QuicR - Media Delivery Protocol Over QUIC

Cullen Jennings

Suhas Nandakumar

% December 2021

This specification outlines the design for a media delivery protocol over QUIC. It is based on a publish/subscribe metaphor where entities publish and subscribe to data that is sent to, and received from, relays in the cloud. The information subscribed to is named such that this forms an overlay information centric network.

Introduction

This outlines the design for QuicR, a media delivery protocol for applications that needs real time and/or near real-time data delivery experiences. It is based on a publish/subscribe metaphor where client endpoints publish and subscribe to data that is sent to, and received from, relays in the cloud. The information subscribed to is named such that this forms an overlay information centric network.

Typical usecases would be a conferencing applications (audio conferencing, webinars) that demand low latencies at larger scale (in terms of participants), where for each endpoint subscribe to the media/data from each of the other participants in the conference and at the same time publish their media/data. The cloud device that receives the subscriptions and distributes data is called a Relay and is similar to an SFU in the audio/video uses cases. QuicR is pronounced something close to "quicker" but with more of a pirate "arrrr" at the end.

The QuicR protocol takes care of transmitting named data from the Publisher to the Relay. It supports services (with the support of underlying transport, where necessary) to deal with detecting and limiting the bandwidth available, congestion control, fragmentation and reassembly, and prioritization of data. The maximum lifetime of data can be controlled and important data can be marked for retransmission and so on. It is designed to be NAT and firewall traversal friendly as well as high speed implementations in relays and use with load balancers. QuicR relay forwards the named data that was sent to it to all the subscribers for that data. Data is named such that it is unique for the relay and scoped to a origin. Subscriptions can include a form of wildcarding to the named data.

The design is usable for sending media between a set of participants in a game or video call with under a hundred milliseconds of latency and meets needs of web conferencing systems. The design can also be used for large scale low latency streaming to millions of participants with latency under a few hundred milliseconds. It can also be used as low latency publish/subscribe system for real time in systems such as messaging and IoT.

In the simple case, an web application could have a single server implementing Relay function in the cloud that forwards packets between users in a video conference. The cloud could also have multiple relays. QuicR is designed to make it easy to implement stateless 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. It should allow for design that support high speed forwarding with ASICs, or NIC cards with on card processing, or use intel DPDK. Relay can also be chained so that a relay in a CDN network gets one copy of data from the central cloud data center and can then forward it to many clients that use that CDN. This approach can be extended to put relays very close to clients in 5G networks and put relays in home routers or enterprise branch office that can be automatically discovered. The use of a relay between the WIFI and WAN allows for different strategies for reliability and congestion control to happen on the WAN and WIFI which can improve the overall end to end user experience.

The development of the protocol is based on optimizing the end user experience for the person using applications built with QuicR. It is not optimized to provide the highest average throughput of data.

In some settings the relay can also be directly connected to a backend to inject published data or receive subscribes that trigger other backend logic.

Contributing

All significant discussion of development of this protocol is in the GitHub issue tracker at TODO.

Terminology

- Relay Function: Functionality of the QuicR architecture, that implements store and forward behavior at the minimum. Such a function typically receives subscrptions and publishes data to the other endpoints that have subscribed to the named data.
- Relay: Server component (physical/logical) in the cloud that implements the Relay Function.

- Publisher: An endpoint that sends named data to a Relay. [also referred to as producer of the data]
- Subscriber: An endpoint that subscribes to anmed data and receives data. Relays can act as subscribers to other relays. Subscribers can also be referred to consumers of the data.
- Client/QuicR Client: An endpoint that acts as a Publisher, Subscriber, or both. May also implement a Relay Function in certain contexts.
- Data: Application level chunk of Data that has a unique Name, a limited lifetime, priority and is transported via this protocol.

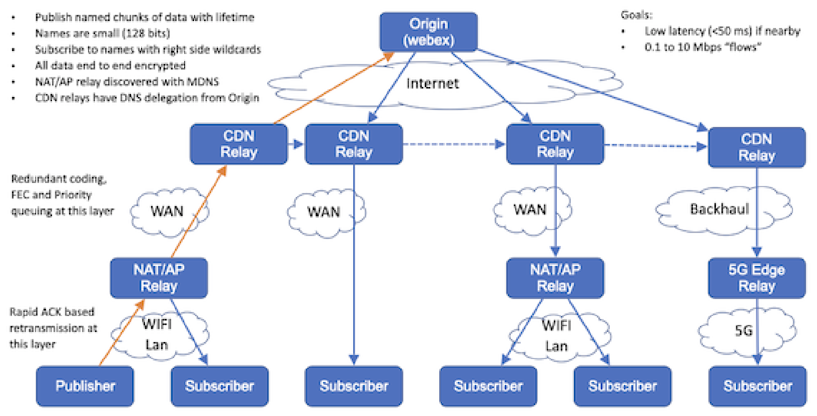
Architecture

Problem Space

This is design for applications such as voice and video system that are deployed in the cloud, games systems, multiuser AR/VR applications, and IoT sensor that produce real time data. It is designed for endpoints with between 0.1 and 10 mbps connection to the Internet that have a need for real time data transports. The main characteristic of real time data is that it is not useful if it is takes longer than some fixed amount of time to deliver.

The client can be behind NATs and firewalls and will often be on a WIFI for cellular network. The Relays need to have a public IP address, or at least an IP address reachable by all the clients they serve, but can be behind firewalls and load balancers.

Components


• Publish named chunks of data with lifetime
• Names are small (128 bits)
• Subscribe to names with right side wildcards
• All data end to end encrypted
• NAT/AP relay discovered with MDNS
• CDN relays have DNS delegation from Origin
Origin (webex)
Goals:
• Low latency (<50 ms) if nearby
• 0.1 to 10 Mbps "flows"
Internet
CDN Relay
CDN Relay
CDN Relay
CDN Relay
Redundant coding, FEC and Priority queuing at this layer
WAN
WAN
WAN
Backhaul
NAT/AP Relay
NAT/AP Relay
5G Edge Relay
Rapid ACK based retransmission at this layer
WIFI Lan
WIFI Lan
5G
Publisher
Subscriber
Subscriber
Subscriber
Subscriber
Subscriber

!—
!—

Above is one way to envision an end to end system architecture for QuicR based deploymets which is formed of the following roles.

[[todo explain the picture]]

Names and Named Data

Names are basic elements with in the QuicR architecture and they uniquely identify objects. Named objects can be cached in relays in a way CDNs cache resources and thus can obtain similar benifits such caching mechanisms would offer.

Names are composed of following components:

1. Domain Component
2. Application Component

Domain component uniquely identifies a given application domain. This is like a HTTP Origin and uniquely identifies the application and a root relay. This is a DNS domain name or IP address combined with a UDP port number mapped to into the domain. Example: sfu.webex.com:5004.

Application component is scoped under a given Domain/Origin. This component is further split into 2 sub-components by a given application. First component represents a static aspect of the application's usage context (meetingId in a conferencing applications) and the final sub-component represent dynamic aspects (stream/encoding time). Such a division would allow for efficient wildcarding rules (see Wildcarding) when supported. The length and interpretation of each application sub-component is application specific. Also to note, such a sub-division is optional and care should be taken when supporting wildcarding rules, if omitted.

Example: In this example, the domain component identifies acme.meeting.com domain, the application compoment identifies an instance of a meeting under this domain, say "meeting123", and one of many meida streams, say camera stream, from the user "alice" `quicr://acme.meeting.com/meeting123/alice/cam5/`

Names within QuicR should adhere to following constraints:

- Names should enable compact representation for efficient transmission and storage.
- Names should be efficiently converted to cache friendly datatypes (like Keys in CDN caches) for storage and lookup purposes.
- Names should enable data lookup at the relays based on partial as well as whole names.

Name Discovery

Names are discovered via manifests. The role of the manifest is to identify the names as well as aspects pertaining to the associated data in a given usage context of the application. The content of Manifest is application defined and end to end encrypted. The manifest is owned by the application's origin server and are accessed as a protected resources by the authorized QuicR clients. The QuicR protocol treats Manifests as first level named object, thus allowing for clients to subscribe for the purposes of bootstrapping into the session as well as to follow manifest changes during a session [new members joining a conference for example].

[todo should the maifest be end to end encrypted ?]

To this extent, the origin Servers MUST support following QuicR name for subscribing to the manifests.

`quicr://domain/<application-static-component>/manifest`

Also to note, a given application might provide non QuicR mechanisms to retrieve the manifest. Such mechanisms are out of scop and can be used complementary to the approaches defined in this specification.

Named Objects

The objects that names point to are application specific. The granularity of such data (say media frame, fragment, datum) and its frequency are fully specified by a given application and they need to be opaque for relays/in-transit caches. The data objects are end-to-end encrypted.

QuicR Protocol

Entities within QuicR architecture communicate using publish and subscirbe protocol messages for setting up media delivery via the intermediaries such as relay and the Origin.

Entities that send data do so by sending Publish message(s) with named data and the entities that are interested in receiving data do so by sending Subscribe messages to the names of interest.

A publisher entity is responbile for producing data (aka named objects) and the subscriber entity subscribes to the data of interest with subscription(s) to the associated name. A combination of Origin and relays share the responsibility of delivering the data corresponding to the names.

[[todo add details on authenticated messages]]

Publish API/Message

Entities the want to send data will use Publish API to trigger `PUBLISH` messages from a QuicR client to a QuicR server (Relay(s) / Origin Server). The publish message identies active flow of named data from the client to the server, such a data can be originating from a given QuicR endpoint client or a might be relayed by other entities. In the latter case, the relaying entitiy MUST NOT change the name associated with the data being published.

[[TODO Add details end to end integrity protected and e2ee protected parts of the message]]

In general, the Publish API specifices following thing about the data being published.

```
NAME           (String)
RELIABLE       (Boolean)
PRIORITY       (Enumeration)
BESTBEFORE     (Number)
TIMESTAMP      (Number)
DISCARDABLE    (Boolean)
IS_SYNC_POINT  (Boolean)
Data[...]      (Bytes)
```

NAME: Every `PUBLISH` message MUST have a name to identify the data to the QuicR components (relays/origin server/other clients). Names specified must be full names and MUST not represent partial names.

DATA: The data to be published is identified by the `Data` field which are timestamped buffers of application data.

RELIABLE: This flag indicates the data to be sent in order and reliably.

BESTBEFORE: Time to live defines the time after which the data can be discarded from the cache.

PRIORITY: Enumeration specifying relative priority of data being published by this end-point. This can help Relay to make dropping/caching decisions.

DISCARDABLE: Provides an hint to the relays for making drop decisions.

IS_SYNC_POINT: Synchronization point acts as reference point for the named data and MUST be cached by the relays. Such data enables quick synchronizations for the subscribers.

The `PUBLISH` message(s) are represented as below and are embedded within a underlying transport packet.

`A> All the integer fields are variable length encoded`

```
PUBLISH {
  NAME              (Number128)
  FLAGS             (Byte)
  FRAGMENT_ID       (Number16)
  BESTBEFORE        (Number64)
  TIMESTAMP         (Number64)
  Data[...]         (ByteArray)
}
```

```
Flags := Reserved (3) | IsDiscardable (1) | Is_Sync_Point(1) | Priority (3)
```

Subscribe API/Message

Entities that intend to receive data will do so via subscriptions to named data. The Subscribe API triggers sending `SUBSCRIBE` messages. Subscriptions are sent from the QuicR clients to the origin server(s) (via relays, if present) and are typically processed by the relays. See {#relay_behavior} for further details. All the subsriptions MUST be authenticated and authorized.

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

- a client local policy dictates expiration of a subscription.
- optionally, a server policy dicates subscription expiration.
- the underlying transport is disconnected.

When an explicit indication is preferred to indicate the expiry of subscription, it is indicated via `SUBSCRIPTION_EXPIRY` message.

While the subscription is active for a given name, the QuicR server should be able to send data it receives for the matched name to all the subscribers. A QuicR client can renew its subscrptions at any point by sending a new `SUBSCRIBE` message to the origin server. Such subscriptions MUST refresh the existing subscriptions for that name.

```
SUBSCRIBE {
  SUBSCRIPTION_ID    (Number64)
  NAMES              [Number128..]
}

NAMES := array of names for subscriptons.
```

Aggregating Subscriptions

Subscriptions are aggregated at entities that perform Relay Function. Aggregating subscriptions helps reduce the number of subscriptions for a given named data in transit and also enables efficient disrtibution of published media with minimal copies between the client and the origin server , as well as reduce the latencies when there are multiple subscribers for a given named-data behind a given cloud server.

Wildcarded Names

The name used in `SUBSCRIBE` can be truncated by skipping the right most segments of the name that is application specific, in which case it will act as a wildcard subscription to all names that match the provided part of the name. For example, in an web conferencing use case, the client may subscribe to just the origin and ResourceID to get all the media for a particular conference. Subscription message get a success or error response.

PUBLISH_INTENT/PUBLISH_INTENT_OK Message

[[todo: re-evaluate these messages - do we need this in the protocol or out of band]]

The `PUBLISH_INTENT` message indicates the names chosen by a Publisher for transmitting data within a session. This message is sent to the Origin Server whenever a given publisher intends to publish on a new name (which can be at the beginning of the session or during mid session). This message is authorized at the Server and thus requires a mechanism to setup the initial trust (via out of band) between the publisher and the origin server.

A `PUBLISH_INTENT` message is represented as below:

```
PUBLISH_INTENT {
 PUBLISHER_ID   (Number64]
 NAMES          [Number64Array]
 ORIGIN         [String]
}
```

The ORIGIN field is used by the cloud Relays to choose the Origin server to forward the PUBLISH_INTENT message.

On a successful validation at the Origin server, a PUBLISH_INTENT_OK message is returned by the Origin server.

```
PUBLISH_INTENT_OK {
 PUBLISHER_ID   (Number64)
 NAMES          [Number64...]
}
```

This message enables cloud relays to know the authorized names from a given Publisher. This helps to make caching decisions, deal with collisions and so on.

```
A>A cloud relay could start caching the data associated with the
names that has not been validated yet by the origin server and
decide to flush its cache if no PUBLISH_INTENT_OK is received
within a given time. This is an optimization that would allow
publishers to start transmitting the data without needing to wait
a RTT.
```

[[todo add a note on allowing these messages to piggybacked with other messages to avoid RTT mid session when there is an intent to publish new names.]]

```
A> Names chosen by the publishers MUST be unique with in a given
session to avoid collisions. It is upto the application define
the necessary rules to ensure the uniqueness constraint. Cloud
entities like Relays are agnostic to these rules and handle
collisions by either overriding or dropping the  associated data.
```

SUBSCRIBE_REPLY Message

A SUBSCRIBE_REPLY provides result of the subsciptions. It lists the names that were successfully in subscrptions and ones that failed to do so.

```
SUBSCRIBE_REPLY
{
    SUBSCRIPTION_ID     (Number64)
    NAMES_SUCCESS       [Number128..]
    NAMES_FAIL          [Numbwe128..]
}
```

SUBSCRIBE_CANCEL Message

A `SUBSCRIBE_CANCEL` message indicates a given subscription is no longer valid. This message is an optional message and is sent to indicate the peer to discontinued interest in a given named data.

```
SUBSCRIBE_CANCEL
{
    SUBSCRIPTION_ID (Number64)
    NAMES             [Number128..]
    Reason       (Optional String)
}
```

SYNC_CATHCUP Message

A `SYNC_CATHCUP` message is sent from a QuicR client to the server requesting to refresh to the latest state of a given named data. This is used to allow for clients to perform near realtime catchup or quick sync flows.

```
SYNC_CATHCUP
{

    NAME      (Number128)
    AFTERTIME (Optional Number64]
}
```

Fragmentation and Reassembly

Application data may need to be fragmented to fit the underlying transport packet size requirements. QuicR protocol is responsbile for performing necessary fragmentation and reassembly. Each fragment needs to be small enough to send in a single transport packet. The low order bit is also a Last Fragment Flag to know the number of Fragments. The upper bits are used as a fragment counter with the frist fragment starting at 1.

```
A>QuicR doesn't support delivery of the partial data to the
application.
```

Relay Function and Relays

Clients may be configured to connect to a local relay which then does a Publish/Subscribe for the appropriate named data towards the origin or towards another Relay. These relays can aggregate the subscriptions of multiple clients. This allows a relay in the LAN to aggregate request from multiple clients in subscription to the same data such that only one copy of the data flows across the WAN. In the case where there is only one client, this may still provides benefit in that a client that is experiencing loss on WIFI WAN has a very short

RTT to the local relay so can recover the lost data much faster, and with less impact on end user QoE, than having to go across the LAN to recover the data.

Relays can also be deployed in classic CDN cache style for large scale streaming applications yet still provide much lower latency than traditional CDNs using Dash or HLS. Moving these relays into the 5G network close to clients may provide additional increase in QoE.

At a high level, Relay Function within QuicR architecture support store and forward behavior. Relay function can be realized in any component of the QuicR architecture depending on the application. Typical use-cases might require the intermediate servers (caches) and the origin server to implement the relay function. However the endpoint themselves can implement the Relay function in a Isomorphic deployment, if needed.

For the data being published, the relay function is responsible for * Forwarding the data (even when fragmented) as they arrive to the subscribers * Store full assembled named data for X seconds worth or Y count worth in the cache.

Non normatively, the Relay function is responsible carryout the following actions to enable the QuicR protocol:

1. On reception of `SUBSCRIBE` message, forward the message to the Origin server, and on the receipt of `SUBSCRIBE_REPLY`, store the subscriber info against the names in the NAMES_SUCCESS field of the `SUBSCRIBE` message. If an entry for the name exists already, add the new subscriber to the list of Subscibers. [See Subscribe Aggregations].

2. If there exists a matching named data for a subscription in the cache, forward the data to the subscriber(s). Since this is a new subscriber, forward the data object corresponding to the SYNC_POINT.

3. Optionally, On reception of `PUBLISH_INTENT` message, forward the message to the Origin server, and on the receipt of `PUBLISH_INTENT_OK`, store the names as authorized against a given publisher.

4. If a named data arrives at the relay via `PUBLISH` message , cache the name and the associated data, also distribute the data to all the active subscribers, if any, matching the given name.

The data associated with a given `PUBLISH` message MUST not be cached longer than the **BESTBEFORE** time specified. Also to note, the local policies dicatated by the caching service provider can always overwrite the caching duration for the published data.

Relays MUST NOT modify the either the `Name` or the contents of `PUBLISH/SUBSCRIBE` messags expect for performing the necessary forwarding and caching operations as described above.

For published data marked with `is_sync_point`, the assembled full data MUST be stored along with the timestamp provided in the `PUBLISH` message. This will enable fast sync for newly subscribing clients, for example. Also store most recent X fully assembled named data objets (X may be 3-5) to allow `SYNC_CATCHUP` flows.

Relay fail over

A relay that wants to shutdown and use the redirect message to move traffic to a new relay If a relay has failed and restarted or been load balanced to a different relay, the client will need to resubscribe to the new relay after setting up the connection.

[[todo Cluster so high reliable relays should share subscription info and publication to minimize of loss of data during a full over.]]

Relay Discovery

Local relays can be discovered via MDNS query to TODO. A Relay can send a message to client with the address of new relay. Client moves to the new relay with all of its Subscription and then Client unsubscribes from old relay and closes connection to it.

This allows for make before break transfer from one relay to another so that no data is lost during transition. One of the uses of this is upgrade of the Relay software during operation.

Implications of Fragmentation and Reassembly

Relay function MUST cache full named-data objects items post assembling the fragmented procedures. The choice of such caching is influence by attributes on the named object - discardable or is_sync_point, for example. A given Relay implementation MAY also stored a few of the most recent full named objects regardless of the attributes to support quick sync up to the new subscribers or to support fast catchup functionalities.

When performing the relay function (forwarding), following 2 steps needs to be carried out:

1. The fragments are relayed to the subscriber as they arrive
2. The fully assembled fragments are stored based on attrbutes associated with the data and cache local policies.

It is upto the applications to define the right sized fragments as it can influence the latency of the delivery.

Catchup/Syncup

Certain applications need to be able catchup to data. This can happen when subscribers loose their connection temporarily or for reasons where their client is behind the live edge and hence needs to catchup.

Entities supporting Relay Function may decide to support such functionality by supporting `SYNC_CATHCUP` protocol message. On reception of this message, if **AFTERTIME** is absent, forward the data object corresponding to the SYNC_POINT, otherwise, check the cache to see if there exists any data objects after the **AFTERTIME** time point and forward the same.

Origin Server

The Origin server within the QuicR architecture performs the following logical roles

- NamedDataIndex Server : NameDataIndex is an authorized server for a given Origin and can be a log- ical component of the Origin server. This component enables discovery and distribution of names within the QuicR architecture. Names and the associated application specific metadata are distributed via containers called Manifests. See {#Naming} for furhter detials on names and manifests.
- Relay Function - See Section on Relats
- Application specific functionality

QUIC Mapping

Streams vs Datagrams

Publishers of the named-data can specify the reliability gaurantees that is expected from the QUIC transport. Setting of `IS_RELIABLE` flag to true enables sendjng the application data as QUIC streams, otherwise as QUIC Datagrams.

`SUBSCRIBE` for manifest always happens over QUIC Stream. Each new `SUBSCRIBE` will be sent on a new QUIC Stream or as QUIC DATAGRAMs based on the setting of `IS_RELIABLE` flag.

`PUBLISH` messages per name are sent over their own QUIC Stream or as QUIC DATAGRAM based on ‘`IS_RELILABLE` setting.

Congestion Control

Based on the application profile in use, the transport needs to be able to choose the appropriate for detecting and adapting to the network congestion. A realtime application is more sensitive to congestion and the underlying mechanism needs to quickly adapt compared to, say, an application that is playing a recorded streaming media for example.

QUIC's congestion control mechanisms needs to be evaluated for efficacy real-time contexts. There needs to be way to negotiate the congestion control algorithm to be used per connection to allow algorithms that support different workloads.

Consider integrating realtime friendly congestion control algorithms as part of the negotiation.

Recovery and Error Correction

It is important for the underlying transport to provide necessary error recovery mechanisms like retransmissions and possibly a suitable forward error correction mechanism. This is especially true for packet loss sensitive applications to be resilient against these losses.

QUIC support ACK & Retranmission for QUIC Streams, but just ACKs for QUIC DATAGRAMs. To embrace realtime flows, QUIC's receiver-ts and/or one-way-delay efforts needs to be evaluated within the context of QuicR work.

This work should evaluate support of forward error correction mechanisms for QUIC DATAGRAMs at the minimum to allow realtime flows to be resilient under losses. The same may be exposed to QUIC Streams as well if deemed necessary.

Real-time Conferencing Application

This subsection expands on using QuicR as the media delivery protocol for a real-time multiparty A/V conferencing applications.

Naming

Objects/Data names are formed by concatenation of the domain and application components. Below provides one possible way to subdivide the application component portion of the data names for a conferencing scenario along with their integer bit lengths when encoded as interger shortnames.

- ResourceID: A identifier for the context of a single group session. Is unique withthing scope of Origin. The is a variable length encoded 40 bit integer. Example: conferences number
- SenderID: Identifies a single endpoint client within that ResourceID that publishes data. This is a variable length encoded 30 bit integer. Example: Unique ID for user logged into the origin application.
- SourceID: Identifies a stream of media or content from that Sender. Example: A client that was sending media from a camera, a mic, and screen share might use a different sourceID for each one. A scalable codec with multiple layers or simulcast streams each would use a different sourceID

for each quality representation. This is a variable length encoded 14 bit integer.

- MediaTime: Identifies an immutable chunk of media in a stream. The TAI (International Atomic Time) time in milliseconds after the unix epoch when the last sample of the chunk of media was recorded. When formatted as a string, it should be formatted as 1990-12-31T23-59.601Z with leading zeros. The is a variable length encoded 44 bit integer.Example: For an audio stream, this could be the media from one frame of the codec representing 20 ms of audio.

A conforming name is formatted as URLs like:

`quicr://domain:port/ResourceID/SenderID/SourceID/MediaTime/`

and ShortNames are formed from the cominaton fo the ResrouceID, SenderID, SourceID, MediaTime. Note the Origin is not in the ShortName and is assumed from the context in which the ShortName is used.

Manifest

As a prerequisite step, participants exchange their transmit and receive capabilities like sources, qualities, media types and so on, with application server (can be origin server). This is done out-of-band and is not in the scope of QuicR protocol.

However, as a outcome of this step is generation of the manifest data that describes the names, qualities and other information that aid in carrying out media delivery with QuicR protocol. This would for example setup unique SourceID sub-part of the application component for each media source or quality layers or a combination thereof. Similarly the SenderID may get mapped from a roster equivalent for the meetng. Also to note, for a given meeting, the static sub-part of the application component is set to the ResourceID that represents a identifier for that meeting.

A manifest may get updated several times during a session - either due to capabilities updates from existing participants or new participants joinings or so on.

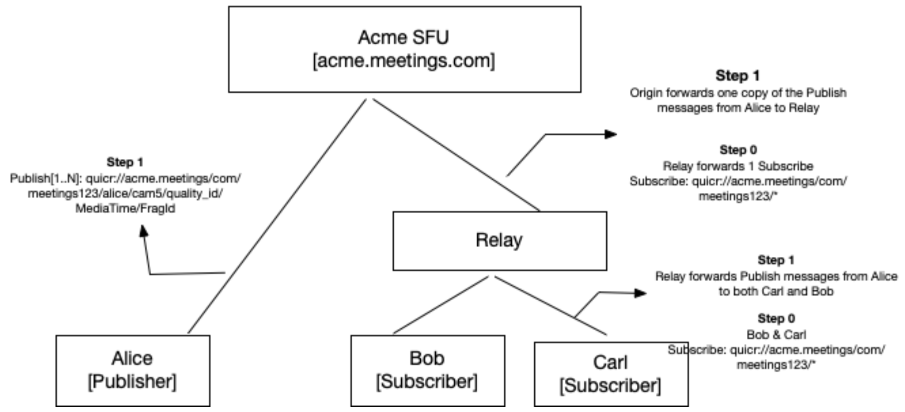
Participants who wish to receive media from a given meeting in a web conference will do so by subscribing to the meeting's manifest. The manifest will list the name of the active publishers.

API Considerations

QuicR client participating in a realtime conference has few options at the API level to choose when published data : * When sending video IDR data, `IS_SYNC_POINT` is set to true. * When sending data for a layer video codec, `IS_RELIABLE` option can be set to true for certain layers. Also the priority levels between the layer may be adjusted to report relative importance. * Selectively retranmissions can be enbaled based on the importance of the data. * [[todo add more flows]]

Example

Below picture depicts a simplified QuicR Publish/Subscribe protocol flow where participants exchange audio in a 3-party realtime audio conference.


Acme SFU
[acme.meetings.com]
Step 1
Origin forwards one copy of the Publish messages from Alice to Relay
Step 0
Relay forwards 1 Subscribe
Subscribe: quicr://acme.meetings/com/meetings123/*
Step 1
Publish[1..N]: quicr://acme.meetings/com/meetings123/alice/cam5/quality_id/MediaTime/FragId
Relay
Step 1
Relay forwards Publish messages from Alice to both Carl and Bob
Step 0
Bob & Carl
Subscribe: quicr://acme.meetings/com/meetings123/*
Alice
[Publisher]
Bob
[Subscriber]
Carl
[Subscriber]

!—
!—

In the depicted protocol flow, Alice is the publisher while Bob and Carl are the subscribers. As part of joining into the conference, Bob and Carl subscribe to the name ___quicr://acme.meetings.com/meeting123/*___ to receive all the media streams being published for the meeting instance **meeting123**. Their subscriptions are sent to Origin Server via a Relay.The Relay aggregates the subscriptions from Bob and Carl forwarding one subscribe message. On Alice publishing her media stream fragments from a camera source to the Origin server, identified via the names ___quicr://acme.meetings.com/meeting123/alice/cam5/t1000/, the same is forwaded to Relay. the relay will in turn forward the same to Alice and Bob based on their subscriptions.

Here is another scenario, where Alice has 2 media sources (mic, camera) and is able to send 3 simulast streams for video and audio encoded via 2 different codecs, might have different sourceIDs as listed below

```
Source       --> SourceID
------------------------
mic_codec_1  --> 1111
mic_codec_2  --> 2222
vid_simul_1  --> 3333
vid_simul_2  --> 4444

Alice's SenderID -> Alice and she is joining meeting with id meeting123
```

```
Names that Alice can publish includes:

quicr://acme.meetings.com/meeting123/Alice/1111/...
quicr://acme.meetings.com/meeting123/Alice/2222/...
quicr://acme.meetings.com/meeting123/Alice/3333/...
quicr://acme.meetings.com/meeting123/Alice/4444/...
```

Manifest encoded as json objects might capture the information as below. [This encoded is for information purposes only.] [[TODO - Do we need a common manifest format ???]]

```
{
    "origin": "acme.meetings.com"
    "meeting": "meeting123",
    publisher {
        id: "Alice",
        source: [
            {
                "type" : "audio",
                "streams": [
                    {
                        "id": "1111",
                        "codec": "opus",
                        "quality": "48khz",
                    }, {
                        "id" : "2222",
                        ....
                    }
                ]
            }, {
            "type": "video",
            "streams": [
                {
                    "id": "3333",
                    "codec:" : "av1",
                    "quality" : "1920x720_60fps"
                },
                {
                    "id": "4444",
                    "codec:" : "av1",
                    "quality" : "640x480_30fps"
                }
            ],
            },
        ]
    }
}
```

With the names as above, any subscriber retrieving the manifest has the necessary information to send `SUBSCRIBE` for the named data of interest. The same happens when the manifest is updated once the session is in progress.

```
A>The details on security/trust relationship established between
the endpoints,the relay and the Origin server is ignored from the
depiction for simplicity purposes.
```

Push To Talk Media Delivery Application

Frontline communications application like Push To Talk have close semblances with the publish/subscribe metaphor. In a typical setup, say a retail store, there are mutiple channels (bakery, garden) and retail workers with PTT communication devices access the chatter over such channels by tuning into them. In a typical use-case, the retails workers might want to tune into one or more channels of their interest and expect the media delivery system to deliver the media asynchornusly as talk spurts.

In general such a system needs the following:

- A way for the end users to tune to their channels of interest and have these interests be longlived transactions.
- A way for system to efficiently distribute the media to all the tuned in end users per channel.
- A way for end user to catch up and playback when switching the channels.

Naming

One can model naming for PTT applications very similar to the design used for "Realtime conferencing applications".

A conforming name is formatted as URLs like:

`quicr://domain:port/ResourceID/SenderID/SourceID/MediaTime/`

In the case of PTT, the following mappings can be considered for the application subcomponents

- ResourceID - Each PTT channel represents its own high level resource
- SenderID - Authenticated user's Id who is actively checked in a given frontline workspace (ex: retail store)
- MediaTime - Same as in the case of "Realtime Conferencing Application"

Manifest

For PTT application, a manifest describes various active PTT channels as the main resource. Subsribers who tune into channels typically get the names from the manifest to do so. Publshers publish their media to channels that they are authorized to.

Example

An example retail store scenario where users Alice, Bob subscribe to the bakery channel and Carl subscribes to the gardening channel. Also an annoucement from the store manager Tom, on bakery channel gets delivered to Alice and Bob but not Carl.

```
Bakery -> Alice and she is authorized to talk/listen on Channel Bakery.
Bob's SenderID -> Bob and he is authorized to talk/listen on Channel Bakery.
Carl's SenderID -> carl and he is authorized to talk/listen on Channel Gardening.
Tom's SenderID -> Tom and he is authorized to talk/listen on channels Bakery and Gardening

Bakery Channel Id -> 1234
Gardening Channel Id -> 5678
```

Manifest encoded as json objects might capture the information as below. [This encoded is for information purposes only.]

```
{
    "origin": "retail19012.sjc.acme.com"
    "channel": [
        {
            "name": "bakery",
            "id" : "1234"
        },
        {
            "name": "gardening",
            "id" : "5678"
        },
    ]
}
```

Alice and Bob shall send `SUBSCRIBE` to channel Bakery and Carl does the same for channel Gardening.

Low Latency Streaming Applications

A typical streaming application can be divided into 2 halves - media ingest and media distribution. Media ingestion is done via pushing one or more streams of different qualities to a central server. Media Distribution downstream is customized (via quality/rate adapation) per consumer by the central server.

One can model ingestion as sending `PUBLISH` mesages and the associated sources as publishers. Similarly, the consumers/end clients of the streaming media `SUBSCRIBE` to the media elements whose names are defined in the manifest. Manifest describes the name and qualities associated with media being published. The central severs (Origin) themselves act as publisher for producing streams with different qualities.

Streaming use-cases requiring lower latencies and high degree of realtime interactivity (chat for example) can fit into QuicR's media delivery protocol over the QUIC transport.

Lower latencies can be achieved by the relay forwarding the data as they arrive to the subscribed clients.

Catch up or quiclk sync can be supported via cache storing fully assembled frames along with the same distribting the fragments as they come in. This will allow clients to get the sync point as well as the data corresponding to the live edge.

Few sample scenarios that have such constrainsts are listed below:

- Professional streamers (gamers/musicians) interacting with a live audience on social media, often via a directly coupled chat function in the viewing app/environment. A high degree of interactivity between the performer and the audience is required to enable engagement.
- Live Auctions are another category of applications where an auction is hroadcasted to serveral participants. The content must be delivered with low latency and more importantly within a well-defined sync across endpoints so customers trust the auction is fair.

Visual and aural quality are secondary in priority in these scenarios to sync and latency. This in turn increases revenue potential from a game/event.

Naming and Manifest Considerations

For downstream distribution of media data to clients with varying requirements, the central server (along with the source) generate different quality media representations. Each such quality is represented with a unique name and subscribers are made know of the same via the Manifest.

```
{
  "resource" : "jon.doe.music.live.tv",
  "streams: [
      {
      "id": "1234",
      "codec": "av1",
      "quality": "1920x720_60fps"
      },
      {
      "id": "5678",
      "codec": "av1",
      "quality": "3840x2160_30fps"
      },
      {
```

```
      "id": "9999",
      "codec": "av1",
      "quality": "640x480_30fps"
      },
  ]
}
```

Consumers end points subscribe to one or more names representing the quality based on their capabilities. This enables the relay to forward the ingested data to be sent as they arrive to the subscribers.

In the scenarios, where the client is trying to catchup, it does so by sending `SYNC_CATCHUP` message to allow relay to forward the data from the most recent fully assembled frame(s) based on the **AFTERTIME** field.

[[todo: Manifest need not be as complicated as HLS/DASH support for the streaming use-cases supported by QuicR]]

[[todo: probably we need to add a note saying QuicR doesn't replace all streaming use-cases]]

Virtual/Augmented Reality, Gaming Applications

Applications, such as games or the ones based on virtual reality environment, have to need to share state about various objects across the network. This involves pariticipants sending small number of objects with state that need to be synchronized to the other side and it needs to be done periodically to keep the state up to date and also reach eventually consistency under losses.

Goals of such applications typically involve - Support 2D and 3D objects - Support delay and rollback based synchronization - Easily extensible for applications to send theirown custom data - Efficient distribution to multiple users

[[todo finsih this]]

Security

The key tenant of the security is that middles boxes are not trusted any more or less than the network. They both needed to be trusted to forward packets or the packets don't arrive but they should not have access to data or know which human is communicating with which human. They do have a need to understand what applications are using them.

End to End Encryption

The data transmitted is encrypted and authenticated end to end with a symmetric key provided out of band. Each publisher has their own key which is distributed to all the subscribers.

Fronting

A given origin relay may actually simply be fronting other relays behind it and effectively doing a NAT style name translation of the ResourceID. This allows for TOR like onion routing systems help preserve privacy of what participants are communicating.

Acknowledgements

Thanks to TODO for review and contributions.

TODO

1. Define trust establishment flows between QuicR Endpoints , Cloud Relays and the Origin Server. Also add security toke to the messages.
2. Messages needs some security considerations - integrity protection and so on.
3. Talk more about relay chaining
4. Define constructs for End to End Encryption
5. Fix the notation of the messages Thanks to for contributions and suggestions to this specification.