUEST_ERROR message is sent in response to any request (SUBSCRIBE, FETCH, PUBLISH, SUBSCRIBE_NAMESPACE, SUBSCRIBE_TRACKS, PUBLISH_NAMESPACE, TRACK_STATUS, REQUEST_UPDATE).

10.6.1. Redirect Structure

A Redirect provides a way for an endpoint to direct the peer to retry a request at a different URI and/or for a different Full Track Name.

```
Redirect {
  Connect URI Length (vi64),
  Connect URI (..),
  Track Namespace (..),
  Track Name Length (vi64),
  Track Name (..),
}
```

* Connect URI: The URI to connect to for this track. If the length is zero, the requester SHOULD use the current session's URI. If a server receives a Redirect with a non-zero Connect URI Length it MUST close the session with a PROTOCOL_VIOLATION.

* Track Namespace and Track Name: The Track Namespace and Track Name to use for the redirected request. If both have zero length, the redirected request uses the same values as the original request. Otherwise, Track Namespace and Track Name are the literal values for the redirected request.

  Track Name is not meaningful for namespace-scoped requests (SUBSCRIBE_NAMESPACE, PUBLISH_NAMESPACE) and MUST be empty; an endpoint that receives a non-empty Track Name in a Redirect for a namespace-scoped request MUST close the session with a PROTOCOL_VIOLATION.

10.6.2. REQUEST_ERROR Message Format

```
REQUEST_ERROR Message {
  Type (vi64) = 0x5,
  Length (16),
  Error Code (vi64),
  Retry Interval (vi64),
  Error Reason (Reason Phrase),
  [Redirect (Redirect),]
}
```

Figure 9: MOQT REQUEST_ERROR Message

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

* Retry Interval: The minimum time (in milliseconds) before the request SHOULD be sent again, plus one. If the value is 0, the request SHOULD NOT be retried.

* Error Reason: Provides a text description of the request error. See Section 1.4.4.

* Redirect: Present only when Error Code is REDIRECT. See Section 10.6.1.

The application SHOULD use a relevant error code in REQUEST_ERROR, as defined below and assigned in Section 15.10.2. Most codepoints have identical meanings for various request types, but some have request-specific meanings.

If a request is retryable with the same parameters at a later time, the sender of REQUEST_ERROR includes a non-zero Retry Interval in the message. To minimize the risk of synchronized retry storms, the sender can apply randomization to each retry interval so that retries are spread out over time. A Retry Interval value of 1 indicates the request can be retried immediately.

INTERNAL_ERROR: An implementation specific or generic error occurred.

UNAUTHORIZED: The subscriber is not authorized to perform the requested action on the given track. This might be retryable if the authorization token is not yet valid.

TIMEOUT: 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: The endpoint does not support the type of request.

MALFORMED_AUTH_TOKEN: Invalid Auth Token serialization during registration (see Section 10.2.2).

EXPIRED_AUTH_TOKEN: Authorization token has expired (Section 10.2.2).

GOING_AWAY: The endpoint has received a GOAWAY and MAY reject new requests.

EXCESSIVE_LOAD: The responder is overloaded and cannot process the request at this time. The sender SHOULD use the Retry Interval to indicate when the request can be retried.

UNSUPPORTED_EXTENSION: The track contains a Mandatory Track Property (see Section 2.5.1) that the endpoint does not understand.

DUPLICATE_SUBSCRIPTION (0x19): The PUBLISH or SUBSCRIBE request attempted to create a subscription to a Track with the same role as an existing subscription.

REDIRECT: The request cannot be fulfilled by this endpoint, but could succeed at the location specified in the Redirect structure. The requester SHOULD establish a new session to the provided URI (if present) and retry the request using the Full Track Name from the Redirect (if present). This error code can appear in response to SUBSCRIBE, FETCH, TRACK_STATUS, PUBLISH_NAMESPACE and SUBSCRIBE_NAMESPACE. Relays are not required to follow redirects from upstream and MAY forward a REDIRECT response to matching downstream requests. A relay MAY cache a REDIRECT response for a Full Track Name for up to Retry Interval milliseconds and use it to respond to subsequent matching requests without forwarding them upstream.

Below are errors for use by the publisher. They can appear in response to SUBSCRIBE, FETCH, TRACK_STATUS, SUBSCRIBE_NAMESPACE, and SUBSCRIBE_TRACKS, unless otherwise noted.

DOES_NOT_EXIST: The track or namespace is not available at the publisher.

INVALID_RANGE: In response to SUBSCRIBE or FETCH, specified Filter or range of Locations cannot be satisfied.

MALFORMED_TRACK: In response to a FETCH, a relay publisher detected the track was malformed (see Section 2.4.2).

The following are errors for use by the subscriber. They can appear in response to PUBLISH or PUBLISH_NAMESPACE, unless otherwise noted.

UNINTERESTED: The subscriber is not interested in the track or namespace.

Errors below can only be used in response to one message type.

PREFIX_OVERLAP: In response to SUBSCRIBE_NAMESPACE or SUBSCRIBE_TRACKS, the namespace prefix shares a common prefix with another subscription of the same type in the same session. SUBSCRIBE_NAMESPACE and SUBSCRIBE_TRACKS have independent overlap spaces, so a SUBSCRIBE_NAMESPACE and a SUBSCRIBE_TRACKS may share the same prefix.

NAMESPACE_TOO_LARGE: In response to SUBSCRIBE_NAMESPACE or SUBSCRIBE_TRACKS, the namespace prefix matches more publishers than the relay is willing to enumerate.

INVALID_JOINING_REQUEST_ID: In response to a Joining FETCH, the referenced Request ID is not an Established Subscription.

10.7. SUBSCRIBE

A subscription causes the publisher to send newly published objects for a track.

Subscribe only requests newly published or received Objects. Objects from the past are retrieved using FETCH (Section 10.12).

The format of SUBSCRIBE is as follows:

```
SUBSCRIBE Message {
  Type (vi64) = 0x3,
  Length (16),
  Request ID (vi64),
  Track Namespace (..),
  Track Name Length (vi64),
  Track Name (..),
  Number of Parameters (vi64),
  Parameters (..) ...
}
```

Figure 10: MOQT SUBSCRIBE Message

* Request ID: See Section 10.1.

* 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).

* Parameters: The parameters are defined in Section 10.2.

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 start sending objects.

10.8. SUBSCRIBE_OK

A publisher sends a SUBSCRIBE_OK as the first response message on the bidi stream for successful subscriptions.

```
SUBSCRIBE_OK Message {
  Type (vi64) = 0x4,
  Length (16),
  Track Alias (vi64),
  Number of Parameters (vi64),
  Parameters (..) ...,
  Track Properties (..),
}
```

Figure 11: MOQT SUBSCRIBE_OK Message

* Track Alias: The identifer used for this track in Subgroups or Datagrams (see Section 11.1).

* Parameters: The parameters are defined in Section 10.2.

* Track Properties : A sequence of Properties. See Section 2.5.

10.9. REQUEST_UPDATE

The sender of a request (SUBSCRIBE, PUBLISH, FETCH, PUBLISH_NAMESPACE, SUBSCRIBE_NAMESPACE, SUBSCRIBE_TRACKS) can later send a REQUEST_UPDATE on the same bidi stream as the request to modify it. A subscriber can also send REQUEST_UPDATE to modify parameters of a subscription established with PUBLISH.

The receiver of a REQUEST_UPDATE MUST respond with exactly one REQUEST_OK or REQUEST_ERROR message indicating if the update was successful.

If a parameter previously set on the request is not present in REQUEST_UPDATE, its value remains unchanged.

There is no mechanism to remove a parameter from a request.

The format of REQUEST_UPDATE is as follows:

```
REQUEST_UPDATE Message {
  Type (vi64) = 0x2,
  Length (16),
  Request ID (vi64),
  Number of Parameters (vi64),
  Parameters (..) ...
}
```

Figure 12: MOQT REQUEST_UPDATE Message

* Request ID: See Section 10.1.

* Parameters: The parameters are defined in Section 10.2.

10.9.1. Updating Subscriptions

When a subscriber decreases the Start Location of the Subscription Filter (see Section 5.1.2), the Start Location can be smaller than the Track's Largest Location, similar to a new Subscription. FETCH can be used to retrieve any necessary Objects smaller than the current Largest Location.

When a subscriber increases the End Location, the Largest Object at the publisher might already be larger than the previous End Location. This will create a gap in the subscription. The REQUEST_UPDATE_OK will include the LARGEST_OBJECT parameter, and the subscriber can issue a FETCH to retrieve the omitted Objects, if any.

When a subscriber narrows their subscription (increase the Start Location and/or decrease the End Group), it might still receive Objects outside the new range if the publisher sent them before the update was processed.

When a REQUEST_UPDATE is unsuccessful, the publisher MUST also terminate the subscription by sending a PUBLISH_DONE with error code UPDATE_FAILED. When a REQUEST_UPDATE fails for a FETCH, the publisher MUST reset the FETCH data stream. When a REQUEST_UPDATE fails for a SUBSCRIBE_NAMESPACE or PUBLISH_NAMESPACE, the responder MUST close the bidi stream.

A receiver of multiple REQUEST_UPDATE messages on the same stream MAY coalesce their processing by applying only the cumulative result. Parameter values from later REQUEST_UPDATE messages override values from earlier ones. The receiver MUST still send a REQUEST_OK for each successful update, but it is not required to process intermediate states individually. If the coalesced REQUEST_UPDATE results in REQUEST_ERROR, only a single REQUEST_ERROR will be sent and the sender of the REQUEST_UPDATEs will not always be able to determine which caused an error.

10.9.2. Updating Namespace Subscriptions

A subscriber can update the Track Namespace Prefix of an established SUBSCRIBE_NAMESPACE or SUBSCRIBE_TRACKS by including the TRACK_NAMESPACE_PREFIX parameter (Section 10.2.14) in a REQUEST_UPDATE. The overlap restriction applies independently per type: the new prefix MUST NOT share a common prefix with any other active SUBSCRIBE_NAMESPACE (for a SUBSCRIBE_NAMESPACE update) or SUBSCRIBE_TRACKS (for a SUBSCRIBE_TRACKS update) in the same session. If the update is accepted, NAMESPACE and NAMESPACE_DONE messages

following the REQUEST_OK will contain Track Namespace suffixes relative to the updated prefix. Updating the prefix of a SUBSCRIBE_TRACKS has no effect on existing subscriptions. If the subscriber is no longer interested it can cancel the corresponding bidirectional stream.

10.10. PUBLISH

The publisher sends PUBLISH as the first message on a new bidirectional stream to initiate a subscription for a Track. The receiver verifies the publisher is authorized to publish this track.

```
PUBLISH Message {
  Type (vi64) = 0x1D,
  Length (16),
  Request ID (vi64),
  Track Namespace (..),
  Track Name Length (vi64),
  Track Name (..),
  Track Alias (vi64),
  Number of Parameters (vi64),
  Parameters (..) ...,
  Track Properties (..),
}
```

Figure 13: MOQT PUBLISH Message

* Request ID: See Section 10.1.

* 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 identifer used for this track in Subgroups or Datagrams (see Section 11.1).

* Parameters: The parameters are defined in Section 10.2.

* Track Properties : A sequence of Properties. See Section 2.5.

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

A publisher that sends the FORWARD parameter (Section 10.2.12) equal to 0 indicates that it will not transmit any objects until the subscriber sets the Forward State to 1. If the FORWARD parameter is omitted or equal to 1, the publisher will start transmitting objects immediately, possibly before PUBLISH_OK.

10.11. PUBLISH_DONE

A publisher sends a PUBLISH_DONE message as the final message before closing the subscription's bidi stream 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 a 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 timeouts (see Section 8).

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 subgroup delivery timeout.

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 sender SHOULD send FIN on the subscription's bidi stream immediately after sending PUBLISH_DONE.

A subscriber that receives PUBLISH_DONE SHOULD set a timer of at least the larger of SUBGROUP_DELIVERY_TIMEOUT or OBJECT_DELIVERY_TIMEOUT in case some objects are still inbound due to prioritization or packet loss. The subscriber MAY dispense with a timer if it unsubscribed 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 (vi64) = 0xB,
  Length (16),
  Status Code (vi64),
  Stream Count (vi64),
  Error Reason (Reason Phrase)
}
```

Figure 14: MOQT PUBLISH_DONE Message

* 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, including streams that contained no Objects (e.g., an empty Subgroup). 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 publisher did not open any streams for this subscription, 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 1.4.4.

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 subscription filter.

GOING_AWAY (0x4): The subscriber or publisher issued a GOAWAY

message.

TOO_FAR_BEHIND (0x5): The publisher's queue of objects to be sent to the given subscriber exceeds its implementation defined limit.

EXPIRED (0x6): The publisher reached the timeout specified in SUBSCRIBE_OK.

MALFORMED_TRACK (0x12): A relay publisher detected that the track was malformed (see Section 2.4.2).

UPDATE_FAILED (0x8): REQUEST_UPDATE failed on this subscription (see Section 10.9).

EXCESSIVE_LOAD (0x9): The publisher is overloaded and is terminating the subscription.

10.12. FETCH

A subscriber sends FETCH as the first message on a new bidi stream 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 6

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

10.12.1. Standalone Fetch

A Fetch of Objects performed independently of any Subscribe.

A Standalone Fetch includes this structure:

```
Standalone Fetch {
  Track Namespace (..),
  Track Name Length (vi64),
  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.

10.12.2. Joining Fetches

A Joining Fetch is associated with a Subscribe request by specifying the Request ID of a subscription in the Established or Pending (subscriber) state. Because Joining Fetch references an existing subscription, if that subscription has not yet been established, the Publisher receiving the Joining Fetch buffers the pending Joining Fetch until either the Subscription is established or the request times out.

A publisher receiving a Joining Fetch uses properties of the associated subscription to determine the Track Namespace, Track Name and End Location such that it is contiguous with the associated subscription. The subscriber can set the Start Location to an absolute Location or a Location relative to the Largest 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 subscription has Forward State 1; otherwise the publisher MUST respond with a REQUEST_ERROR with error code INVALID_RANGE. A publisher MUST process any pending REQUEST_UPDATE messages for the associated subscription before evaluating the current request. Relays with an upstream subscription in transition from Forward State 0 to 1 can either send a Joining Fetch upstream or buffer the Joining Fetch until the upstream subscription returns REQUEST_UPDATE_OK with the new Largest Object. Changing the Forward State of the associated subscription to 0 after the Joining Fetch has been accepted has no effect on the Joining Fetch.

If no Objects have been published for the track the publisher MUST respond with a REQUEST_ERROR with error code INVALID_RANGE.

A Joining Fetch includes this structure:

```
Joining Fetch {
  Joining Request ID (vi64),
  Joining Start (vi64)
}
```

* Joining Request ID: The Request ID of the subscription to be joined. If a publisher receives a Joining Fetch with a Request ID that does not correspond to a subscription in the same session in the Established or Pending (subscriber) states, it MUST return a REQUEST_ERROR with error code INVALID_JOINING_REQUEST_ID.

* Joining Start : A relative or absolute value used to determine the Start Location, described below.

10.12.2.1. Joining Fetch Range Calculation

The Joining 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 {Joining Location.Group, Joining Location.Object + 1} (see Section 5.1.

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

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

For an Absolute Joining Fetch, the publisher sets the Start Location to {Joining Start, 0}.

10.12.3. Fetch Handling

The format of FETCH is as follows:

```
FETCH Message {
  Type (vi64) = 0x16,
  Length (16),
  Request ID (vi64),
  Fetch Type (vi64),
  [Standalone (Standalone Fetch),]
  [Joining (Joining Fetch),]
  Number of Parameters (vi64),
  Parameters (..) ...
}
```

Figure 15: MOQT FETCH Message

* Request ID: See Section 10.1.

* 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 10.2.

A publisher responds to a FETCH request with either a FETCH_OK or a REQUEST_ERROR message. The publisher creates a new unidirectional stream that is used to send the Objects. The FETCH_OK or REQUEST_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 (see Section 10.2.8). 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. 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 opens the unidirectional stream, sends the FETCH_HEADER (see Section 11.4.4) and closes the stream with a FIN.

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 REQUEST_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 larger than the Largest Object will not be retrieved by a FETCH. If the requested End Location exceeds the Largest available Object, the actual end of the FETCH response is indicated in the FETCH_OK End Location.

If no Objects have been published for the track or Start Location is greater than the Largest Object (Section 10.7) the publisher MUST return REQUEST_ERROR with error code INVALID_RANGE.

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

If a Publisher receives a FETCH with a range that includes one or more Objects with unknown status (e.g. a Relay has temporarily lost contact with the Original Publisher and does not have the Object in cache), it can choose to reset the FETCH data stream with UNKNOWN_OBJECT_STATUS (Section 3.3.3), or indicate the range of unknown Objects and continue serving other known Objects.

10.13. FETCH_OK

A publisher sends a FETCH_OK as the first message on the bidi stream in response to a successful fetch. 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 (vi64) = 0x18,
  Length (16),
  End Of Track (8),
  End Location (Location),
  Number of Parameters (vi64),
  Parameters (..) ...
  Track Properties (..),
}
```

Figure 16: MOQT FETCH_OK Message

* 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 end of the range covered by the FETCH response, using the same encoding as the FETCH request End Location (the last Object, plus 1; or 0 to indicate the entire Group). This is the End Location from the FETCH request unless the requested range extends beyond published data:

  - If the requested FETCH End Location was beyond the Largest known (possibly final) Object, End Location is {Largest.Group, Largest.Object + 1} Where Fetch.End Location is either Fetch.Standalone.End Location or the computed End Location described in Section 10.12.2.1.

  If End Location is smaller than the Start Location in the corresponding FETCH the receiver MUST close the session with a PROTOCOL_VIOLATION.

* Parameters: The parameters are defined in Section 10.2.

* Track Properties : A sequence of Properties. See Section 2.5.

10.14. TRACK_STATUS

A potential subscriber sends TRACK_STATUS as the first and only message on a new bidi stream to obtain information about the current status of a given track.

The TRACK_STATUS message format is identical to the SUBSCRIBE message (Section 10.7), but subscriber parameters related to Track delivery (e.g. SUBSCRIBER_PRIORITY) are not included.

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. If successful, the publisher responds with a TRACK_STATUS_OK with the same parameters and Track Properties it would have set in a SUBSCRIBE_OK. Track Alias is not used. A publisher responds to a failed TRACK_STATUS with an appropriate REQUEST_ERROR message. The bidi stream is closed with a FIN after TRACK_STATUS_OK or REQUEST_ERROR are sent.

Relays without an Established subscription MAY forward TRACK_STATUS to one or more publishers, or MAY initiate a subscription (subject to authorization) as described in Section 9.5 to determine the response. The publisher does not send PUBLISH_DONE for this request, and the subscriber cannot send REQUEST_UPDATE.

10.15. PUBLISH_NAMESPACE

The publisher sends the PUBLISH_NAMESPACE message as the first message on a new bidi stream 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 (vi64) = 0x6,
  Length (16),
  Request ID (vi64),
  Track Namespace (..),
  Number of Parameters (vi64),
  Parameters (..) ...
}
```

Figure 17: MOQT PUBLISH_NAMESPACE Message

* Request ID: See Section 10.1.

* Track Namespace: Identifies a track’s namespace as defined in Section 2.4.1.

* Parameters: The parameters are defined in Section 10.2.

10.16. NAMESPACE

The NAMESPACE message is similar to the PUBLISH_NAMESPACE message, except it is sent on the response stream of a SUBSCRIBE_NAMESPACE request. All NAMESPACE messages are in response to a SUBSCRIBE_NAMESPACE, so only the namespace tuples after the 'Track Namespace Prefix' are included in the 'Track Namespace Suffix'.

```
NAMESPACE Message {
  Type (i) = 0x8,
  Length (16),
  Track Namespace Suffix (..),
}
```

Figure 18: MOQT NAMESPACE Message

* Track Namespace Suffix: Specifies the final portion of a track's namespace as defined in Section 2.4.1 after removing namespace tuples included in 'Track Namespace Prefix' Section 10.18.

10.17. NAMESPACE_DONE

The publisher sends the NAMESPACE_DONE control message to indicate its intent to stop serving new subscriptions for tracks within the provided Track Namespace. All NAMESPACE_DONE messages are in response to a SUBSCRIBE_NAMESPACE, so only the namespace tuples after the 'Track Namespace Prefix' are included in the 'Track Namespace Suffix'.

```
NAMESPACE_DONE Message {
  Type (i) = 0xE,
  Length (16),
  Track Namespace Suffix (..)
}
```

Figure 19: MOQT NAMESPACE_DONE Message

* Track Namespace Suffix: Specifies the final portion of a track's namespace as defined in Section 2.4.1. The namespace begins with the 'Track Namespace Prefix' specified in Section 10.18.

10.18. SUBSCRIBE_NAMESPACE

The subscriber sends a SUBSCRIBE_NAMESPACE control message on a new bidirectional stream to a publisher to request the current set of matching published namespaces, as well as future updates to the set.

```
SUBSCRIBE_NAMESPACE Message {
  Type (vi64) = 0x50,
  Length (16),
  Request ID (vi64),
  Track Namespace Prefix (..),
  Number of Parameters (vi64),
  Parameters (..) ...
}
```

Figure 20: MOQT SUBSCRIBE_NAMESPACE Message

* Request ID: See Section 10.1.

* Track Namespace Prefix: A Track Namespace structure as described in Section 2.4.1 with between 0 and 32 Track Namespace Fields. This prefix is 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 consisting of greater than 32 Track Namespace Fields, it MUST close the session with a PROTOCOL_VIOLATION.

* Parameters: The parameters are defined in Section 10.2.

The publisher will respond with REQUEST_OK or REQUEST_ERROR on the response half of the stream. If the subscriber receives any message other than a REQUEST_OK or a REQUEST_ERROR as the first message on the response half of the stream, then it MUST close the session with a PROTOCOL_VIOLATION. If the SUBSCRIBE_NAMESPACE is successful, the publisher will send matching NAMESPACE messages on the response stream. If it is an error, the stream will be immediately closed via FIN. When there are changes to the namespaces being published and the subscriber is subscribed to them, the publisher sends the corresponding NAMESPACE or NAMESPACE_DONE messages.

Within a session, if a publisher receives a SUBSCRIBE_NAMESPACE with a Track Namespace Prefix that shares a common prefix with an established SUBSCRIBE_NAMESPACE, it MUST respond with REQUEST_ERROR with error code PREFIX_OVERLAP. SUBSCRIBE_NAMESPACE and SUBSCRIBE_TRACKS have independent overlap spaces (see Section 10.19).

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

The publisher MUST NOT send NAMESPACE_DONE for a namespace suffix before the corresponding NAMESPACE. If a subscriber receives a NAMESPACE_DONE before the corresponding NAMESPACE, it MUST close the session with a 'PROTOCOL_VIOLATION'.

If the publisher is unable to send NAMESPACE or NAMESPACE_DONE messages in a timely manner because the SUBSCRIBE_NAMESPACE response stream is blocked by flow control, the publisher MAY reset the SUBSCRIBE_NAMESPACE response stream. When a subscriber receives a stream reset or FIN on a SUBSCRIBE_NAMESPACE response stream, it SHOULD treat this as though each active namespace received a NAMESPACE_DONE. Subscriptions established via PUBLISH on separate bidi streams are not affected by closure of the SUBSCRIBE_NAMESPACE stream.

10.19. SUBSCRIBE_TRACKS

The subscriber sends a SUBSCRIBE_TRACKS control message on a new bidirectional stream to a publisher to request PUBLISH messages for all tracks within matching namespaces, as well as future track publications within those namespaces.

```
SUBSCRIBE_TRACKS Message {
  Type (vi64) = 0x51,
  Length (16),
  Request ID (vi64),
  Track Namespace Prefix (..),
  Number of Parameters (vi64),
  Parameters (..) ...
}
```

Figure 21: MOQT SUBSCRIBE_TRACKS Message

* Request ID: See Section 10.1.

* Track Namespace Prefix: A Track Namespace structure as described in Section 2.4.1 with between 0 and 32 Track Namespace Fields. This prefix is matched against track namespaces known to the publisher. If an endpoint receives a Track Namespace Prefix consisting of greater than 32 Track Namespace Fields, it MUST close the session with a PROTOCOL_VIOLATION.

* Parameters: The parameters are defined in Section 10.2.

The publisher will respond with REQUEST_OK or REQUEST_ERROR on the response half of the stream. If the subscriber receives any message other than a REQUEST_OK or a REQUEST_ERROR as the first message on the response half of the stream, then it MUST close the session with

a PROTOCOL_VIOLATION. If the SUBSCRIBE_TRACKS is successful, the publisher will send PUBLISH messages on new bidirectional streams for tracks within matching namespaces. If it is an error, the stream will be closed via FIN after REQUEST_ERROR is sent.

Within a session, if a publisher receives a SUBSCRIBE_TRACKS with a Track Namespace Prefix that shares a common prefix with an established SUBSCRIBE_TRACKS, it MUST respond with REQUEST_ERROR with error code PREFIX_OVERLAP. SUBSCRIBE_TRACKS and SUBSCRIBE_NAMESPACE have independent overlap spaces (see Section 10.18).

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

SUBSCRIBE_TRACKS is not required for a publisher to send PUBLISH messages to a subscriber. It is useful for subscribers that are only interested in or authorized to access a subset of available tracks.

If the FORWARD parameter (Section 10.2.12) is present in this message and equal to 0, PUBLISH messages resulting from this SUBSCRIBE_TRACKS will set the FORWARD parameter to 0. If the FORWARD parameter is equal to 1 or omitted from this message, PUBLISH messages resulting from this SUBSCRIBE_TRACKS will set the FORWARD parameter to 1, or indicate that value by omitting the parameter (see Section 5.1).

10.20. PUBLISH_BLOCKED

The publisher sends the PUBLISH_BLOCKED control message to indicate it cannot send a PUBLISH message to initiate a new Subscription for a Track in the SUBSCRIBE_TRACKS's Track Namespace. All PUBLISH_BLOCKED messages are in response to a SUBSCRIBE_TRACKS, so only the namespace tuples after the 'Track Namespace Prefix' are included in the 'Track Namespace Suffix'.

```
PUBLISH_BLOCKED Message {
  Type (vi64) = 0xF,
  Length (16),
  Track Namespace Suffix (..),
  Track Name Length (vi64),
  Track Name (..),
}
```

Figure 22: MOQT PUBLISH_BLOCKED Message

* Track Namespace Suffix: Specifies the final portion of a track's namespace as defined in Section 2.4.1. The namespace begins with the 'Track Namespace Prefix' specified in Section 10.19.

* Track Name: Identifies the track name as defined in (Section 2.4.1).

11. Data Streams and Datagrams

A publisher sends Objects matching a subscription on Data Streams or Datagrams and sends Objects matching a FETCH request on one Data Stream.

Unidirectional stream types are defined in Section 3.4. Data streams use SUBGROUP_HEADER or FETCH_HEADER types.

All MOQT datagrams start with a variable-length integer indicating the type of the datagram. See Section 11.3.1.

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

Every Object has a 'Object Forwarding Preference' and the Original Publisher MAY use both Subgroups and Datagrams within a Group or Track.

11.1. 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 10.8) or PUBLISH (Section 10.10).

The same Track Alias MUST NOT be used by a publisher to refer to two different Tracks simultaneously in the same session. If a subscriber receives a PUBLISH or SUBSCRIBE_OK that uses the same Track Alias as a different Track with an Established subscription, it MUST close the session with error DUPLICATE_TRACK_ALIAS.

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.

11.2. 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.

11.2.1. Object Header

A canonical MOQT Object has the following fields:

* Track Namespace and Track Name: The track this object belongs to.

* Group ID: The identifier of the Object's Group (see 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. Object Forwarding Preference is a property of an individual Object and can vary among Objects in the same Track. In a subscription, an Object MUST be sent according to its Object Forwarding Preference.

* Subgroup ID: The identifier of the Object's Subgroup (see Section 2.2) within the Group. This field is omitted if the Object Forwarding Preference is Datagram.

* Object Status: An enumeration used to indicate whether the Object is a normal Object or mark the end of a group or track. See Section 11.2.1.1 below.

* Object Properties : A sequence of Properties associated with the object. See Section 11.2.1.2.

* 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).

11.2.1.1. Object Status

The Object Status is a field that is only present in objects that are delivered via a SUBSCRIPTION, and is absent in Objects delivered via a FETCH. It allows the publisher to explicitly communicate that a specific range of objects does not exist.

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.

* 0x3 := Indicates End of Group. Indicates that no objects with the specified Group ID and the Object ID that is greater than or equal to the one specified exist in the group identified by the Group ID.

* 0x4 := Indicates End of Track. Indicates that no objects with the location that is equal to or greater than the one specified exist.

All of those SHOULD be cached.

Any other value SHOULD be treated as a protocol error and the session SHOULD be closed with a PROTOCOL_VIOLATION (Section 3.5). Any object with a status code other than zero MUST have an empty payload.

11.2.1.2. Object Properties

Any Object with status Normal can have properties (Section 2.5). If an endpoint receives properties on an Object with status that is not Normal, it MUST close the session with a PROTOCOL_VIOLATION.

Object Properties are visible to relays and are intended to be relevant to MOQT Object distribution. Any Object metadata never intended to be accessed by the transport or Relays SHOULD be serialized as part of the Object payload and not as an Object Property.

Object Properties are serialized as a length in bytes followed by Key-Value-Pairs (see Figure 2).

```
Object Properties {
  Properties Length (vi64),
  Properties (..),
}
```

Object Property types are registered in the IANA table 'MOQ Properties'. See Section 15.

11.3. Datagrams

A single object can be conveyed in a datagram. The Track Alias field (Section 11.1) 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 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 the 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 Properties (Section 11.2.1.2) 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.

11.3.1. Object Datagram

An OBJECT_DATAGRAM carries a single object in a datagram.

```
OBJECT_DATAGRAM {
  Type (i) = 0x00..0x0F / 0x20..0x21 / 0x24..0x25 /
             0x28..0x29 / 0x2C..0x2D,
  Track Alias (vi64),
  Group ID (vi64),
  [Object ID (vi64),]
  [Publisher Priority (8),]
  [Properties (..),]
  [Object Status (vi64),]
  [Object Payload (..),]
}
```

Figure 23: MOQT OBJECT_DATAGRAM

The Type field in the OBJECT_DATAGRAM takes the form 0b00X0XXXX (or the set of values from 0x00 to 0x0F, 0x20 to 0x2F). However, not all Type values in this range are valid. The four low-order bits and bit 5 of the Type field determine which fields are present in the datagram:

* The *PROPERTIES* bit (0x01) indicates when the Properties field is present. When set to 1, the Object Properties structure defined in Section 11.2.1.2 is present. When set to 0, the field is absent. If an endpoint receives a datagram with the PROPERTIES bit set and an Properties Length of 0, it MUST close the session with a PROTOCOL_VIOLATION.

* The *END_OF_GROUP* bit (0x02) indicates End of Group. When set to 1, this indicates that no Object with the same Group ID and an Object ID greater than the Object ID in this datagram exists.

* The *ZERO_OBJECT_ID* bit (0x04) indicates when the Object ID field is present. When set to 1, the Object ID field is omitted and the Object ID is 0. When set to 0, the Object ID field is present.

* The *DEFAULT_PRIORITY* bit (0x08) indicates when the Priority field is present. When set to 1, the Priority field is omitted and this Object inherits the Publisher Priority specified in the control message that established the subscription. When set to 0, the Priority field is present.

* The *STATUS* bit (0x20) indicates whether the datagram contains an Object Status or Object Payload. When set to 1, the Object Status field is present and there is no Object Payload. When set to 0, 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 contains the payload.

The following Type values are invalid. If an endpoint receives a datagram with any of these Type values, it MUST close the session with a PROTOCOL_VIOLATION:

* Type values with both the STATUS bit (0x20) and END_OF_GROUP bit (0x02) set: 0x22, 0x23, 0x26, 0x27, 0x2A, 0x2B, 0x2E, 0x2F. An object status message cannot signal end of group.

* Type values that do not match the form 0b00X0XXXX (i.e., Type values outside the ranges 0x00..0x0F and 0x20..0x2F).

If an Object Datagram includes both the STATUS bit and PROPERTIES bit, and the Object Status is not Normal (0x0), the endpoint MUST close the session with a PROTOCOL_VIOLATION, because only Normal Objects can have Properties.

11.4. 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 closed with a PROTOCOL_VIOLATION.

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

11.4.1. Stream Cancellation

Streams aside from the control streams MAY be canceled due to congestion or other reasons by either the publisher or subscriber. Early termination of a unidirectional stream does not affect the MOQT application state, and therefore has no effect on outstanding subscriptions. Termination of a bidi request stream terminates the Subscription, Fetch, Track Status, Publish Namespace, or Subscribe Namespace request. When possible, Publishers SHOULD send a PUBLISH_DONE when terminating a subscription instead of abruptly terminating the associated control stream.

11.4.2. Subgroup Header

All Objects on a Subgroup stream belong to the track identified by Track Alias (see Section 11.1) and the Subgroup indicated by 'Group ID' and Subgroup ID indicated by 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, endpoints MUST allocate connection flow control to the control streams before allocating it to 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..0x15 / 0x18..0x1D / 0x30..0x35 / 0x38..0x3D /
             0x50..0x55 / 0x58..0x5D / 0x70..0x75 / 0x78..0x7D,
  Track Alias (vi64),
  Group ID (vi64),
  [Subgroup ID (vi64),]
  [Publisher Priority (8),]
}
```

Figure 24: MOQT SUBGROUP_HEADER

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

The Type field in the SUBGROUP_HEADER takes the form 0b0XX1XXXX (or the set of values from 0x10 to 0x1F, 0x30 to 0x3F, 0x50 to 0x5F, 0x70 to 0x7F), where bit 4 is always set to 1. However, not all Type values in this range are valid. The four low-order bits and bits 5-6 determine which fields are present in the header:

* The *PROPERTIES* bit (0x01) indicates when the Properties field is present in all Objects in this Subgroup. When set to 1, the Object Properties structure defined in Section 11.2.1.2 is present in all Objects. When set to 0, the field is never present. Objects with no properties set Properties Length to 0.

* The *SUBGROUP_ID_MODE* field (bits 1-2, mask 0x06) is a two-bit field that determines the encoding of the Subgroup ID. To extract this value, perform a bitwise AND with mask 0x06 and right-shift by 1 bit:

  - 0b00: The Subgroup ID field is absent and the Subgroup ID is 0.

  - 0b01: The Subgroup ID field is absent and the Subgroup ID is the Object ID of the first Object transmitted in this Subgroup.

  - 0b10: The Subgroup ID field is present in the header.

  - 0b11: Reserved for future use.

* The *END_OF_GROUP* bit (0x08) indicates that this subgroup contains the largest Object in the Group. When set to 1, the subscriber can infer the final Object in the Group when the data stream is terminated by a FIN. In this case, Objects that have the same Group ID and an Object ID larger than the last Object received on the stream do not exist. This does not apply when the data stream is reset.

* The *DEFAULT_PRIORITY* bit (0x20) indicates when the Priority field is present. When set to 1, the Priority field is omitted and this Subgroup inherits the Publisher Priority specified in the control message that established the subscription. When set to 0, the Priority field is present in the Subgroup header.

* The *FIRST_OBJECT* bit (0x40) indicates that the first object in this subgroup stream is the first object published in the subgroup by the original publisher.

The following Type values are invalid. If an endpoint receives a stream header with any of these Type values, it MUST close the session with a PROTOCOL_VIOLATION:

* Type values with SUBGROUP_ID_MODE set to 0b11: 0x16, 0x17, 0x1E, 0x1F, 0x36, 0x37, 0x3E, 0x3F, 0x56, 0x57, 0x5E, 0x5F, 0x76, 0x77, 0x7E, 0x7F. This mode is reserved for future use.

* Type values that do not match the form 0b0XX1XXXX (i.e., Type values outside the ranges 0x10..0x1F, 0x30..0x3F, 0x50..0x5F, and 0x70..0x7F, or values where bit 4 is not set).

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. If the resulting Object ID would be greater than 2^64 - 1, the endpoint MUST close the session with a PROTOCOL_VIOLATION. 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 a Prior Object ID Gap property (see Section 12.9).

```
{
  Object ID Delta (vi64),
  [Properties (..),]
  Object Payload Length (vi64),
  [Object Status (vi64),]
  [Object Payload (..),]
}
```

Figure 25: MOQT Subgroup Object Fields

11.4.3. 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 reset the stream. This includes, but is not limited to:

* Either of the delivery timeouts defined in Section 8

* Early termination of subscription due to request cancellation

* A publisher’s decision to end the subscription early

* A REQUEST_UPDATE moving the subscription’s End Group to a smaller Group or the Start Location to a larger Location

* Omitting a Subgroup Object due to the subscriber’s Forward State

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 10.11). 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 subsequent reset.

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 determines 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 determined 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 reset the stream. 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 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 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 a reset. 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.

Resets 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.

A publisher that receives a STOP_SENDING on a Subgroup stream SHOULD NOT attempt to open a new stream to deliver additional Objects in that Subgroup. However, if the publisher subsequently receives a REQUEST_UPDATE that changes the Forward State from 0 to 1, it MAY open a new stream to deliver Objects in that Subgroup, as the update indicates the subscriber has renewed interest in forwarded Objects.

The application SHOULD use a relevant error code when resetting a stream, as defined in Section 3.3.3.

11.4.4. 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 (vi64) = 0x5,
  Request ID (vi64),
}
```

Figure 26: MOQT FETCH_HEADER

Each Object sent on a FETCH stream after the FETCH_HEADER has the following format:

```
{
  Serialization Flags (vi64),
  [Group ID Delta (vi64),]
  [Subgroup ID (vi64),]
  [Object ID Delta (vi64),]
  [Publisher Priority (8),]
  [Properties (..),]
  Object Payload Length (vi64),
  [Object Payload (..),]
}
```

Figure 27: MOQT Fetch Object Fields

The Serialization Flags field defines the serialization of the Object. It is a variable-length integer. When less than 128, the bits represent flags described below. The following additional values are defined:

Value	Meaning
0x8C	End of Non-Existent Range
0x10C	End of Unknown Range

Table 7

Any other value is a PROTOCOL_VIOLATION.

11.4.4.1. Flags

The two least significant bits (LSBs) of the Serialization Flags form a two-bit field that defines the encoding of the Subgroup. To extract this value, the Subscriber performs a bitwise AND operation with the mask 0x03.

Bitmask Result (Serialization Flags & 0x03)	Meaning
0x00	Subgroup ID is zero
0x01	Subgroup ID is the prior Object's Subgroup ID
0x02	Subgroup ID is the prior Object's Subgroup ID plus one
0x03	The Subgroup ID field is present

Table 8

The following table defines additional flags within the Serialization Flags field. Each flag is an independent boolean value, where a set bit (1) indicates the corresponding condition is true.

Bitmask	Condition if set	Condition if not set (0)
0x04	Object ID Delta is present	Object ID is the prior Object's ID plus one
0x08	Group ID Delta is present	Group ID is the prior Object's Group ID
0x10	Priority field is present	Priority is the prior Object's Priority
0x20	Properties field is present	Properties field is not present
0x40	Datagram: ignore the two least significant bits	Decode the Subgroup ID as indicated by the two least significant bits

Table 9

The first Object MUST include a Group ID Delta and Object ID Delta, and these values are the absolute Group ID and Object ID. If the first Object in the FETCH response uses a flag that references fields in the prior Object, the Subscriber MUST close the session with a PROTOCOL_VIOLATION.

If the Group ID Delta field is present on an Object other than the first, the Group ID is computed from the Group ID Delta and the prior Object's Group ID. If the Group Order is Ascending, the Group ID is the prior Object's Group ID plus the Group ID Delta + 1. If the Group Order is Descending, the Group ID is the prior Object's Group ID minus the (Group ID Delta + 1). If the computed Group ID would be less than 0 or greater than 2^64-1, the Subscriber MUST close the Session with error 'PROTOCOL_VIOLATION'.

When the Group ID Delta field is present, the Object ID is the value of Object ID Delta if present. When the Group ID Delta field is not present, the Object ID is the prior Object's ID plus the Object ID Delta if present. If Object ID Delta is not present, the Object ID is the prior Object's ID plus one, regardless of which group it belongs to. If the computed Object ID would be greater than 2^64-1, the Subscriber MUST close the Session with error 'PROTOCOL_VIOLATION'.

The Object Properties structure is defined in Section 11.2.1.2.

When encoding an Object with a Forwarding Preference of "Datagram" (see Section 11.2.1.2), the object has no Subgroup ID. The publisher MUST SET bit 0x40 to '1'. When 0x40 is set, it SHOULD set the two least significant bits to zero and the subscriber MUST ignore the bits.

11.4.4.2. End of Range

When Serialization Flags indicates an End of Range (e.g. values 0x8C or 0x10C), the Group ID and Object ID fields are present. Subgroup ID, Priority and Properties are not present. All Objects with Locations between the last serialized Object, if any, and this Location, inclusive, either do not exist (when Serialization Flags is 0x8C) or are unknown (0x10C). A publisher SHOULD NOT use End of Non-Existent Range in a FETCH response except to split a range of Objects that will not be serialized into those that are known not to exist and those with unknown status.

When an Object follows an End of Range indicator and uses flags that reference the "prior Object", the prior Object fields are determined as follows:

* Prior Group ID and prior Object ID: The values from the End of Range indicator.

* Prior Subgroup ID: The Subgroup ID from the last actual Object before the End of Range indicator. If there was no prior Object, using a flag that references the prior Subgroup ID is a PROTOCOL_VIOLATION.

* Prior Priority: The Priority from the last actual Object before the End of Range indicator. If there was no prior Object, using a flag that references the prior Priority is a PROTOCOL_VIOLATION.

11.5. Padding

An endpoint MAY send padding on unidirectional streams or datagrams. Padding does not carry Objects or any other application data. An endpoint can use padding to probe for additional bandwidth while minimizing the impact on the delivery of application data.

To avoid interfering with the delivery of Objects, senders SHOULD send padding streams at a lower priority than any control stream or Object data.

11.5.1. Padding Streams

An endpoint MAY open a unidirectional stream with a stream type of 0x132B3E28 to send padding data. The stream begins with the stream type, followed by zero or more bytes that MUST all be set to zero.

```
PADDING STREAM {
  Type (vi64) = 0x132B3E28,
  Padding Data (..) = 0x00..
}
```

Figure 28: MOQT Padding Stream

The receiver MUST discard all data received on a padding stream to prevent exhausting flow control.

Either the sender or the receiver MAY cancel a padding stream at any time without affecting any MOQT application state.

11.5.2. Padding Datagrams

An endpoint MAY send a datagram with a type of 0x132B3E29 to send padding data. The datagram contains the type followed by zero or more bytes that MUST all be set to zero.

```
PADDING DATAGRAM {
  Type (vi64) = 0x132B3E29,
  Padding Data (..) = 0x00..
}
```

Figure 29: MOQT Padding Datagram

The receiver MUST discard all data received in a padding datagram.

11.6. Examples

Sending a subgroup on one stream:

```
Stream = 2

SUBGROUP_HEADER {
  Type = 0x14
  Track Alias = 2
  Group ID = 0
  Subgroup ID = 0
  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 Properties.

```
Stream = 2

SUBGROUP_HEADER {
  Type = 0x35
  Track Alias = 2
  Group ID = 0
  Subgroup ID = 0
}
{
  Object ID Delta = 0 (Object ID is 0)
  Properties 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)
  Properties Length = 0
  Object Payload Length = 4
  Payload = "efgh"
}
```

12. MOQT Properties

The following Properties are defined in MOQT. Each Property specifies whether it can be used with Tracks, Objects, or both.

Property types in ranges reserved for application-specific use (0x38-0x3F, 0x3800-0x3FFF) are not defined by MOQT. See Section 2.5 for usage guidance.

12.1. SUBGROUP_DELIVERY_TIMEOUT

SUBGROUP_DELIVERY_TIMEOUT (Property Type 0x06) is a Track Property. It is a varint. Its semantics are defined in Section 8.

12.2. OBJECT_DELIVERY_TIMEOUT

OBJECT_DELIVERY_TIMEOUT (Property Type 0x02) is a Track Property. It is a varint. Its semantics are defined in Section 8.

12.3. MAX CACHE DURATION

MAX_CACHE_DURATION (Property Type 0x04) is a Track Property.

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.

If MAX_CACHE_DURATION is not sent by the publisher, the Objects can be cached until implementation constraints cause them to be evicted.

12.4. DEFAULT PUBLISHER PRIORITY

DEFAULT PUBLISHER PRIORITY (Property Type 0x0E) is a Track Property that specifies the priority of a subscription relative to other subscriptions in the same session. The value is from 0 to 255 and lower numbers get higher priority. See Section 7. Priorities above 255 are invalid. Subgroups and Datagrams for this subscription inherit this priority, unless they specifically override it.

If omitted, the Default Publisher Priority is 128.

12.5. DEFAULT PUBLISHER GROUP ORDER

DEFAULT_PUBLISHER_GROUP_ORDER (Property Type 0x22) is a Track Property.

It is an enum indicating the publisher’s preference for prioritizing Objects from different groups within the same subscription (see Section 7). The allowed values are Ascending (0x1) or Descending (0x2). If an endpoint receives a value outside this range, it MUST close the session with PROTOCOL_VIOLATION.

If omitted, the publisher’s preference is Ascending (0x1).

12.6. DYNAMIC GROUPS

DYNAMIC_GROUPS (Property Type 0x30) is a Track Property. The allowed values are 0 or 1. When the value is 1, it indicates that the subscriber can request the Original Publisher to start a new Group by including the NEW_GROUP_REQUEST parameter in PUBLISH_OK or REQUEST_UPDATE for this Track. If an endpoint receives a value larger than 1, it MUST close the session with PROTOCOL_VIOLATION.

If omitted, the value is 0.

12.7. Immutable Properties

Immutable Properties (Property Type 0xB) is a Track or Object Property that contains a sequence of Key-Value-Pairs (see Figure 2) that are themselves Track or Object Properties, respectively.

```
Immutable Properties {
  Type (0xB),
  Length (vi64),
  Key-Value-Pair (..) ...
}
```

This Property can be added by the Original Publisher, but MUST NOT be added by Relays. This Property MUST NOT be modified or removed and the serialization (e.g. variable-length integer encodings) of the Key-Value-Pairs MUST NOT change). Like other Properties, Relays MUST cache Immutable Properties if the Object or Track are cached and MUST forward it. Relays MAY decode and view the Properties in the Key-Value-Pairs.

Unless specified by a particular Property specification, Properties MAY appear either in the mutable property list or inside Immutable Properties. When looking for the value of a property, processors MUST search both the mutable properties and the contents of Immutable Properties.

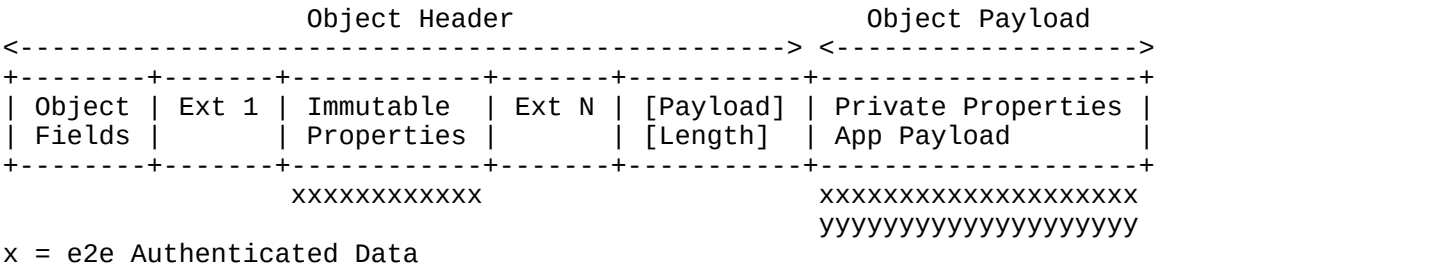
If a Property allows multiple values, the same Property Type MAY appear in both the mutable list and inside Immutable Properties, unless prohibited by the Property specification.

A Track is considered malformed (see Section 2.4.2) if any of the following conditions are detected:

* An Object contains an Immutable Properties property that contains another Immutable Properties key.

* A Key-Value-Pair cannot be parsed.

The following figure shows an example Object structure with a combination of mutable and immutable properties 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

An Object MUST NOT contain more than one instance of this property.

12.8. Prior Group ID Gap

Prior Group ID Gap only applies to Objects, not Tracks.

Prior Group ID Gap (Property Type 0x3C) is a variable length integer containing the number of Groups prior to the current Group that do not and will never exist. 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 2.4.2) 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.

Use of this property is optional, as publishers might not know the prior gap size, 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 property can be added by the Original Publisher, but MUST NOT be added by relays. This property MAY be removed by a relay when the object in question is served via FETCH, and the gap that the property communicates is already communicated implicitly in the FETCH response; it MUST NOT be modified or removed otherwise.

An Object MUST NOT contain more than one instance of this property.

12.9. Prior Object ID Gap

Prior Object ID Gap only applies to Objects, not Tracks.

Prior Object ID Gap (Property Type 0x3E) is a variable length integer containing the number of Objects prior to the current Object that do not and will never exist. 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 2.4.2) 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.

Use of this property is optional, as publishers might not know the prior gap size, or there might 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 property can be added by the Original Publisher, but MUST NOT be added by relays. This property MAY be removed by a relay when the object in question is served via FETCH, and the gap that the property communicates is already communicated implicitly in the FETCH response; it MUST NOT be modified or removed otherwise.

An Object MUST NOT contain more than one instance of this property.

13. Security Considerations

MOQT is a protocol used hop-by-hop between original publishers to relay, (possibly) relay to relay, and relay to end subscribers. Thus, the security considerations need to consider first what happens between two nodes, but also consider the impacts end to end over several hops of MOQT.

MOQT uses a trust model where on each hop the nodes need to be securely identified, authorized to use resources of the peer, provide confidentiality and integrity to prevent third party attacks and limit monitoring and leakage of privacy sensitive information. The relays within the chain from original publisher to end subscribers will have access to Track names, Track Properties, Object Properties, as well as the object's content unless it is end-to-end encrypted Section 13.4.

Publishers, including Relays, require authorization to prevent unauthorized subscriptions to content. Subscription requests can carry authorization tokens (see Section 13.3) to prove the subscriber's right to access specific tracks or namespaces. Relays that aggregate subscriptions from multiple downstream subscribers MUST ensure each subscriber is independently authorized.

13.1. Subscription Amplification

A malicious subscriber could attempt to overwhelm a publisher or relay by requesting subscriptions to many tracks simultaneously. Relays SHOULD implement rate limiting on subscription requests and MAY reject excessive subscriptions with REQUEST_ERROR using the EXCESSIVE_LOAD error code. Publishers SHOULD monitor the number of active subscriptions and enforce limits to prevent resource exhaustion from a single subscriber or session.

TODO: Describe Cache Poisoning attacks

13.2. Communication Security

MOQT depends on a secure transport to provide confidentiality, integrity and endpoint authentication between subscriber and publisher. Implementations use QUIC or WebTransport to fulfill the basic communication security requirements and these implementations SHOULD follow best practices for TLS 1.3 and QUIC. Relays MUST use authentication to prevent impersonation.

Note that the basic security protection offered by QUIC or TCP/TLS does not prevent traffic pattern analysis. Object sizes, sizes of request messages, etc can make it possible for a third party observer to identify media content, user patterns and media stream origin.

13.3. Authorization

MOQT supports authorization via mutual TLS for node-level identification and token-based schemes for fine-grained access control.

Mutual TLS is expected to be widely used for node level identification between relays, especially within one organization. However, in some deployments mutual TLS can also be used for end subscribers or original publishers. However, as only node level authentication is provided, what a particular identified node is allowed to do is not provided at TLS level.

MOQT has functionality to carry Authorization tokens as message parameters. These tokens can vary based on the application requirements. Two variants of authorization tokens have already been defined for MOQT, and more are expected in the future. The current tokens are Privacy Pass Authentication for Media over QUIC [PPA] and Authentication scheme for MOQT using Common Access Tokens [CAT].

Tokens are expected to contain information about which actions and which resources the endpoint providing the token is authorized to perform and access. Relays will verify the token to ensure that the request is authorized.

13.3.1. Replay Attacks

Replay protection for authorization tokens is the responsibility of the specific token scheme used. Token schemes such as [CAT] and [PPA] include requirements for relays when processing tokens and requests.

13.4. Media Security

MOQT uses secure transports that provide confidentiality and integrity protection. However, media objects are accessible to relays, and are subject to both intentional and accidental modification, unless they are additionally end-to-end protected.

The media objects transported by MOQT in various tracks from various original publishers are subject to several considerations. The first is source authenticity, i.e. to know that the received media objects are what the original publisher actually published. In addition to

the media objects, it can also be important to authenticate some Track and Object Properties. For example, timestamps are crucial to understand where on the timeline this media fragment belongs.

The second aspect is content confidentiality. Beyond direct relay access to media objects, object sizes and traffic patterns enable analysis of content. Track namespace and track name can also be analyzed and correlated between end subscribers by relays.

The end-to-end media security is handled by mechanisms external to this specification. They need to provide source authenticity and confidentiality. MOQT's object model does enable both the object data itself as well as Object Properties to be confidentiality and integrity protected. MOQT also supports Object Properties being integrity protected but not encrypted.

Current proposals for media security include: - An E2EE scheme based on SFRAME: [I-D.ietf-moq-secure-objects].

Secure key distribution for end-to-end encryption is specific to the encryption system and deployment, and outside the scope of this document.

13.5. 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 one with the lowest priority, after reaching a resource limit.

13.6. 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 timeouts.

13.6.1. Idle Connection Handling

The transport connection (e.g., QUIC) underlying a MOQT session can close due to idle timeout if no data is exchanged, either because there are no established subscriptions or the established subscriptions are not publishing Objects frequently. This includes publisher sessions that have issued a PUBLISH_NAMESPACE and are waiting for subscribers.

Implementations that want to keep idle sessions open have several options:

* Use transport-layer keep-alive mechanisms, such as QUIC PING frames, to prevent idle timeout closure.

* Send periodic control messages, for example REQUEST_UPDATE with no modified Message Parameters.

* Accept that idle connections can close and implement reconnection logic when needed.

The choice of mechanism is implementation-specific.

13.7. Relay security considerations

13.7.1. State maintenance

A Relay SHOULD have mechanisms to prevent malicious endpoints from flooding it with PUBLISH_NAMESPACE, SUBSCRIBE_NAMESPACE, or SUBSCRIBE_TRACKS requests that could bloat data structures. It could use QUIC stream limits to limit the number of such requests, or could have application-specific policies that can reject incoming requests that cause the state maintenance for the session to be excessive.

13.7.2. SUBSCRIBE_NAMESPACE and SUBSCRIBE_TRACKS 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 or SUBSCRIBE_TRACKS message with a short prefix, it can cause the relay to send a large volume of NAMESPACE or PUBLISH messages. As changes occur in the tree of namespaces, the relay would have to send matching NAMESPACE/ NAMESPACE_DONE messages or initiate new PUBLISH streams.

13.8. Implementation Identification Fingerprinting

The MOQT_IMPLEMENTATION option (Section 10.3.1.5) can reveal information that contributes to fingerprinting, a set of techniques for identifying a specific endpoint over time through its unique set of characteristics.

Detailed implementation information, including specific version numbers, build identifiers, or platform details, can create a unique fingerprint that enables tracking endpoints across sessions without their awareness. When combined with other session characteristics, even minimal implementation identification can contribute to distinguishing one endpoint from another.

To mitigate fingerprinting risks:

* Implementations SHOULD send only the minimum information necessary for interoperability debugging. A short implementation name and major version number are typically sufficient.

* Implementations SHOULD NOT include detailed system information, build numbers, or other attributes that could uniquely identify a specific instance or user.

* Privacy-conscious deployments MAY omit the MOQT_IMPLEMENTATION option entirely or send a generic value.

* Implementations MAY provide users with the ability to configure or disable the MOQT_IMPLEMENTATION option.

Operators are advised that detailed implementation identification facilitates the same privacy concerns as persistent identifiers, since it enables correlation of sessions across time.

14. Grease

To ensure that implementations correctly handle unknown values and do not fail when encountering protocol extensions they do not understand, this document reserves a range of values for the purpose of greasing; see Section 3.3 of [RFC9170].

Grease values follow the pattern 0x7f * N + 0x9D for non-negative integer values of N (that is, 0x9D, 0x11C, ..., 0x3fffffffffffde).

The following registries include GREASE reservations:

* Setup Options (Section 15.4)

* Properties (Section 15.8)

* Session Termination Error Codes (Section 15.10.1)

* REQUEST_ERROR Codes (Section 15.10.2)

* PUBLISH_DONE Codes (Section 15.10.3)

* Stream Reset Error Codes (Section 15.10.4)

* MOQT Auth Token Type

Because new values in these registries can be defined without negotiation, implementations MUST handle unknown values gracefully. Endpoints MUST NOT close the session solely because they received an unknown value. The following rules apply:

Setup Options with reserved identifiers have no semantics and can carry arbitrary values. Endpoints MUST ignore unknown Setup Options as specified in Section 10.3.

Unknown Properties MUST be handled as specified in Section 2.5.

Receipt of an unknown error code in any error context (Session Termination, REQUEST_ERROR, PUBLISH_DONE, or Data Stream Reset) MUST be treated as equivalent to INTERNAL_ERROR for that context. An endpoint MUST NOT close the session because it received an unknown error code in a REQUEST_ERROR or PUBLISH_DONE.

15. IANA Considerations

TODO: fill out currently missing registries:

* MOQT ALPN values

* Message types

* Session-Level Track Names

15.1. URI Scheme Registrations

This document requests the registration of the following URI schemes in the "Uniform Resource Identifier (URI) Schemes" registry, per [RFC7595]:

15.1.1. "moqt" URI Scheme Registration

Scheme name: moqt

Status: Permanent

Applications/protocols that use this scheme name: Media over QUIC Transport (MOQT) over native QUIC or WebTransport, as defined in this document.

Contact: IETF MoQ Working Group (moq@ietf.org)

Change controller: IETF

References: This document

15.2. Media Type Registration

This document registers the following media type in the "Media Types" registry [RFC6838]:

Type name: application

Subtype name: moqt

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: This media type is used to identify resources accessed via the moqt URI scheme. It is not used to label the content of MOQT objects, which are defined by separate media types in application-specific specifications.

Security considerations: See the Security Considerations section of this document.

Interoperability considerations: N/A

Published specification: This document

Applications that use this media type: Implementations of the Media over QUIC Transport (MOQT) protocol.

Fragment identifier considerations: Fragment identifiers for application/moqt follow the syntax defined in Section 3.1.2.

Additional information: N/A

Contact: IETF MoQ Working Group (moq@ietf.org)

Change controller: IETF

15.3. MOQT URI Fragment Types

This document establishes the "MOQT URI Fragment Types" registry. This registry governs fragment type identifiers used in moqt URI fragments as defined in Section 3.1.2.

New fragment type identifiers are registered using the Specification Required policy ([RFC8126], Section 4.6).

Each entry in the registry contains the following fields:

| Fragment Type | Description | Specification | |:--------------|:------------|:--------------|

This registry is initially empty.

15.4. Setup Options

Type	Name	Specification
0x01	PATH	Section 10.3.1.2
0x03	AUTHORIZATION_TOKEN	Section 10.3.1.4
0x04	MAX_AUTH_TOKEN_CACHE_SIZE	Section 10.3.1.3
0x05	AUTHORITY	Section 10.3.1.1
0x07	MOQT_IMPLEMENTATION	Section 10.3.1.5
0x7f * N + 0x9D	Reserved for greasing	Section 14

Table 10

Endpoints MUST ignore unknown Setup Options as specified in Section 10.3.

New Setup Option types are registered using the Specification Required policy ([RFC8126], Section 4.6). Provisional registrations are permitted to allow experimentation and avoid codepoint collisions between independent implementations. There is no reserved range for private or application-specific use; implementations that need custom Setup Options SHOULD request a provisional registration.

15.5. Authorization Token Alias Type

Code	Name	Specification
0x0	DELETE	Section 10.2.2
0x1	REGISTER	Section 10.2.2
0x2	USE_ALIAS	Section 10.2.2
0x3	USE_VALUE	Section 10.2.2

Table 11

15.6. MOQT Auth Token Type

Code	Name	Specification
0x0	Reserved	Section 10.2.2
0x7f * N + 0x9D	Reserved for greasing	Section 14

Table 12

15.7. Message Parameters

Parameter Type	Parameter Name	Specification
0x02	OBJECT_DELIVERY_TIMEOUT	Section 10.2.4
0x03	AUTHORIZATION_TOKEN	Section 10.2.2
0x04	RENDEZVOUS_TIMEOUT	Section 10.2.6
0x06	SUBGROUP_DELIVERY_TIMEOUT	Section 10.2.3

0x08	EXPIRES	Section 10.2.10
0x09	LARGEST_OBJECT	Section 10.2.11
0x0A	FILL_TIMEOUT	Section 10.2.5
0x10	FORWARD	Section 10.2.12
0x20	SUBSCRIBER_PRIORITY	Section 10.2.7
0x21	SUBSCRIPTION_FILTER	Section 10.2.9
0x22	GROUP_ORDER	Section 10.2.8
0x32	NEW_GROUP_REQUEST	Section 10.2.13
0x34	TRACK_NAMESPACE_PREFIX	Section 10.2.14

Table 13

* Message Parameters - List which params can be repeated in the table.

15.8. Properties

Type	Name	Scope	Specification
0x02	OBJECT_DELIVERY_TIMEOUT	Track	Section 12.2
0x04	MAX_CACHE_DURATION	Track	Section 12.3
0x06	SUBGROUP_DELIVERY_TIMEOUT	Track	Section 12.1
0x0B	IMMUTABLE_PROPERTIES	Track, Object	Section 12.7
0x0E	DEFAULT_PUBLISHER_PRIORITY	Track	Section 12.4
0x22	DEFAULT_PUBLISHER_GROUP_ORDER	Track	Section 12.5
0x30	DYNAMIC_GROUPS	Track	Section 12.6
0x3C	PRIOR_GROUP_ID_GAP	Object	Section 12.8
0x3E	PRIOR_OBJECT_ID_GAP	Object	Section 12.9

0x7f * N + 0x9D	Reserved for greasing	Any	Section 14

Table 14

The following table contains provisional registrations for other active drafts in the moq wg. These entries share the same Property Type space as the table above.

Type	Name	Scope	Specification
0x06	TIMESTAMP	Object	draft-ietf-moq-loc
0x08	TIMESCALE	Track, Object	draft-ietf-moq-loc
0x0A	VIDEO_FRAME_MARKING	Object	draft-ietf-moq-loc
0x0C	AUDIO_LEVEL	Object	draft-ietf-moq-loc
0x0D	VIDEO_CONFIG	Object	draft-ietf-moq-loc

Table 15

Endpoints MUST ignore unknown Property types, skipping them using the length field.

* MOQ Properties - we wish to define the following registration policies:

  - 0x00 to 0x77: Standards Action or IESG Approval (1-byte encoding)

  - 0x78 to 0x7F: Reserved for application-specific use (1-byte encoding, no registration permitted)

  - 0x80 to 0x37FF: Specification Required (2-byte encoding)

  - 0x3800 to 0x3FFF: Reserved for application-specific use (2-byte encoding, no registration permitted)

  - 0x4000 to 0x7FFF: Reserved for Mandatory Track Properties (see Section 2.5.1). Properties registered in this range MUST have Track scope; Object scope properties MUST NOT be registered in this range.

- 0x8000 and above: First Come First Served

Code points reserved for application-specific use will never be allocated by IANA. Applications using these values do not need to coordinate with IANA. Note that applications consuming tracks from uncoordinated sources may encounter different semantics for the same code points, creating potential collision risks.

15.9. Session-Level Track Names

This document establishes a registry for session-level track names under the .session namespace (see Section 3.2.2). The registration policy is Specification Required (per [RFC8126], Section 4.6).

Each registration must include:

Field	Description
Track Namespace	The track namespace under the .session namespace, can be empty
Track Name	The track name (bytes) within the full namespace
Description	Brief description of the track's purpose
Change Controller	Who may update the registration
Specification	Reference to the defining specification

Table 16

This document does not define any initial entries.

15.10. Error Codes

15.10.1. Session Termination Error Codes

Name	Code	Specification
NO_ERROR	0x0	Section 3.5
INTERNAL_ERROR	0x1	Section 3.5
UNAUTHORIZED	0x2	Section 3.5
PROTOCOL_VIOLATION	0x3	Section 3.5
INVALID_REQUEST_ID	0x4	Section 3.5
DUPLICATE_TRACK_ALIAS	0x5	Section 3.5
KEY_VALUE_FORMATTING_ERROR	0x6	Section 3.5
INVALID_PATH	0x8	Section 3.5
MALFORMED_PATH	0x9	Section 3.5
GOAWAY_TIMEOUT	0x10	Section 3.5
CONTROL_MESSAGE_TIMEOUT	0x11	Section 3.5
DATA_STREAM_TIMEOUT	0x12	Section 3.5
AUTH_TOKEN_CACHE_OVERFLOW	0x13	Section 3.5
DUPLICATE_AUTH_TOKEN_ALIAS	0x14	Section 3.5
VERSION_NEGOTIATION_FAILED	0x15	Section 3.5
MALFORMED_AUTH_TOKEN	0x16	Section 3.5
UNKNOWN_AUTH_TOKEN_ALIAS	0x17	Section 3.5
EXPIRED_AUTH_TOKEN	0x18	Section 3.5
INVALID_AUTHORITY	0x19	Section 3.5
MALFORMED_AUTHORITY	0x1A	Section 3.5
Reserved for greasing	0x7f * N + 0x9D	Section 14

Table 17

15.10.2. REQUEST_ERROR Codes

Name	Code	Specification
INTERNAL_ERROR	0x0	Section 10.6
UNAUTHORIZED	0x1	Section 10.6
TIMEOUT	0x2	Section 10.6
NOT_SUPPORTED	0x3	Section 10.6
MALFORMED_AUTH_TOKEN	0x4	Section 10.6
EXPIRED_AUTH_TOKEN	0x5	Section 10.6
GOING_AWAY	0x6	Section 10.6
EXCESSIVE_LOAD	0x9	Section 10.6
DOES_NOT_EXIST	0x10	Section 10.6
INVALID_RANGE	0x11	Section 10.6
MALFORMED_TRACK	0x12	Section 10.6
DUPLICATE_SUBSCRIPTION	0x19	Section 10.6
UNINTERESTED	0x20	Section 10.6
PREFIX_OVERLAP	0x30	Section 10.6
NAMESPACE_TOO_LARGE	0x31	Section 10.6
INVALID_JOINING_REQUEST_ID	0x32	Section 10.6
UNSUPPORTED_EXTENSION	0x33	Section 10.6
REDIRECT	0x34	Section 10.6
Reserved for greasing	0x7f * N + 0x9D	Section 14

Table 18

15.10.3. PUBLISH_DONE Codes

Name	Code	Specification
INTERNAL_ERROR	0x0	Section 10.11
UNAUTHORIZED	0x1	Section 10.11
TRACK_ENDED	0x2	Section 10.11
SUBSCRIPTION_ENDED	0x3	Section 10.11
GOING_AWAY	0x4	Section 10.11
TOO_FAR_BEHIND	0x5	Section 10.11
EXPIRED	0x6	Section 10.11
UPDATE_FAILED	0x8	Section 10.11
EXCESSIVE_LOAD	0x9	Section 10.11
MALFORMED_TRACK	0x12	Section 10.11
Reserved for greasing	0x7f * N + 0x9D	Section 14

Table 19

15.10.4. Stream Reset Error Codes

Name	Code	Specification
INTERNAL_ERROR	0x0	Section 3.3.3
CANCELLED	0x1	Section 3.3.3
DELIVERY_TIMEOUT	0x2	Section 3.3.3
SESSION_CLOSED	0x3	Section 3.3.3
GOING_AWAY	0x4	Section 3.3.3
TOO_FAR_BEHIND	0x5	Section 3.3.3
UNKNOWN_OBJECT_STATUS	0x6	Section 3.3.3

EXPIRED_AUTH_TOKEN	0x7	Section 3.3.3
EXCESSIVE_LOAD	0x9	Section 3.3.3
MALFORMED_TRACK	0x12	Section 3.3.3
Reserved for greasing	0x7f * N + 0x9D	Section 14

Table 20

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

Use of Generative AI

Anthropic’s Claude was used to assist with drafting and editing text for this document. All AI-generated content was reviewed and approved by the editors.

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>.

[I-D.ietf-webtrans-overview]
Kinnear, E. and V. Vasiliev, "The WebTransport Protocol Framework", Work in Progress, Internet-Draft, draft-ietf-webtrans-overview-12, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-overview-12>.

[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>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/rfc/rfc3629>.

[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>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/rfc/rfc6838>.

[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>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-editor.org/rfc/rfc7595>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[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>.

[RFC9221] Pauly, T., Kinnear, E., and D. Schinazi, "An Unreliable Datagram Extension to QUIC", RFC 9221, DOI 10.17487/RFC9221, March 2022, <https://www.rfc-editor.org/rfc/rfc9221>.

[WebTransport] Frindell, A., Kinnear, E., and V. Vasiliev, "WebTransport over HTTP/3", Work in Progress, Internet-Draft, draft-ietf-webtrans-http3-15, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-http3-15>.

Informative References

[CAT] Law, W., Lemmons, C., Simon, G., and S. Nandakumar, "Authentication scheme for MOQT using Common Access Tokens", Work in Progress, Internet-Draft, draft-ietf-moq-c4m-00, 19 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-moq-c4m-00>.

[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-moq-secure-objects]
Jennings, C. F., Nandakumar, S., and R. Barnes, "End-to-End Secure Objects for Media over QUIC Transport", Work in Progress, Internet-Draft, draft-ietf-moq-secure-objects-00, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-moq-secure-objects-00>.

[I-D.ietf-scone-protocol]
Thomson, M., Huitema, C., Oku, K., Joras, M., and L. M. Ihlar, "Standard Communication with Network Elements (SCONE) Protocol", Work in Progress, Internet-Draft, draft-ietf-scone-protocol-04, 14 December 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-scone-protocol-04>.

[PPA]
Nandakumar, S., Jennings, C. F., and T. Meunier, "Privacy Pass Authentication for Media over QUIC (MoQ)", Work in Progress, Internet-Draft, draft-ietf-moq-privacy-pass-auth-02, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-moq-privacy-pass-auth-02>.

[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>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC8470] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <https://www.rfc-editor.org/rfc/rfc8470>.

[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>.

[RFC9170] Thomson, M. and T. Pauly, "Long-Term Viability of Protocol Extension Mechanisms", RFC 9170, DOI 10.17487/RFC9170, December 2021, <https://www.rfc-editor.org/rfc/rfc9170>.

[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-17

Session and Control Plane

* Unified moqt:// URI scheme for QUIC and WebTransport (#1486)

* Add fragment identifier support for moqt URIs (#1571)

* Split SUBSCRIBE_NAMESPACE into SUBSCRIBE_NAMESPACE and SUBSCRIBE_TRACKS (#1542)

* Remove Required Request ID (#1615)

* Add REDIRECT for request errors and established subscriptions (#1534)

* Allow GOAWAY on request streams to migrate individual requests (#1617)

* Add Request ID to GOAWAY (#1559)

* Remove PUBLISH_OK message type, make it a REQUEST_OK alias (#1611)

* Generalize stream reset codes to all request streams, add new codes, align with PUBLISH_DONE (#1606)

* Add Track Properties to REQUEST_OK (#1576)

* Add support for mandatory-to-understand track extensions (#1509)

* Exclude your own tracks from SUBSCRIBE_NAMESPACE (#1596)

* Add Session-Level Tracks reserved namespace (#1562)

* Allow coalescing REQUEST_UPDATE processing (#1540)

* SUBSCRIBE takes precedence over SUBSCRIBE_NAMESPACE at relay (#1533)

* Don’t close the Session for unknown errors (#1561)

* Clarify REQUEST_UPDATE failure behavior for all request types (#1539)

* Clarify SUBSCRIBE_NAMESPACE stream closure semantics (#1541)

* FETCH to a track with no objects returns INVALID_RANGE (#1537)

* Clarify FETCH_OK End Location semantics (#1536)

* Clarify definition of scope (#1629)

* Clarify Joining Fetch behavior:

  - Joining FETCH is unaffected by forward changing to 0 (#1620)

  - Joining Fetch forward state mismatch is a request error (#1609)

  - Clarify Joining Fetch ordering with Forward State transitions (#1577)

Data Plane Wire Format and Handling

* Make Object ID and Group ID delta encoded in Fetch responses (#1586)

* Add FIRST_OBJECT bit to SUBGROUP_HEADER type (#1618)

* Split DELIVERY_TIMEOUT into two types of timeout (#1605)
* FILL_TIMEOUT parameter (#1490)
* Forbid relays from lying about LARGEST_OBJECT (#1621)
* Allow publisher to reopen subgroup after REQUEST_UPDATE forward 0->1 (#1583)
* Allow 7-byte varint and non-minimal encodings (#1595)
* Padding streams and datagrams (#1475)
* Close session when delta encoding wraps (#1560)

Notable Editorial Changes

* Clarify Object existence and cross-source contradictions (#1566)
* Clarify immutable track properties (#1535)
* Improve Startup Latency and 0-RTT guidance (#1544)
* Improve Security Considerations section (#1625)
* Rewrite abstract and introduction (#1556)
* Define textual aliases for REQUEST_OK by request type (#1610)
* Add IANA registry for Setup Options (#1564)
* Add provisional registry for LOC properties (#1624)
* Update MOQ Properties registration policies (#1525)
* Add stream type column to message type table (#1555)
* Fix grease examples to match 0x7f multiplier (#1569)

A.2. Since draft-ietf-moq-transport-16

Session and Control Plane

* Change control stream from bidi to a pair of uni streams (#1510)
* Collapse CLIENT_SETUP and SERVER_SETUP into a single SETUP message (#1510)

* Move requests to bidirectional streams; remove cancel messages (#1389)
* Remove MAX_REQUEST_ID/REQUESTS_BLOCKED (#1471)
* New variable-length integer encoding (#1016)
* Encode Message Parameters as Type-Value pairs (#1462)
* Add GREASE for Setup Options, Properties, and error code registries (#1460)
* Add RENDEZVOUS_TIMEOUT parameter for SUBSCRIBE (#1447)
* Add PUBLISH_BLOCKED message for SUBSCRIBE_NAMESPACE flow control (#1452)
* Add Timeout field to GOAWAY message (#1497)
* Add GOING_AWAY to REQUEST_ERROR codes (#1434)
* Add EXCESSIVE_LOAD error code (#1479)
* Add NAMESPACE_TOO_LARGE error and stream reset for large namespaces (#1496)
* Add TOO_FAR_BEHIND stream reset code (#1445)
* Add REQUEST_UPDATE to list of REQUEST_ERROR causes (#1466)
* Enforce REQUEST_OK/ERROR as first message on the response stream (#1499)
* Allow joining FETCH for PUBLISH and REQUEST_UPDATE with forward=1 (#1335)
* Allow DELIVERY_TIMEOUT value of 0 to mean no timeout (#1450)
* Allow zero-element namespaces (#1472)
* Clarify EXPIRES parameter update mechanism (#1448)
* Remove TRACK_STATUS from REQUEST_UPDATE (#1436)
* Define how to use auth token cache safely with multiple streams (#1430)
* Constrain encoding/parsing of track namespace and names (#1512)

* Reserve Property type ranges for application-specific use (#1473)

* Make EndGroup in Subscription Filters a delta (#1470)

* Copy DELIVERY_TIMEOUT min requirement from parameter to property (#1427)

Data Plane Wire Format and Handling

* Clarify prior Object semantics after End of Range indicators in FETCH (#1513)

* Clarify datagram status and properties cases (#1444)

* Clarify Stream Count includes empty subgroups (#1449)

* Clarify language for malformed tracks in a subgroup with END_OF_GROUP (#1464)

* Properties can appear in mutable list or inside Immutable Properties (#1442)

* Clarify immutable property preservation requirements (#1441)

* Clarification for Track Alias uniqueness (#1418)

Notable Editorial Changes

* Rename Setup Parameters to Setup Options (#1461)

* Rename Extension Headers to Properties (#1504)

* Add security/privacy considerations for MOQT_IMPLEMENTATION (#1511)

* Add editorial text on bandwidth probing techniques (#1477)

* Explain idle connection handling (#1443)

* Fix "SUBSCRIBE_NAMESPACE with short prefixes" (#1502)

* Add generative AI disclosure per IRTF guidelines

A.3. Since draft-ietf-moq-transport-15

Setup and Control Plane

* Delta encode Key-Value-Pairs for Parameters and Headers (#1315)

* Use Request ID in PUBLISH_NAMESPACE_{DONE/CANCEL} (#1329)

* Remove delivery related params from TRACK_STATUS for Subscribers (#1325)

* PUBLISH does not imply PUBLISH_NAMESPACE (#1364)

* Allow Start Location to decrease in SUBSCRIBE_UPDATE (#1323)

* Change SUBSCRIBE_UPDATE to REQUEST_UPDATE and expand ability to update (#1332)

* Put SUBSCRIBE_NAMESPACE on a stream, make Namespaces and PUBLISH independent (#1344)

* Require NAMESPACE before NAMESPACE_DONE (#1392)

* Allow the '*' or the empty namespace in SUBSCRIBE_NAMESPACE (#1393)

* Relays match SUBSCRIBE to both Tracks and Namespaces (#1397)

* Clarify sending requests after sending GOAWAY (#1398)

* Add Retry Interval to REQUEST_ERROR (#1339)

* Add Extension Headers to PUBLISH, SUBSCRIBE_OK, and FETCH_OK (#1374)

* Move track properties to extensions, scope parameters (#1390)

* Add LARGEST_OBJECT parameter to TRACK_STATUS (#1367)

* Duplicate subscription processing (#1341)

* Address Track Name/Namespace edge cases (#1399)

Data Plane Wire Format and Handling

* Enable mixing datagrams with streams in one track (#1350)

* Clarify datagrams and subgroups (#1382)

* Redo the way we deal with missing Objects and Object Status (#1342)

* Allow unknown ranges in a FETCH response (#1331)

* Do not reopen subgroups after delivery timeout or STOP_SENDING (#1396)

* Clarify handling of unknown extensions (#1395)

* Clarify Delivery Timeout for datagrams (#1406)

* Disallow DELIVERY_TIMEOUT=0 (#1330)

* Malformed track due to multiple priorities for one subgroup (#1317)

Notable Editorial Changes

* Subscribers can migrate networks too (#1410)

* Rename Version Specific Parameters to Message Parameters (#1411)

* Clarify valid joining fetch subscription states (#1363)

* Formatting names for logs (#1355)

* A Publisher might not use the congestion window (#1408)

A.4. Since draft-ietf-moq-transport-14

Setup and Control Plane

* Always use ALPN for version negotiation (#499)

* Consolidate all the Error Message types (#1159)

* Change MOQT IMPLEMENTATION code point to 0x7 (#1191)

* Add Forward to SUBSCRIBE_NAMESPACE (#1220)

* Parameters for Group Order, Subscribe Priority and Subscription Filter (redo) (#1273)

* REQUEST_OK message (#1274)

* Subscribe Update Acknowledgements (#1275)

* Disallow DELETE and USE_ALIAS in CLIENT_SETUP (#1277)

* Remove Expires field from SUBSCRIBE_OK (#1282)

* Make Forward a Parameter (#1283)

* Allow SUBSCRIBE_UPDATE to increase the end location (#1288)

* Add default port for raw QUIC (#1289)

* Unsubscribe Namespace should be linked to Subscribe Namespace (#1292)

Data Plane Wire Format and Handling

* Fetch Object serialization optimization (#949)

* Make default PUBLISHER PRIORITY a parameter, optional in Subgroup/ Datagram (#1056)

* Allow datagram status with object ID=0 (#1197)

* Disallow object extension headers in all non-Normal status objects (#1266)

* Objects for malformed track must not be cached (#1290)

* Remove NO_OBJECTS fetch error code (#1303)

* Clarify what happens when max_cache_duration parameter is omitted (#1287)

Notable Editorial Changes

* Rename Request ID field in MAX_REQUEST_ID (#1250)

* Define and draw subscription state machine (#1296)

* Omitting a subgroup object necessitates reset (#1295)

* Define duplication rules for header extensions (#1293)

* Clarify joining fetch end location (#1286)

A.5. 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.6. 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.7. 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.8. 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