TODO Working Group
Internet-Draft
Intended status: Informational
Expires: 23 October 2025

K. Pugin
N. Garg
A. Frindell
J. Cenzano
J. Weissman
Meta

21 April 2025

RUSH - Reliable (unreliable) streaming protocol
draft-kpugin-rush-03

Abstract

RUSH is an application-level protocol for ingesting live video. This document describes the protocol and how it maps onto QUIC.

Discussion Venues

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

Discussion of this document takes place on the mailing list (), which is archived at .

Source for this draft and an issue tracker can be found at https://github.com/afrind/draft-rush.

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 23 October 2025.

Copyright Notice

Table of Contents

1. Introduction

RUSH is a bidirectional application level protocol designed for live video ingestion that runs on top of QUIC.

RUSH was built as a replacement for RTMP (Real-Time Messaging Protocol) with the goal to provide support for new audio and video codecs, extensibility in the form of new message types, and multi-track support. In addition, RUSH gives applications option to control data delivery guarantees by utilizing QUIC streams.

This document describes the RUSH protocol, wire format, and QUIC mapping.

2. Conventions 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.

Frame/Message: logical unit of information that client and server can exchange

PTS: presentation timestamp

DTS: decoding timestamp

AAC: advanced audio codec

NALU: network abstract layer unit

VPS: video parameter set (H265 video specific NALU)

SPS: sequence parameter set (H264/H265 video specific NALU)

PPS: picture parameter set (H264/H265 video specific NALU)

ADTS header: _Audio Data Transport Stream Header_

ASC: Audio specific config

GOP: Group of pictures, specifies the order in which intra- and inter-frames are arranged.

3. Theory of Operations

3.1. Connection establishment

In order to live stream using RUSH, the client establishes a QUIC connection using the ALPN token "rush".

After the QUIC connection is established, client creates a new bidirectional QUIC stream, choses starting frame ID and sends Connect frame Section 4.2.1 over that stream. This stream is called the Connect Stream.

The client sends mode of operation setting in Connect frame Section 4.2.1 payload.

One connection SHOULD only be used to send one media stream, for now 1 video and 1 audio track are supported. In the future we could send multiple tracks per stream.

3.2. Sending Video Data

The client can choose to wait for the ConnectAck frame Section 4.2.2 or it can start optimistically sending data immediately after sending the Connect frame.

A track is a logical organization of the data, for example, video can have one video track, and two audio tracks (for two languages). The client can send data for multiple tracks simultaneously.

The encoded audio or video data of each track is serialized into frames (see Section 4.2.6 or Section 4.2.5) and transmitted from the client to the server. Each track has its own monotonically increasing frame ID sequence. The client MUST start with initial frame ID = 1.

Depending on mode of operation (Section 4.3), the client sends audio and video frames on the Connect stream or on a new QUIC stream for each frame.

In Multi Stream Mode (Section 4.3.2), the client can stop sending a frame by resetting the corresponding QUIC stream. In this case, there is no guarantee that the frame was received by the server.

3.3. Receiving data

Upon receiving Connect frame Section 4.2.1, if the server accepts the stream, the server will reply with ConnectAck frame Section 4.2.2 and it will prepare to receive audio/video data.

It’s possible that in Multi Stream Mode (Section 4.3.2), the server receives audio or video data before it receives the Connect frame Section 4.2.1. The implementation can choose whether to buffer or drop the data. The audio/video data cannot be interpreted correctly before the arrival of the Connect frame Section 4.2.1.

In Single Stream Mode (Section 4.3.1), it is guaranteed by the transport that frames arrive into the application layer in order they were sent.

In Multi Stream Mode, it’s possible that frames arrive at the application layer in a different order than they were sent, therefore the server MUST keep track of last received frame ID for every track that it receives. A gap in the frame sequence ID on a given track can indicate out of order delivery and the server MAY wait until missing frames arrive. The server must consider frame lost if the corresponding QUIC stream was reset.

Upon detecting a gap in the frame sequence, the server MAY wait for the missing frames to arrive for an implementation defined time. If missing frames don’t arrive, the server SHOULD consider them lost and continue processing rest of the frames. For example if the server receives the following frames for track 1: 1 2 3 5 6 and frame #4 hasn’t arrived after implementation defined timeout, thee server SHOULD continue processing frames 5 and 6.

It is worth highlighting that in multi stream mode there is a need for a de-jitter function (that introduces latency). Also the subsequent processing pipeline should tolerate lost frames, so "holes" in the audio / video streams.

When the client is done streaming, it sends the End of Video frame (Section 4.2.3) to indicate to the server that there won’t be any more data sent.

3.4. Reconnect

If the QUIC connection is closed at any point, client MAY reconnect by simply repeat the Connection establishment process (Section 3.1) and resume sending the same video where it left off. In order to support termination of the new connection by a different server, the client SHOULD resume sending video frames starting with I-frame, to guarantee that the video track can be decoded from the 1st frame sent.

Reconnect can be initiated by the server if it needs to "go away" for maintenance. In this case, the server sends a GOAWAY frame (Section 4.2.7) to advise the client to gracefully close the connection. This allows client to finish sending some data and establish new connection to continue sending without interruption.

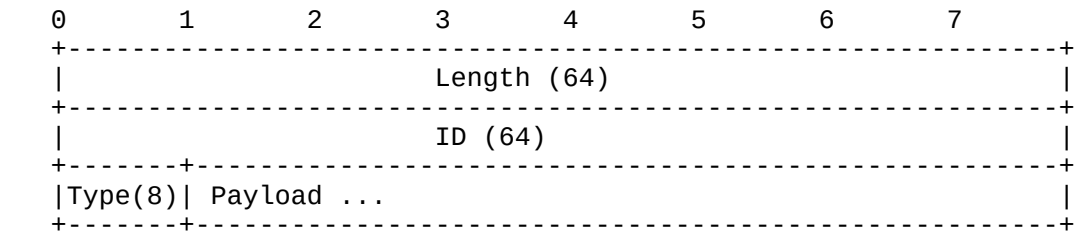
4. Wire Format

4.1. Frame Header

The client and server exchange information using frames. There are different types of frames and the payload of each frame depends on its type.

The bytes in the wire are in *big endian*

Generic frame format:



Length(64)': Each frame starts with length field, 64 bit size that tells size of the frame in bytes (including predefined fields, so if LENGTH is 100 bytes, then PAYLOAD length is 100 - 8 - 8 - 1 = 82 bytes).

ID(64): 64 bit frame sequence number, every new frame MUST have a sequence ID greater than that of the previous frame within the same track. Track ID would be specified in each frame. If track ID is not specified it's 0 implicitly.

Type(8): 1 byte representing the type of the frame.

Predefined frame types:

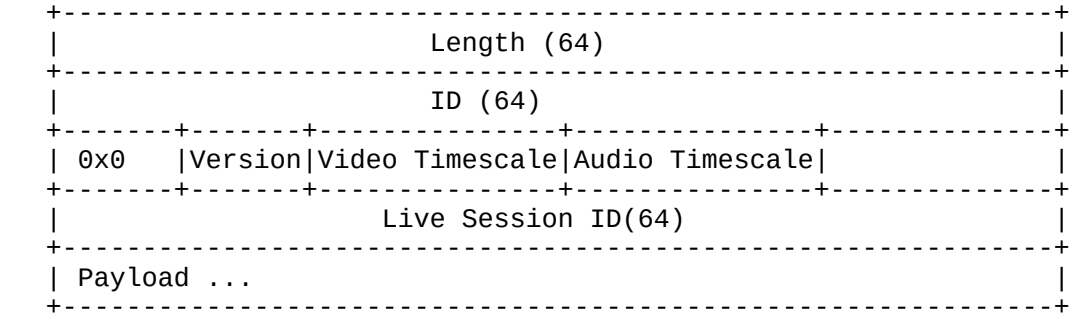
Frame Type	Frame
0x0	connect frame
0x1	connect ack frame
0x2	reserved
0x3	reserved
0x4	end of video frame

0x5	error frame
0x6	reserved
0x7	reserved
0x8	reserved
0x9	reserved
0xA	reserved
0XB	reserved
0xC	reserved
0xD	video frame
0xE	reserved
0XF	reserved
0X10	reserved
0x11	reserved
0x12	reserved
0x13	reserved
0x14	audio frame
0x15	GOAWAY frame
0x16	Timed metadata

Table 1

4.2. Frames

4.2.1. Connect frame



Version (unsigned 8bits): version of the protocol (initial version is 0x0).

Video Timescale(unsigned 16bits): timescale for all video frame timestamps on this connection. For instance 25

Audio Timescale(unsigned 16bits): timescale for all audio samples timestamps on this connection, recommended value same as audio sample rate, for example 44100

Live Session ID(unsigned 64bits): identifier of broadcast, when reconnect, client MUST use the same live session ID

Payload: application and version specific data that can be used by the server. OPTIONAL A possible implementation for this could be to add in the payload a UTF-8 encoded JSON data that specifies some parameters that server needs to authenticate / validate that connection, for instance: ~~~ payloadBytes = strToJSonUtf8('{"url": "/rtmp/BID?s_bl=1&s_l=3&s_sc=VALID&s_sw=0&s_vt=usr_dev&a=TOKEN"}') ~~~

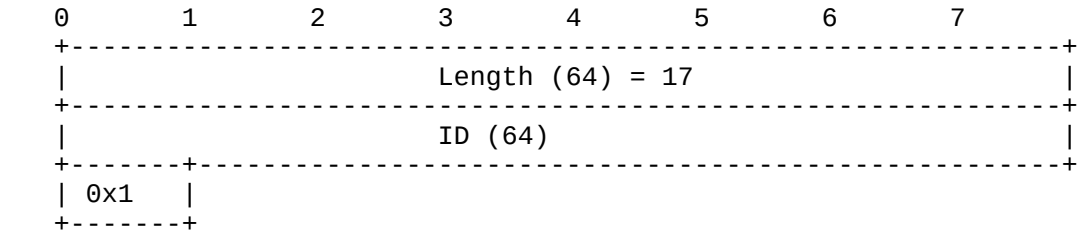
This frame is used by the client to initiate broadcasting. The client can start sending other frames immediately after Connect frame Section 4.2.1 without waiting acknowledgement from the server.

If server doesn't support VERSION sent by the client, the server sends an Error frame Section 4.2.4 with code UNSUPPORTED VERSION.

If audio timescale or video timescale are 0, the server sends error frame Section 4.2.4 with error code INVALID FRAME FORMAT and closes connection.

If the client receives a Connect frame from the server, the client sends an Error frame Section 4.2.4 with code TBD.

4.2.2. Connect Ack frame



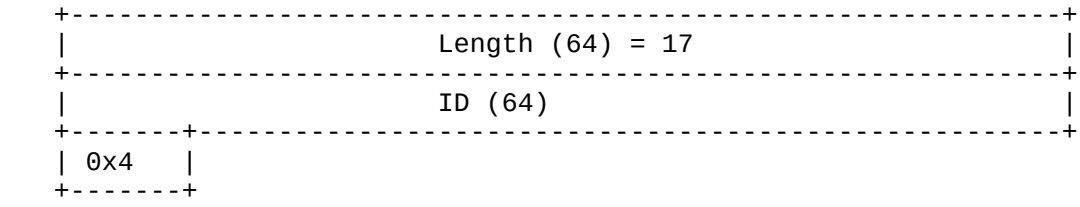
The server sends the "Connect Ack" frame in response to "Connect" Section 4.2.1 frame indicating that server accepts "version" and the stream is authenticated / validated (optional), so it is ready to receive data.

If the client doesn’t receive "Connect Ack" frame from the server within a timeout, it will close the connection. The timeout value is chosen by the implementation.

There can be only one "Connect Ack" frame sent over lifetime of the QUIC connection.

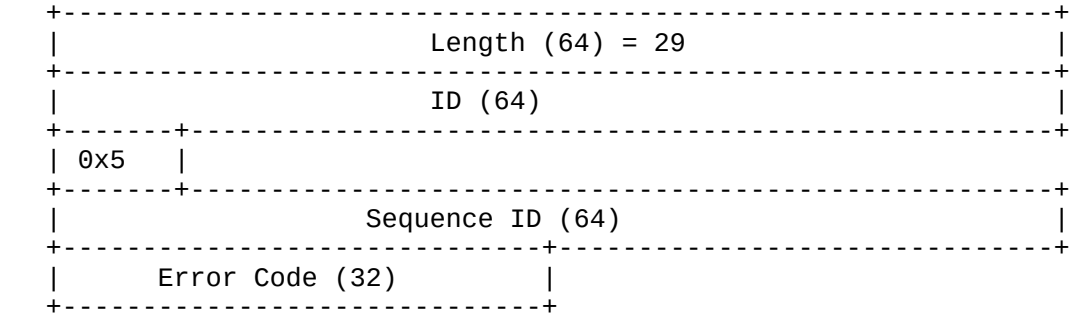
If the server receives a Connect Ack frame from the client, the client sends an Error frame with code TBD.

4.2.3. End of Video frame



End of Video frame is sent by a client when it’s done sending data and is about to close the connection. The server SHOULD ignore all frames sent after that.

4.2.4. Error frame



Sequence ID(unsigned 64bits): ID of the frame sent by the client that error is generated for, ID=0x0 indicates connection level error.

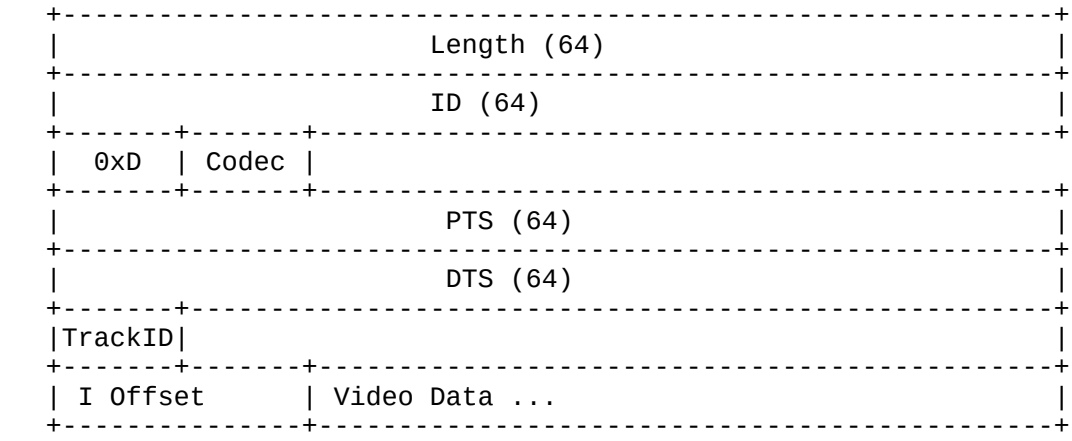
Error Code(unsigned 32bits): Indicates the error code

Error frame can be sent by the client or the server to indicate that an error occurred.

Some errors are fatal and the connection will be closed after sending the Error frame.

See section Section 5.1 and Section 5.2 for more information about error codes

4.2.5. Video frame



Codec (unsigned 8bits): specifies codec that was used to encode this frame.

PTS (signed 64bits): presentation timestamp in connection video timescale

DTS (signed 64bits): decoding timestamp in connection video timescale

Supported type of codecs:

Type	Codec
0x1	H264
0x2	H265
0x3	VP8
0x4	VP9

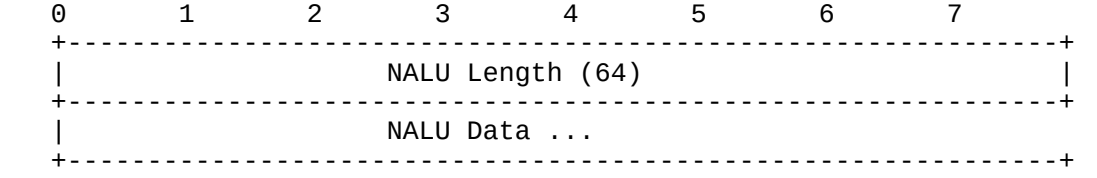
Table 2

Track ID (unsigned 8bits): ID of the track that this frame is on

I Offset (unsigned 16bits): Distance from sequence ID of the I-frame that is required before this frame can be decoded. This can be useful to decide if frame can be dropped.

Video Data: variable length field, that carries actual video frame data that is codec dependent

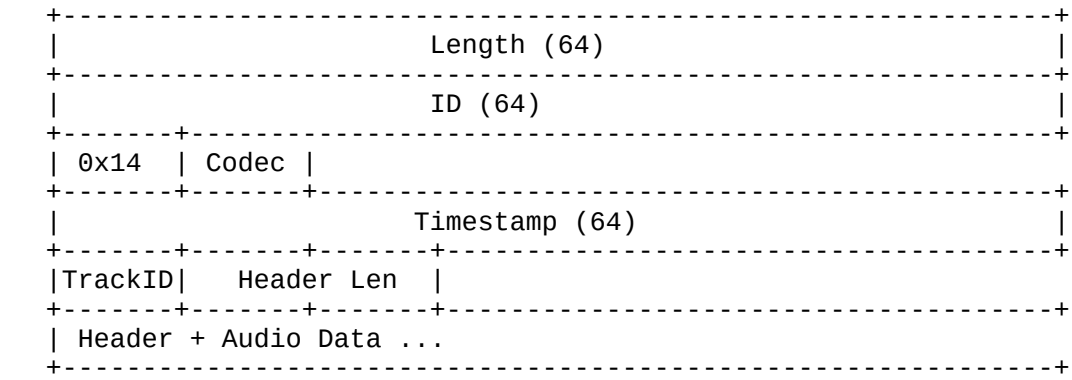
For h264/h265 codec, "Video Data" are 1 or more NALUs in AVCC format (4 bytes size header):



EVERY h264 video key-frame MUST start with SPS/PPS NALUs. EVERY h265 video key-frame MUST start with VPS/SPS/PPS NALUs.

Binary concatenation of "video data" from consecutive video frames, without data loss MUST produce VALID h264/h265 bitstream.

4.2.6. Audio frame



Codec (unsigned 8bits): specifies codec that was used to encode this frame.

Supported type of codecs:

Type	Codec
0x1	AAC
0x2	OPUS

Table 3

Timestamp (signed 64bits): timestamp of first audio sample in Audio Data.

Track ID (unsigned 8bits): ID of the track that this frame is on

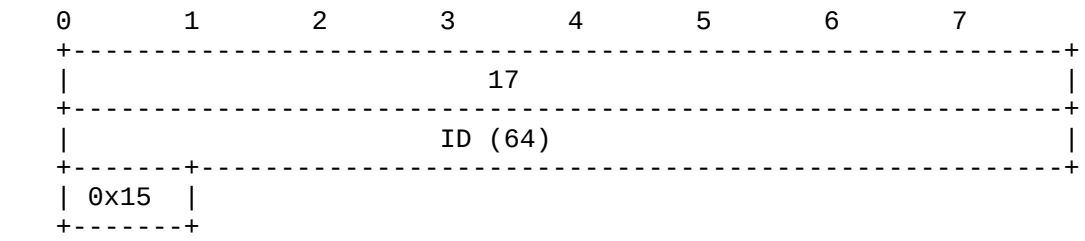
Header Len (unsigned 16bits): Length in bytes of the audio header contained in the first portion of the payload

Audio Data (variable length field): it carries the audio header and 1 or more audio frames that are codec dependent.

For AAC codec: - "Audio Data" are 1 or more AAC samples, prefixed with Audio Specific Config (ASC) header defined in ISO 14496-3 - Binary concatenation of all AAC samples in "Audio Data" from consecutive audio frames, without data loss MUST produce VALID AAC bitstream.

For OPUS codec: - "Audio Data" are 1 or more OPUS samples, prefixed with OPUS header as defined in [RFC7845]

4.2.7. GOAWAY frame

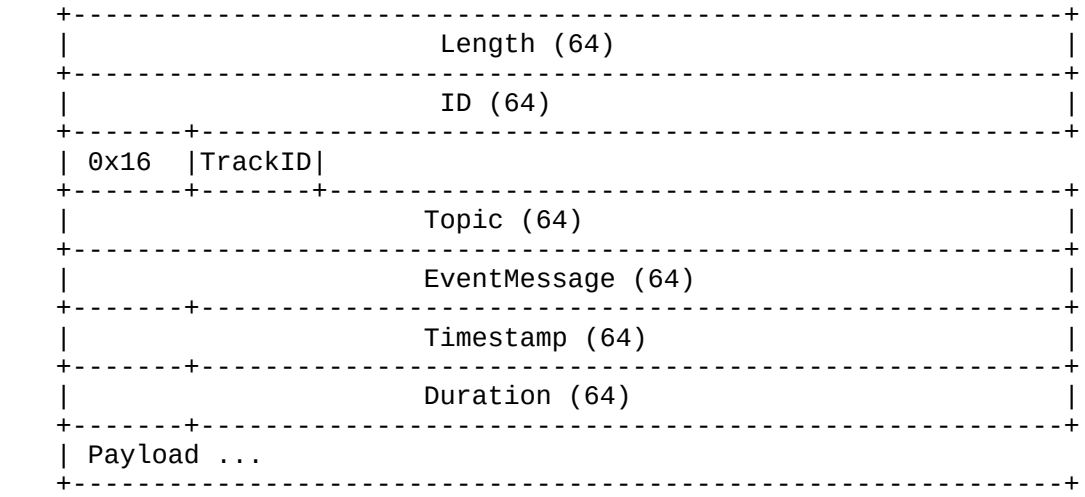


The GOAWAY frame is used by the server to initiate graceful shutdown of a connection, for example, for server maintenance.

Upon receiving GOAWAY frame, the client MUST send frames remaining in current GOP and stop sending new frames on this connection. The client SHOULD establish a new connection and resume sending frames there, so when resume video frame will start with an IDR frame.

After sending a GOAWAY frame, the server continues processing arriving frames for an implementation defined time, after which the server SHOULD close the connection.

4.2.8. TimedMetadata frame



Track ID (unsigned 8bits): ID of the track that this frame is on

Timestamp (signed 64bits): PTS of the event

Topic (unsigned 64bits): A unique identifier of the app level feature. May be used to decode payload or do other application specific processing

EventMessage (unsigned 64bits): A unique identifier of the event message used for app level events deduplication

Duration (unsigned 64bits): duration of the event in video PTS timescale. Can be 0.

Payload: variable length field. May be used by the app to send additional event metadata. UTF-8 JSON recommended

4.3. QUIC Mapping

One of the main goals of the RUSH protocol was ability to provide applications a way to control reliability of delivering audio/video data. This is achieved by using a special mode Section 4.3.2.

4.3.1. Single Stream Mode

In single stream mode, RUSH uses one bidirectional QUIC stream to send data and receive data. Using one stream guarantees reliable, in-order delivery - applications can rely on QUIC transport layer to retransmit lost packets. The performance characteristics of this mode are similar to RTMP over TCP.

4.3.2. Multi Stream Mode

In single stream mode Section 4.3.1, if packet belonging to video frame is lost, all packets sent after it will not be delivered to application, even though those packets may have arrived at the server. This introduces head of line blocking and can negatively impact latency.

To address this problem, RUSH defines "Multi Stream Mode", in which one QUIC stream is used per audio/video frame.

Connection establishment follows the normal procedure by client sending Connect frame, after that Video and Audio frames are sent using following rules:

* Each new frame is sent on new bidirectional QUIC stream

* Frames within same track must have IDs that are monotonically increasing, such that ID(n) = ID(n-1) + 1

The receiver reconstructs the track using the frames IDs.

Response Frames (Connect AckSection 4.2.2 and ErrorSection 4.2.4), will be in the response stream of the stream that sent it.

The client MAY control delivery reliability by setting a delivery timer for every audio or video frame and reset the QUIC stream when the timer fires. This will effectively stop retransmissions if the frame wasn't fully delivered in time.

Timeout is implementation defined, however future versions of the draft will define a way to negotiate it.

5. Error Handling

An endpoint that detects an error SHOULD signal the existence of that error to its peer. Errors can affect an entire connection (see Section 5.1), or a single frame (see Section 5.2).

The most appropriate error code SHOULD be included in the error frame that signals the error.

5.1. Connection Errors

Affects the the whole connection:

1 - UNSUPPORTED VERSION - indicates that the server doesn't support version specified in Connect frame 4- CONNECTION_REJECTED - Indicates the server can not process that connection for any reason

5.2. Frame errors

There are two error codes defined in core protocol that indicate a problem with a particular frame:

2 - UNSUPPORTED CODEC - indicates that the server doesn't support the given audio or video codec

3 - INVALID FRAME FORMAT - indicates that the receiver was not able to parse the frame or there was an issue with a field's value.

6. Extensions

RUSH permits extension of the protocol.

Extensions are permitted to use new frame types (Section 4), new error codes (Section 4.2.4), or new audio and video codecs (Section 4.2.6, Section 4.2.5).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements, except codec id, which returns an UNSUPPORTED CODEC error. Implementations MUST discard frames that have unknown or unsupported types.

7. Security Considerations

RUSH protocol relies on security guarantees provided by the transport.

Implementation SHOULD be prepared to handle cases when sender deliberately sends frames with gaps in sequence IDs.

Implementation SHOULD be prepare to handle cases when server never receives Connect frame (Section 4.2.1).

A frame parser MUST ensure that value of frame length field (see Section 4.1) matches actual length of the frame, including the frame header.

Implementation SHOULD be prepare to handle cases when sender sends a frame with large frame length field value.

8. IANA Considerations

TODO: add frame type registry, error code registry, audio/video codecs registry

9. 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>.

[RFC7845] Terriberry, T., Lee, R., and R. Giles, "Ogg Encapsulation for the Opus Audio Codec", RFC 7845, DOI 10.17487/RFC7845, April 2016, <https://www.rfc-editor.org/rfc/rfc7845>.

[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>.

Acknowledgments

This draft is the work of many people: Vlad Shubin, Nitin Garg, Milen Lazarov, Benny Luo, Nick Ruff, Konstantin Tsoy, Nick Wu.

Authors' Addresses

Kirill Pugin
Meta
Email: ikir@meta.com

Nitin Garg
Meta
Email: ngarg@meta.com

Alan Frindell
Meta
Email: afrind@meta.com

Jordi Cenzano
Meta
Email: jcenzano@meta.com

Jake Weissman
Meta
Email: jakeweissman@meta.com