Independent Submission
Internet-Draft
Intended status: Informational
Expires: 14 September 2023

S. Nandakumar
Cisco
C. Huitema
Private Octopus Inc.
W. Law
Akamai

13 March 2023

MoQ Transport (moqt) - Unified Media Delivery Protocol over QUIC
draft-nandakumar-moq-transport-00

Abstract

This specification defined MoqTransport (moqt), an unified media delivery protocol over QUIC. It aims at supporting multiple application classes with varying latency requirements including ultra low latency applications such as interactive communication and gaming. It is based on a publish/subscribe metaphor where entities publish and subscribe to data that is sent through, and received from, relays in the cloud. The data is delivered in the strict priority order. The information subscribed to is named such that this forms an overlay information centric network. The relays allow for efficient large scale deployments.

Status of This Memo

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

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

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

This Internet-Draft will expire on 14 September 2023.

Table of Contents

1. Introduction

Recently new use cases have emerged requiring higher scalability of delivery for interactive realtime applications and much lower latency for streaming applications and a combination thereof. On one side are use cases such as normal web conferences wanting to distribute out to millions of viewers and allow viewers to instantly move to being a presenter. On the other side are uses cases such as streaming a soccer game to millions of people including people in the stadium watching the game live. Viewers watching an e-sports event want to be able to comment with minimal latency to ensure the interactivity aspects between what different viewers are seeing is preserved. All of these uses cases push towards latencies that are in the order of 100ms over the natural latency the network causes.

Interactive realtime applications, such as web conferencing systems, require ultra low latency (< 150ms) delivery. Such applications create their own application specific delivery network over which latency requirements can be met. Realtime transport protocols such as RTP over UDP provide the basic elements needed for realtime communication, both contribution and distribution, while leaving aspects such as resiliency and congestion control to be provided by each application. On the other hand, media streaming applications are much more tolerant to latency and require highly scalable media distribution. Such applications leverage existing CDN networks, used for optimizing web delivery, to distribute media. Streaming protocols such as HLS and MPEG-DASH operates on top of HTTP and gets transport-level resiliency and congestion control provided by TCP.

This specification defines MOQTransport, a publish and subscribe based media delivery protocol over QUIC, where the principal idea is entities publish unique named objects that are end-to-end encrypted and consume data by subscribing to the named objects. The names used are scoped and authorized to the domain operating the application server (referred to as Origin/Provider in this specification).

The published data carry metadata identifying relative priority, time-to-live and other useful metadata that's authenticated for components implementing Relay functions to make drop/forwarding decisions. MoQTransport is designed to make it easy to implement relays so that fail over could happen between relays with minimal impact to the clients and relays can redirect a client to a different relay.

1.1. 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.

Commonly used terms in this document are described below:

* Provider/Origin: Entity capable of hosting media application session based on the MoQTransport. It is responsible for authorizing the publishers/subscribers, managing the names used for Tracks and is scoped under domain for a specific application. In certain deployments, a provider is also responsible for establishing trust between clients and relays for delivering media.

* Emitter: Authorized entities that participate in a MoQTransport Session under an Provider. Emitters are trusted with E2E encryption keys. They operate on one or more uncompressed media inputs, compress and possible encrypt it and send over Data Streams. Each such encoded and/or encrpyted stream corresponds to a Track within the MoQTransport.

* Catalog Maker: Entities performing Catalog Maker role compose or aggregate tracks from multiple emissions to form a new emission. Akin to the role of entities with the Relay role, Catalog Maker role entities are not trusted with the E2E keys and they perform publisher and subscriber roles. Catalog Makers are allowed to publish tracks with a new name without changing the media content of the received tracks.

* Control Stream: QUIC Stream to exchange control message to setup appropriate context for media delivery and is scoped to a given QUIC Connection. Functionally, Control Messages enable authorization of names for tracks, setting up media properties and starting/terminating media sessions.

* Data Stream: QUIC Stream or QUIC Datagram based transport for delivering end to end encrypted application media objects. Such objects shall carry metadata (unencrypted) for Relays to make store/forwarding decisions along with the application payload.

2. Object Model

This section define various concepts that make up the object model enabling media delivery over QUIC.

2.1. Tracks

Tracks form the central concept within the MoQ Transport protocol for delivering media. A Track identifies the namespace and the authorization scope under which MoQTransport objects Section 2.2 are

delivered.

A track is a transform of a uncompresss media using a specific encoding process, a set of parameters for that encoding, and possibly an encryption process.

The MoQTransport is designed to transport tracks.

Tracks have the following properties:

* Tracks MUST be owned by a single authorized MoQ Entity, such as an Emitter or a Catalog Maker, under a single provider domain.

* Tracks MUST have a single encoding configuration.

* Tracks MUST have a single security configuration.

Tracks are identified by a globally unique identifier, called "Track ID". Track ID MUST identify its owning provider by a standardized identifier, such as domain name or equivalent, then followed by the application context specific "Track Name", encoded a opaque string.

2.2. Objects

The binary content of a track is composed of a sequence of objects. An Object is the smallest unit that makes sense to decode and may not be independently decodable. An Object MUST belong to a group Section 2.3

Few examples include, for video media an object could be an H.264 P frame or could be just a single slice from inside the P Frame. For audio media, it could be a single audio frame. Or a catalog payload.

Objects are not partially decodable. The end to end encryption and authentication operations are performed across the whole object, thus rendering partial objects unusable.

Objects MUST be uniquely identifiable within the MoQ delivery system. Objects carry associated header/metadata containining priority, time to live, and other information aiding the caching/forwarding decision at the Relays. Objects MAY be optionally cached at Relays. The content of the Objects are opaque to Relays and delivered on the strict priority order Section 5

2.3. Object Groups

An Object MUST belong to a group. Groups are composition of objects and the objects within the group carry the necessary dependecy information needed to process the objects in the group. Objects that carry information required to resolve dependencies are marked appropriately in their headers. In cases where such information MAY NOT be available, the first object in the group MUST have all the

dependency information needed to processs therest of the objects.

A group shall provide following utilities

* A way for subscribers to specifiy the appropriate consumption point for enabling joins, rewinds and replay the objects, for certain media usecases.

* A way to specify refresh points within a group, serving as decode points, points of switching between qualties for audio/video media

* Serve as checkpoint for relays to implement appropriate congestion responses.

3. Concepts

3.1. Emission

An Emission represents a collection of tracks sourced by an emission source (Emitter or Catalog Maker) and owned by an application Provider. An Emitter MUST be authorized to publish objects of the tracks in a Emission. An Emitter can have one or more emissions.

Few example of Emissions include,

* Collection of audio and video tracks that makes up a broadcast for a live stream by OBS client, the Emitter, to the provider, say Twitch.

* Tracks from different participants (emitters) in a interactive video conference

3.2. Catalog

Catalog is a MOQ Object scoped to a MoQ Session Section 3.3 that provides information about tracks from one of more Emissions and is used by the subscribers for consuming tracks and for publishers to advertise the tracks. The content of "Catalog" is opaque to the Relays and may be end to end encrypted in certain scenarios.

3.3. MoQ Session

A MoQ Session is a top level container under an application Provider that represents one or more emissions, optionally a set of participating relays, and set of publisher publishing content and subscribers that are interested in consuming the content being published.

4. Protocol Design

Media delivery is started by the publisher/subscriber setting up a "Control Stream" for one or more Tracks. The control stream, which

is based on QUIC stream, is used to configure and setup properties for the "Data Stream". Track media objects is delivered over one or more "Data Streams" which can be unidirectional QUIC streams or over QUIC Datagrams. The Control Channel can also be used to configure in-session parameters.

4.1. Control Stream and Messages

The client starts by opening a new bilateral stream, acting as the "control stream" for the exchange of data, carrying a series of control messages in both directions.

The control stream is created for one or more tracks to be published or subscribed to and will remain open as long as the peers are still sending or receiving the media. If either peer closes the control stream, the other peer will close its end of the stream and discard the state associated with the media transfer.

Streams are "one way". If a peer both sends and receive media, there will be different control streams for sending and receiving.

The control channel carry series of messages, encoded as a length followed by a message value:

```
message {
    length(16),
    value(...)
}
```

The length is encoded as a 16 bit number in big endian network order.

Below sub-sections define various control messages defined in this specification.

4.1.1. Subscribe Message

Entities that intend to receive media will do so via subscriptions to one or more Tracks.

```
enum subscribe_intent
{
  immediate(0),
  catch_up(1),
  wait_up(2),
}

track_info {
  track_id_length(i),
  track_id(...)...,
  subscribe_intent intent
}
```

```
subscribe_message {
  message_type(i),
  tracks_length(i),
  track_info tracks (...),
}
```

The message type will be set to SUBSCRIBE (1). tracks identifies the list of tracks as defined by the track_info type.

The track_id captures the Track ID and the intent field specifies the intended consumption point.

Following options are defined for the intent

* immediate: Deliver any new objects it receives for a given track.

* catch_up: Deliver any new objects it receives and in addition send any previous objects it has received, beginning from the most recent group and matching the given track.

* wait_up: Wait until next group before delivering the objects.

Subscriptions are typically long-lived transactions and they stay active until one of the following happens

* a client local policy dictates expiration of a subscription.

* optionally, a server policy dictates subscription expiration.

* the underlying transport is disconnected.

The subscribe message is sent over the associated control stream and the same is closed when the subscriptions for the tracks included are no longer required. This implies the termination of all associated data streams.

4.1.1.1. Aggregating Subscriptions

Subscriptions are aggregated at entities that perform Relay Function. Aggregating subscriptions helps reduce the number of subscriptions for a given track in transit and also enables efficient distribution of published media with minimal copies between the client and other relays, as well as reduce the latencies when there are multiple subscribers for a given track behind a Relay or the provider.

4.1.2. SUBSCRIBE_REPLY Message

A subscribe_reply provides results of the subscription and is sent on the control stream over which the subscribe control message was received.

enum response

```
{
  ok(0),
  expired(1),
  fail(2)
}

track_response {
  Response response,
  track_id_length(i),
  track_id(...)...,
  [ Reason Phrase Length (i) ],
  [ Reason Phrase (...) ],
  [ media_id(i) ]
}

subscribe_reply
{
  message_type(i),
  track_response tracks(...)
}
```

The message type will be set to SUBSCRIBE_REPLY (2). tracks capture the result of subscription per track included in the subscribe message.

For each track, the track_response provides result of subscription in the response field, where a response of ok indicates successful subscription. For failed or expired responses, the "Reason Phrase" shall be populated with appropriate reason code.

The media_id for a given track is populated for a successful subscription and represents an handle to the subscription to be provided by the peer over the data streams(s). Given that the media corresponding to a track can potentially arrive over multiple data streams, the media_id provides the necessary mapping between the control stream and the corresponding data streams. It also serves as compression identifier for containing the size of object headers instead of carrying complete track identifier information in every object message.

While the subscription is active for a given name, the Relay(s) must send objects for tracks it receives to all the matching subscribers. Optionally, a client can refresh its subscriptions at any point by sending a new subscribe_message.

4.1.3. PUBLISH REQUEST Message

The publish_request message provides one or more tracks that the publisher intends to publish data.

```
track_info {
```

```
  track_id_length(i),
  track_id(...)...,
  media_id(i),
}

publish_request {
  message_type(i),
  track_info tracks(...),
}
```

The message type will be set to PUBLISH_REQUEST (3). tracks identifies the list of tracks. The media_id represents an handle to the track to be used over the data streams(s). Given that media corresponding to the track can potentially be sent over multiple data streams, the media_id provides the necessary mapping between the control stream and the associated data streams. media_id also serves as compression identifier for containing the size of object headers instead of carrying full formed Track Id in every object.

The publish_request message is sent on its own control stream and akin to subscribes, the control stream's lifecycle bounds the media transfer state. Terminating the control stream implies closing of all the associated data streams for the tracks included in the request.

4.1.4. PUBLISH_REPLY Message.

publish_reply provides the result of request to publish on the track(s) in the publish_request. The publish_reply control message is sent over the same control stream the request was received on.

```
publish_reply {
  message_type(i),
  track_response tracks(...),
}
```

The message_type is set to PUBLISH_REPLY (4).

tracks capture the result of publish request per track included in the publish_request message. The semantics of track_response is same as defined in Section 4.1.2 except the media_id is optionally populated in the case where the media_id in the request cannot be used.

4.1.5. RELAY_REDIRECT MESSAGE

relay_redirect control message provides an explicit signal to indicate relay failover scenarios. This message is sent on all the control streams that would be impacted by reduced operations of the Relay.

relay_redirect

```
{
  message_type(i),
  relay_address_length(i),
  relay_address(...)...
}
```

The message_type is set to RELAY_REDIRECT (5). relay_address identifies the address of the relay to setup the new subscriptions or publishes to.

4.1.6. Catalog Message

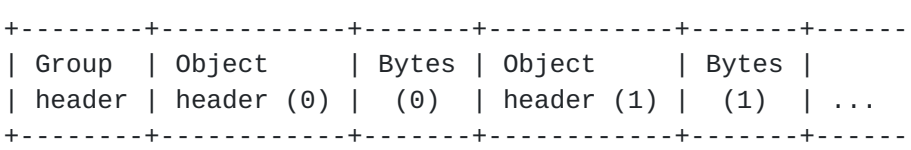
Catalog message provides information on tracks for a given MoQ Session.

```
catalog {
  message_type(i),
  catalog_length(i),
  data(...)...
}
```

The message_type is set to CATALOG (6). data is container specific encoding of catalog information.

4.2. Stream Considerations

Certain applications can choose to send each group in their own unidirectional QUIC stream. In such cases, stream will start with a "group header" message specifying the media ID and the group ID, followed for each object in the group by an "object header" specifying the object ID and the object length and then the content of the objects (as depicted below)



```
+--------+------------+-------+------------+-------+------
| Group  | Object     | Bytes | Object     | Bytes |
| header | header (0) |  (0)  | header (1) |  (1)  | ...
+--------+------------+-------+------------+-------+------
```

The first object in the stream is object number 0, followed by 1, etc. Arrival of objects out of order will be treated as a protocol error.

TODO: this strict "in order" arrival is not verified if there is one stream per drop-priority level. Add text to enable that.

Alternatively, certain applications can choose to send each object in its own unidirectional QUIC stream. In such cases, each stream will start with a "group header" message specifying the media ID and the group ID, followed by a single "object header" and then the content of the objects (as depicted below).


```
+--------+------------+-------+
```

```
| Group  | Object     | Bytes |
| header | header (n) |  (n)  |
+--------+------------+-------+
```

The MOQTransport doesn't enforce a rule to follow for the applications, but instead aims to provide tools for the applications to make the choices appropriate for their use-cases.

4.2.1. Group Header

The group header message is encoded as:

```
group_header {
  message_type(i),
  media_id(i),
  group_id(i)
}
```

The message type is set to GROUP_HEADER, 11. media_id MUST correspond to the one that was setup as part of publish_request control message exchange Section 4.1.3. group_id always starts at 0 and increases sequentially at the original media publisher.

4.2.2. Object header

Each object in the stream is encoded as an Object header, followed by the content of the object. The Object header is encoded as:

```
object_header {
  message_type(i),
  object_id(i),
  [nb_objects_previous_group(i),]
  flags[8],
  object_length(i)
}
```

The message type is set to OBJECT_HEADER, 12. object_id is identified by a sequentially increasing integer, starting at 0.

The nb_objects_previous_group is present if and only if this is the first fragment of the first object in a group, i.e., object_id and offset are both zero. The number indicates how many objects were sent in the previous group. It enables the receiver to check whether all these objects have been received.

The flags field is used to maintain low latency by selectively dropping objects in case of congestion. The flags field is encoded as:

```
{
    maybe_dropped(1),
    drop_priority(7)
```

```
}
```

4.3. Datagram considerations

MoQ objects can be transmitted as QUIC datagrams, if the datagram transmission option has been validated during the subscribe or publish transaction. Such a option is chosen for non-relaible media delivery scenarios.

When sent as datagrams, the object is split into a set of fragments. Each fragment is sent as a separate datagram. The fragment header contains enough information to enable reassembly. If the complete set of fragments is not received in a reasonable time, the whole object shall be considered lost.

4.3.1. Fragment Message

In the datagram variants, instead of sending a series of whole objects on a stream, objects are sent as series of fragments, using the Fragment message:

```
fragment {
  message_type(i),
  [ media_id(i) ],
  [ group_id(i) ],
  [ object_id(i) ],
  fragment_offset(i),
  object_length(i),
  fragment_length(i),
  data(...)
}
```

The message type will be set to FRAGMENT (13). The optional fields media_id, group_id and object_id are provided in the cases where they cannot be obtained from the context where the fragment message is published. For typical cases, the group_header and the object_header messages preceed the series of fragment messages and thus provide the necessary context to tie the data to the object.

The fragment_offset value indicates where the fragment data starts in the object designated by group_id and object_id. Successive messages are sent in order, which means one of the following three conditions must be verified:

* The group id and object id match the group id and object id of the previous fragment, the previous fragment is not a last fragment, and the offset matches the previous offset plus the previous length.

* The group id matches the group id of the previous message, the object id is equal to the object id of the previous fragment plus 1, the offset is 0, and the previous message is a last fragment.

* The group id matches the group id of the previous message plus 1, the object id is 0, the offset is 0, and the previous message is a last fragment.

The nb_objects_previous_group is present if and only if this is the first fragment of the first object in a group, i.e., object_id and offset are both zero. The number indicates how many objects were sent in the previous groups. It enables the receiver to check whether all these objects have been received.

The flags field is used to maintain low latency by selectively dropping objects in case of congestion. The value must be the same for all fragments belonging to the same object.

The flags field is encoded as:

```
{
    maybe_dropped(1),
    drop_priority(7)
}
```

The high order bit maybe_dropped indicates whether the object can be dropped. The drop_priority allows nodes to selectively drop objects. Objects with the highest priority as dropped first.

When an object is dropped, the relays will send a placeholder, i.e., a single fragment message in which:

* offset_and_fin indicates offset=0 and fin=1

* the length is set to zero

* the flags field is set to the all-one version 0xff.

Sending a placeholder allows node to differentiate between a temporary packet loss, which will be soon corrected, and a deliberate object drop.

5. Drop Priority

In case of congestion, the MoQ nodes may have to drop some traffic in order to avoid building large queues. The drop algorithm must respect the relative importance of objects within a track, as well as the relative importance of tracks within an MoQ connection. Relays base their decisions on two properties of objects:

* a "droppable" flag, which indicates whether the application would rather see the object queued (droppable=False) or dropped (droppable=True) in case of congestion.

* a "drop-priority" value, which indicates the relative priority of

this object versus other objects in the track or other tracks in the connection.

Higher values of the drop-priority field indicate higher drop priorities: objects mark with priority 0 would be the last to be dropped, objects marked with priority 3 would be dropped before dropping objects with priority 2, etc. Nodes support up to 8 drop-priority levels, numbered 0 to 7.

Nodes may use drop-priorities in two ways: either by delegating to the QUIC stack, or by monitoring the state of congestion and performing their own scheduling.

5.1. Applying drop-priorities through the QUIC stack

Many QUIC stack allow application to associate a priority with a stream. The MoQ transports can use that feature to delegate priority enforcement to the QUIC stack. The actual delegation depends on the transport choice.

If the MoQ transport uses the strategy where each object is transmitted on a separate unidirectional QUIC stream, then that stream should be marked with the object's priority. The QUIC API should be set to request FIFO ordering of streams within a priority layer.

If all the objects of a given group, say GOP, within a track are sent in a single unidirectional QUIC stream. This strategy can be modified to be priority aware. In a priority aware strategy, there will be one unidirectional stream per group and per priority level, and the priority of the unidirectional stream will match that level.

In both cases, if congestion happens, objects marked as "droppable" will have be dropped by resetting the corresponding unidirectional streams. This decision will happen separately for each track, typically at the end of a group. At that point, the decision depends on whether the content of the unidirectional streams have been sent or not:

* if all objects have been sent, the stream can be closed normally.

* if some objects have not been sent, or not acknowledged, the stream shall be reset, causing the corresponding objects to be dropped.

These policies will normally ensure that for any congestion state, only the most urgent objects are sent.

5.2. Applying drop-priorities through active scheduling

Some transport strategies prevent delegation of priority enforcement to the QUIC stack. For example, if the policy is to use a single

QUIC stream or a single stream carrying objects of different priorities. In such cases, nodes react to congestion by scheduling some objects for transmission and explicitly dropping other objects.

Node should schedule objects as follow:

* if congestion is noticed, the node will delay or drop first the numerically higher priority level. The node will drop all objects marked at that priority, from the first dropped object to the end of the group.

* if congestion persists despite dropping or delaying the "bottom" level, the node will start dropping the next level, and continue doing so until the end of the group.

* if congestion eases, the node will increase the delay or drop level.

While the "drop level" is computed per connection, specific actions will have to be performed at the "track" level:

* for a given track, the node remembers the highest priority level for which objects were dropped in the current group. That level will be maintained for that track until the end of the group.

* at the beginning of a group, the priority level is set to the currently computed value for the connection.

5.3. Tracking drops

For management purposes, it is important to indicate which objects have been dropped, as in "there was supposed to be here an object number X or priority P but it has been dropped." In the scheduling approach, this can be achieved by inserting a small placeholder for the missing object. In the delegating approach, we need another solution. One possibility would be to send a "previous group summary" at the beginning of each group, stating the expected content of the previous group.

5.4. Marking objects with priorities

The publishers mark objects with sequence numbers within groups and with drop and priority values according to the need of the application. This marks must be consistent with the encoding requirements, making sure that:

* objects can only have encoding dependencies on other objects in the same group,

* objects can only have encoding dependencies on other objects with equal or or numerically lower priority levels.

With these constraints, applications have broad latitude to pick priorities in order to match the desired user experience. When using scalable video codecs, this could mean for example chosing between "frame rate first" or "definition first" priorities, or some compromise.

6. Relays

The Relays play an important role for enabling low latency media delivery within the MoQ architecture. This specification allows for a delivery protocol based on a publish/subscribe metaphor where some endpoints, called publishers, publish media objects and some endpoints, called subscribers, consume those media objects. Some relays can leverage this publish/subscribe metaphor to form an overlay delivery network similar/in-parallel to what CDN provides today. While this type of overlay is expected to be a major application of relays, other types of relays can also be defined to offer various types of services.

Objects are received by "subscribing" to it. Objects are identified such that it is unique for the relay/delivery network.

Relays provide several benefits including

* Scalability - Relays provide the fan-out necessary to scale up streams to production levels (millions) of concurrent subscribers.

* Reliability - relays can improve the overall reliability of the delivery system by providing alternate paths for routing content.

* Performance - Relays are usually positioned as close to the edge of a network as possible and are well-connected to each other and to the Origin via high capacity managed networks. This topography minimizes the RTT over the unmanaged last mile to the end-user, improving the latency and throughput compared to the client connecting directly to the origin.'

* Security - Relays act to shield the origin from DDOS and other malicious attacks.

6.1. Relay - Subscriber Interactions

Subscribers interact with the "Relays" by sending a "subscribe" Section 4.1.1 command for the tracks of interest.

Relays MUST be willing to act on behalf of the subscriptions before they can forward the media, which implies that the subscriptions MUST to be authorized and it is done as follows:

1. Provider serving the tracks MUST be authorized. Track IDs provide the necessary information to identify the Origin/ Provider.

2. Subscriptions MUST be authorized. This is typically done by either subscriptions carrying enough authorization information or subscriptions being forwarded to the Origin for obtaining authorization. The mechanics of either of these approaches are out of scope for this specification.

In all the scenarios, the end-point client making the subscribe request is notified of the result of the subscription.

6.2. Relay - Publisher Interactions

Publishers MAY be configured to publish the objects to a relays based on the application configuration and topology. Publishing set of tracks through the relay starts with a "publish_request" transaction that describes the track identifiers. That transaction will have to be authorized by the Origin, using mechanisms similar to authorizing subscriptions.

As specified with subscriber interactions, Relays MUST be authorized to serve the provider and the publish_request MUST be authorized before the Relays are willing to forward the published data for the tracks.

Relays makes use of priority order and other metadata properties from the published objects to make forward or drop decisions when reacting to congestion as indicated by the underlying QUIC stack. The same can be used to make caching decisions.

6.3. Relay Discovery and Failover

Relays are discovered via application defined ways that are out of scope of this document. A Relay that wants to shutdown can send a message to the client with the address of new relay. Client moves to the new relay with all of its Subscriptions and then Client unsubscribes from old relay and closes connection to it.

6.4. Restoring connections through relays

The transmission of a track can be interrupted by various events, such as loss of connectivity between subscriber and relay. Once connectivity is restored, the subscriber will want to resume reception, ideally with as few visible gaps in the transmission as possible, and certainly without having to "replay" media that was already presented.

There is no guarantee that the restored connectivity will have the same characteristics as the previous instance. The throughput might be lower, forcing the subscriber to select a media track with lower definition. The network addresses might be different, with the subscriber connecting to a different relay.

6.5. Examples

Let’s consider the example as show in the picture below, where a large number of subscribers are interested in media streams from the publisher Alice. In this scenario, the publisher Alice has a live broadcast on channel8 with video streams at 3 different quality (HD, 4K and SD)

More specifically,

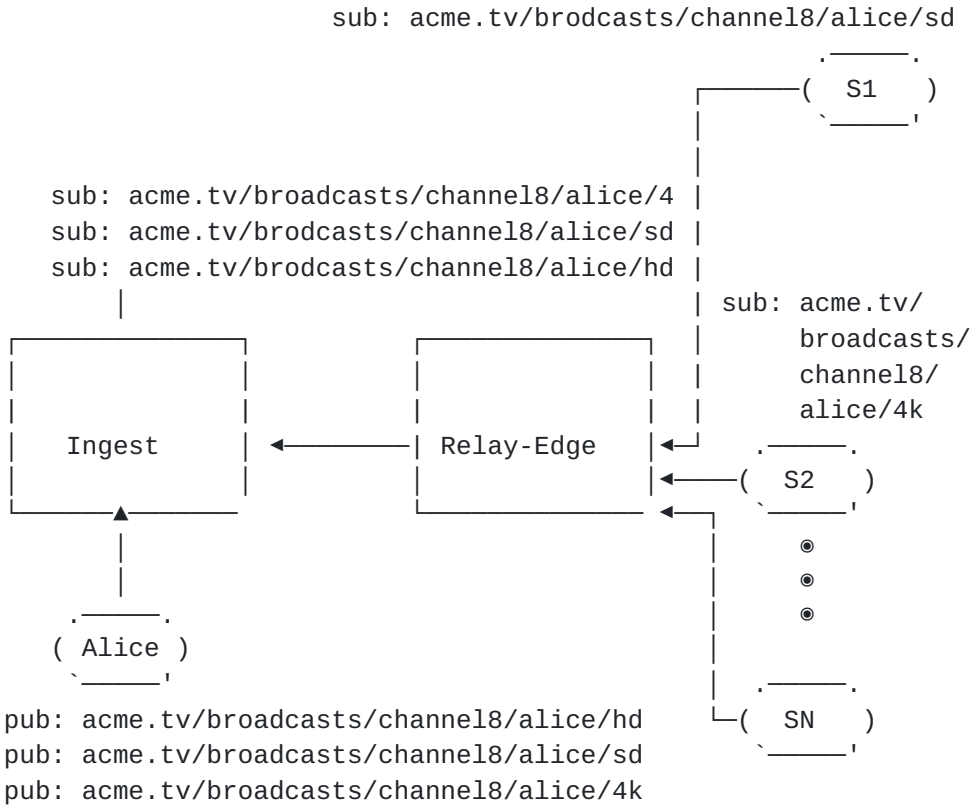
1. Subscriber - S1 is interested in the just the low quality version of the media, and asks for the all the media groups/objects under the specific representation for "sd" quality.

2. Subscriber - S2 is fine with receiving highest quality video streams published by Alice, hence asks for the all the media objects under these representations 4k.

3. Rest of the Subscribers (say Sn,...) are fine with getting just the Hi-def and low quality streams of the video from Alice, and asks for the representations "sd" and "hd" qualities.

The Relay must forward all these subscription requests to the ingest server in order to receive the content.

Note: The notation for identifying the resources for subscription are for illustration purposes only.

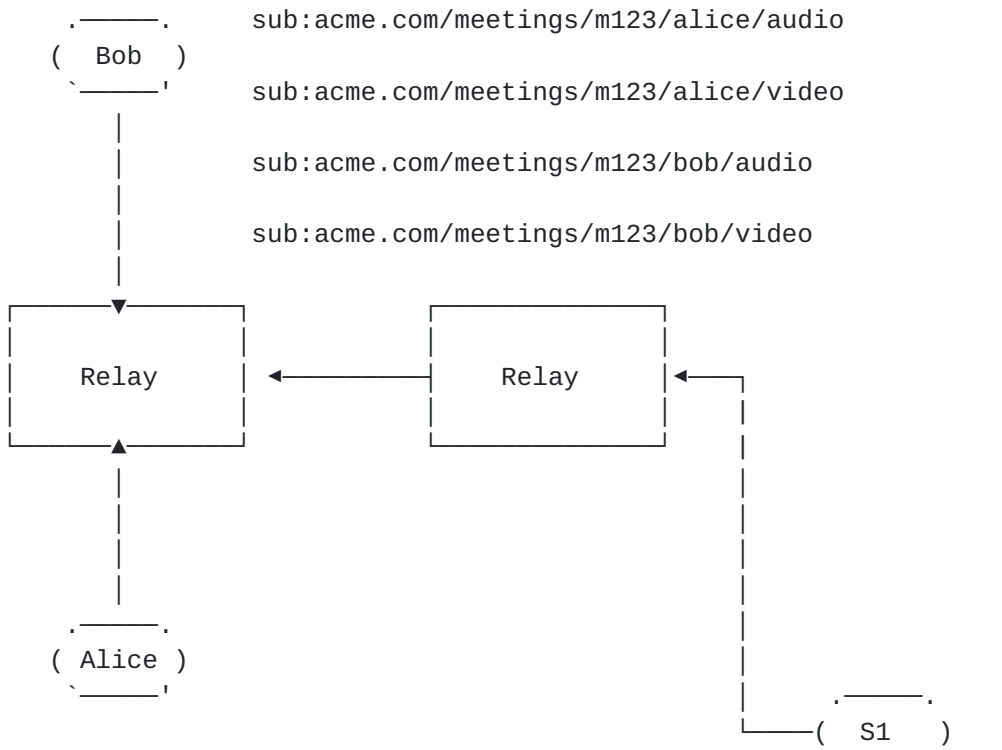


sub: acme.tv/brodcasts/channel8/alice/sd
sub: acme.tv/brodcasts/channel8/alice/hd

The relay does not intercept and parse the CATALOG messages, therefore it does not know the entireity of the content being produced by Alice. It simply aggregates and forwards all subscription requests that it receives.

Similarly, below example shows an Interactive media session

pub: acme.com/meetings/m123/bob/video
pub: acme.com/meetings/m123/bob/audio



pub: acme.com/meetings/m123/alice/video
pub: acme.com/meetings/m123/alice/audio
sub:acme.com/meetings/m123/alice/audio
sub:acme.com/meetings/m123/alice/video
sub:acme.com/meetings/m123/bob/audio
sub:acme.com/meetings/m123/bob/video

The above picture shows as sample media delivery, where a tree topography is formed with multiple relays in the network. The example has 4 participants with Alice and Bob being the publishers and S1 being the subscribers. Both Alice and Bob are capable of publishing audio and video identified by their appropriate names. S1 subscribes to all the streams being published. The edge Relay forwards the unique subscriptions to the downstream Relays as needed, to setup the delivery network.

7. Transport Usages

Following subsections define usages of the MoQTransport over WebTransport and over raw QUIC.

7.1. MoQ over QUIC

MoQ can run directly over QUIC. In that case, the following apply:

* Connection setup corresponds to the establishment of a QUIC connection, in which the ALPN value indicates use of MoQ. For versions implementing this draft, the ALPN value is set to "moq-n00".

* Bilateral and unilateral streams are mapped directly to equivalent QUIC streams

* Datagrams, when used, are mapped directly to QUIC datagram frames.

7.2. MoQ over WebTransport

MoQ can benefit from an infrastructure designed for HTTP3 by running over WebTransport.

WebTransport provides protocol framework that enables clients constrained by the Web security model to communicate with a remote server using a secure multiplexed transport. WebTransport protocol also provides support for unidirectional streams, bidirectional streams and datagrams, all multiplexed within the same HTTP/3 connection.

Clients (publishers and subscribers) setup WebTransport Session via HTTP CONNECT request for the application provided MoQSession and provide the necessary authentication information (in the form of authentication token). The ":protocol" value indicates use of MoQ. For versions implementing this draft, the :protocol value is set to "moq-n00".

In case of any errors, the session is terminated and reported to the application.

Bilateral and unilateral streams are opened and used through the WebTransport APIs.

7.2.1. Catalog Retrieval

On a successful connection setup, subscribers proceed by retrieving the catalog (if not already retrieved), subscribing to the tracks of their interest and consuming the data published as detailed below.

Catalog provides the details of tracks such as Track IDs and corresponding configuration details (audio/video codec detail, gamestate encoding details, for example).

Catalogs are identified as a special track, with its track name as "catalog". Catalog objects are retrieved by subscribing to its TrackID over its own control channel and the TrackID is formed as shown below

Catalog TrackID :=<provider-domain>/<moq-session-id>/catalog

Ex: streaming.com/emission123/catalog

A successfull subscription will lead to one or more catalog objects being published on a single unidirectional data stream. Successfull subscriptions implies authorizaiton for subscribing to the tracks in the catalog.

Unsuccessful subscriptions MUST result in closure of the WebTransport session, followed by reporting the error obtained to the application.

Catalog Objects obtained MUST parse successfully, otherwise MUST be treated as error, thus resulting the closure of the WebTransport session.

7.2.2. Subscribing to Media

Once a catalog is successfully parsed, subscribers proceed to subscribe to the tracks listed in the catalog. Applications can choose to use the same WebTransport session or multiple of them to perform the track subscriptions based on the application requirements.

Also, It is typical for certain applications to group set of tracks in to a single prioritization relationship and transmit them over a single WebTransport Session.

Tracks subscription is done by sending subscribe message as definedin Section 4.1.1

On successful subscription, subscribers should be ready to consume media on one or more Data Streams as identified by their media_ids.

Failure to subscribe MUST result on closure of the control stream associated with the track whose subscription failed and the error MUST be reported to the application.

7.2.3. Publishing Media

On successful setup of the WebTransport session, publishers send publish_request message listing the tracks they intend to publish data on. Publisher MUST be authorized to publish on the tracks and Relays MUST be willing to participate in the media delivery.

A sucessfull publish_reply allows publishers to publish on the tracks

advertised.

Publishing objects on the tracks follow the procedures defined in Section 4.2 and Section 4.3.

8. Normative References

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

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

Appendix A. TODO

1. Add authorization details for the protocol messages.

Appendix B. Security Considerations

This section needs more work

Appendix C. IANA Considerations

TODO: fill this section. Register ALPN. Register WebTransport protocol. Open new registry for MoQ message types. Possibly, open registry for MoQ errors.

Appendix D. References

D.1. Normative References

[RFC XXX] Nandakumar, S "MoQ Base Protocol" Work in progress

D.2. Informative references

[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, %gt;https://www.rfc-editor.org/info/rfc8126>.

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

[WebTransport] Frindell, A., Kinnear, E., and V. Vasiliev, "WebTransport over HTTP/3", Work in Progress, Internet-Draft, draft-ietf-webtrans-http3-04, 24 January 2023,

>https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-http3-04>.

Appendix E. Acknowledgments

Cullen Jennings, the IETF MoQ mailing lists and discussion groups.

Authors' Addresses

Suhas Nandakumar
Cisco
Email: snandaku@cisco.com

Christian Huitema
Private Octopus Inc.
Email: huitema@huitema.net

Will Law
Akamai
Email: wilaw@akamai.com