avtcore WG
Internet-Draft
Intended status: Informational
Expires: 17 September 2025

E. Sprꬽng
Google

16 March 2025

RTP Header Extension for Automatic Detection of Video Corruptions
draft-sprang-avtcore-corruption-detection-00

Abstract

This document describes an RTP header extensions that can be use to carry select filtered samples from a video frame together with metrics relating to the expected distortions caused by lossy compression of said frame. This data allows a receiver to validate that the output from a decoder matches the frame the went into the encoder on the remote - i.e. validating that the video pipeline is in a correct state free of any video corruptions.

About This Document

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

The latest revision of this draft can be found at https://github.com/sprangerik/corruption-detection/blob/main/draft-sprang-avtcore-corruption-detection.md. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-sprang-avtcore-corruption-detection/.

Discussion of this document takes place on the avtcore WG Working Group mailing list (mailto:avt@ietf.org), which is archived at https://datatracker.ietf.org/wg/avtcore. Subscribe at https://www.ietf.org/mailman/listinfo/avt/.

Source for this draft and an issue tracker can be found at https://github.com/sprangerik/corruption-detection.

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/.

Copyright Notice

Table of Contents

1. Introduction

The Corruption Detection (sometimes referred to as automatic corruption detection or ACD) extension is intended to be a part of a system that allows estimating a likelihood that a video transmission is in a valid state. That is, the input to the video encoder on the send side corresponds to the output of the video decoder on the receive side with the only difference being the expected distortions from lossy compression.

The goal is to be able to detect outright coding errors caused by things such as bugs in encoder/decoders, malformed packetization data, incorrect relay decisions in SFU-type servers, incorrect handling of packet loss/reordering, and so forth. This should be accomplished with a high signal-to-noise ratio while consuming a minimum of resources in terms of bandwidth and/or computation. It should be noted that it is not a goal to be able to e.g. gauge general video quality using this method.

This is accomplished in two steps. On the sender side:

* Pseudo randomly (using a Halton Sequence) sampling a small set of locations within the input to the encoder.

* Applying a gaussian low-pass filter in order to supress encoding distortions.

* Determining an "allowed error" threshold, below which samples are not determined as erroneus.

Then, on the receiver side:

* Sampling the same locations as the sender, but in the output image from the decoder.

* Applying the same low-pass filter as the sender

* Comparing the values in the samples provided in the header extension with those sample from the output image, subtracting the allowed error from each sample pair.

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.

3. Applicability

The Corruption Detection extension can be used to validate the correctness of any video pipeline in an end-to-end manner, as long as there is no termination of the bitstream in the process. It is designed to be codec agnostic.

In terms of [RFC7667], any Point-to-Point or Transport Translator topologies can be used. The header extension just needs to be propagated together with the media packet until the decode stage. If however a Media Translator is used, the header extension needs to be extracted at the time of decoding there (e.g during transcoding). If corruption detection is desired to the end user, then a new Corruption Detection header extension needs to be created as part of the middlebox encoding step and thet can then be carried forward.

4. Corruption Detection

Name: "Corruption Detection"; "Extension for Automatic Detection of Video Corruptions"

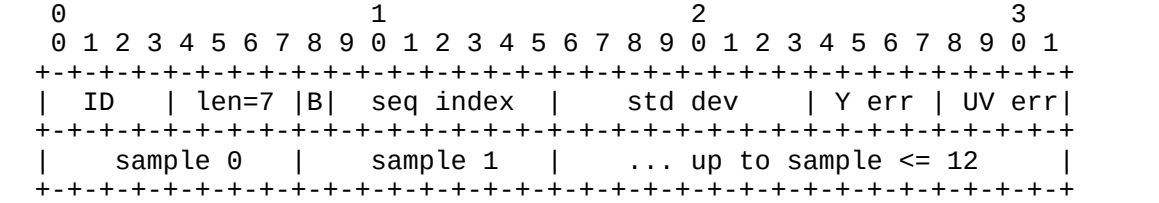
Formal name: http://www.webrtc.org/experiments/rtp-hdrext/corruption-detection

Status: This extension is defined here to allow for experimentation. Experience with the experiment has shown that it is useful; this draft therefore presents it to the IETF for consideration of whether to standardize it or leave it as a proprietary extension.

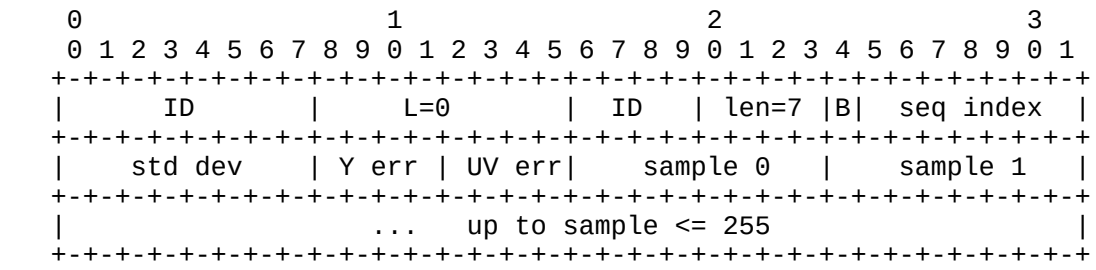
4.1. RTP header extension format

4.1.1. Data layout overview

Data layout of a corruption-detection header extension, using a One-Byte header (see [RFC5285]) section 4.2:



Data layout of a corruption-detection header extension, using a Two-Byte header (see [RFC5285]) section 4.3:



4.1.2. Data layout details

4.1.2.1. B (1 bit)

Whether the sequence number should be interpreted as the MSB or LSB of the full size 14 bit sequence index described in the next point.

4.1.2.2. seq index (7 bits)

The index into the Halton sequence (used to locate where the samples should be drawn from).

If B is set: the 7 most significant bits of the true index. The 7 least significant bits of the true index shall be interpreted as 0. This is because this is the point where we can guarantee that the sender and receiver has the same full index. B MUST be set on keyframes. On droppable frames B MUST NOT be set. If B is not set: The 7 LSB of the true index. The 7 most significant bits should be inferred based on the most recent message.

4.1.2.3. std dev (8 bits)

The standard deviation of the Gaussian filter used to weigh the samples. The value is scaled using a linear map: 0 = 0.0 to 255 = 40.0. A std dev of 0 is interpreted as directly using just the sample value at the desired coordinate, without any weighting.

4.1.2.4. Y err (4 bits)

The allowed error for the luma channel.

4.1.2.5. UV err (4 bits)

The allowed error for the chroma channels.

4.1.2.6. Sample N (8 bits)

The N:th filtered sample from the input image. Each sample represents a new point in one of the image planes, the plane and coordinates being determined by index into the Halton sequence (starting at seq# index and is incremented by one for each sample). Each sample has gone through a Gaussian filter with the std dev specified above. The samples have been floored to the nearest integer.

4.1.3. Synchronization Message

A special case is the so-called "synchronization" message. Such a message only contains the first byte. They are used to keep the sender and receiver in sync even if no "full" message has been received for a while. Such messages MUST NOT be sent on droppable frames.

4.2. Details of Operation

4.2.1. Halton Sequence Handling

The sender and receiver need to find the same set of pseudo random sampling locations for an instrumented frame. Sending explicit coordinates for each sample would however use a significant amount of bandwidth, in fact potentially several times the size of the sample values themselves.

To avoid this overhead, we instead define that a 2D Halton Sequence is used to generate the sample coordinates - meaning that the clients only need to keep track of the index into that sequence in order to generate the desired sequence of sampling coordinates.

The bases used are 2 for the row and 3 for the column.

4.2.2. Halton Sequence Algorithm

In pseudo code, the Halton sequence can implemented as follows:

Inputs: index i, base b Where b is 2 when calculating the row (y-axis) and 3 when calculating the column (x-axis)

```
f = 1;
r = 0;
while i > 0 do {
  f = f / b;
  r = r + f * (i mod b);
  i = floor(i / b);
}
return r;  // Coordinate component.
```

4.2.2.1. Sequence Index Counter

The sequence index into the Halton sequence is a 14 bit unsigned integer, which wraps around to zero on overflow. Only half of those bits are transmitted with any given message.

If B = 1 then the 7 most significant bits are transmitted and the 7 least significant bits are implicitly set to 0. If B = 1 then the 7 least significant bits are transmitted and the most significant bits are inferred for the current state.

For keyframes, the sender MUST set the value of B to 1.

For droppable frames (e.g. non-baselayer frames when using temporal layers), the sender MUST set B to 0 to prevent sender and receiver from out of sync should that frame be lost.

4.2.2.2. Sequence Index Stepping

A sequence index MUST be present on all keyframes of an instrumented video stream, however there are no requirements around which sequence index to start at; the sender may choose that arbitrarily. This means we always start with a known index.

For each sample within the corruption detection header extension, the sequence index is assumed to be incremented by one. This allows a variable amount of samples to be present in an extension, if so desired.

When droppable frames are used (e.g. when using temporal layering), a middle box may drop some frames resulting in a gap between the last sequence of the previous extension and sequence index indicated by the header of the next extension. If this happens and the new frame has B = 0, then the receiver is intended to just increment the sequence number index until the 7 least significant bits in the index matches the value in the extension header (possibly including an increment of the more significant bits).

This means that the sender MUST NOT add >= 127 consecutive samples to droppable frames, otherwise the most significant bits may come out of sync.

If needed, a "sync message" can be added to guarantee sequence index alignment even if no samples are available. This is done by sending a truncated message containing just B and the sequence index.

4.2.3. Spatial Layer Considerations

When multiple spatial layers are present within a single SSRC, the sender MUST produce a separate and independent stream of corruption detection headers for each spatial layer. This ensures that a receiver can decode and verify the highest spatial layer that is part of the stream they are receiving, and any layers culled by a middlebox does not affect the integrity of e.g. the sequence index series for that layer.

When using a scalability mode where a higher spatial layer uses inter-layer prediction (prediction between frames belonging to the same temporal unit, e.g. the "SxTx" modes in the W3C [SVC] spec), then the frame should be treated as a key-frame if any frame in the dependency chain within that temporal layer is a key-frame.

4.2.4. Sample Filtering

The std dev field of the extension indicates the standard deviation of a 2D Gaussian blur kernel that should be applied taking samples - both to the raw input frame before encoding at the sender and to the raw output frame from the decoder at the receiver.

Further, in order to reduce the number of samples the algorithm needs to average - it is optionally allowed to ignore parts of the image where weight from the Gaussian becomes small. In practice, weights below 0.2 have shown to be small enough that they have negligible effect on the end result.

Any samples outside the plane are considered to have weight 0.

Note that this processing is done on a per-image-plane basis.

In pseudo-code, that means we get the following:

```
// Translate 8bit header value to floating point.
stddev = header.std_dev * (40.0 / 255);
// Max number of samples away from the center.
max_d = ceil(sqrt(-2.0 * ln(0.2) * stddev^2) - 1;
sample_sum = 0;
weight_sum = 0;
for (y = max(0, row - max_d) to min(plane_height, row + max_d) {
  for (x = max(0, col - max_d) to min(plane_width, col + max_d) {
    weight = e^(-1 * ((y - row)^2 + (x - col)^2) / (2 * stddev^2));
    sample_sum += SampleAt(x, y) * weight;
    weight_sum += weight;
  }
}
filtered_sample = sample_sum / weight_sum;
```

4.2.5. Handling of Image Planes

For now this header extension is only defined for 4:2:0 chroma subsampling. // TODO: Discuss supporting other formats.

In order to translate the row/column calculated from the Halton sequence into a coordinate within a given image plane, visualize the U/V (chroma) planes as being attached to the Y (luma) planes as follows:

```
+------+---+
|      | U |
+  Y   +---+
|      | V |
+------+---+
```

In pseudo code:

```
row = GetHaltonSequence(seq_index, /*base=*/2) * image_height;
col = GetHaltonSequence(seq_index, /*base=*/3) * image_width * 1.5;

if (col < image_width) {
  HandleSample(Y_PLANE, row, col);
} else if (row < image_height / 2) {
  HandleSample(U_PLANE, row, col - image_width);
} else {
  HandleSample(V_PLANE, row - (image_height / 2), col - image_width);
}
```

4.2.6. Allowed Error Thresholds

The header extension contains two fields for an "allowed error"; one for the luma channel and one for the chrome channels. The two values allow for accommodating different spatial resolutions and thus different error magnitudes for the respective image planes.

When calculating the difference between two samples in a receiving client (i.e. one locally filtered sample and the corresponding remote sample from the header extension) the absolute magnitude of the error should be reduced by the specified amount.

4.2.7. Determining Filter Size and Error Thresholds

The filter size (std dev of the blur kernel) and the allowed error thresholds SHOULD be set such that 99.5% of filtered samples end up with a delta <= the error threshold for that plane, based on a representative set of test clips and bandwidth constraints.

This text does not put restrictions on how exactly an implementation should determine appropriate thresholds, but a straightforward way is to look at the QP and codec type as this translates roughly the expected compression distortion. Individual implementations may exhibit better or worse performance than an "average" encoder implementation for a given codec type - so a sender is encouraged to compensate for this if the implementation is far outside the expected bounds based on averages.

4.2.8. Calculating A Corruption Score

This text does not put exact requirements on how the differences calculated should be translated into an output. While it is tempting to try and use the 99.5% percentile aspect and use a simple statistical method to estimate the probability of this magnitude of error with a given confidence interval - in practice that has proven tricky due to the number outliers that don’t quite follow a normal distribution.

In practice, it has been found that just taking the sum of all differences squared and dividing by two yields a reasonably good result.

Further improvements in this area should be possible without having to make changes to the message format itself.

5. Security Considerations

The de facto way of encrypting RTP packets is using SRTP as defined in [RFC3711]. Unfortunately, SRTP does not provide encryption for header extensions as they are not seen as part of the payload. That poses a serious security risk, as the corruption detection samples are in essence a very sparse encoding of the source image - so given a sufficient number of samples and a static scene an eavesdropper could rather trivially reconstruct the scene from just header extensions.

While technically not required, the sender SHOULD negotiate some form of encryption that covers this header extension. The most straightforward methods is to use either [RFC6904] or [RFC9335], both of which are specifically designed to allow selective encryption of sensitive RTP header extensions.

6. IANA Considerations

If the WG decides that this extension should be registered as a standardized extension, IANA is requested to perform the appropriate registration.

If the WG decides that this is a private extension, the URL http://www.webrtc.org/experiments/rtp-hdrext/corruption-detection is used to identify the extension.

7. References

7.1. 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>.

[RFC5285] Singer, D. and H. Desineni, "A General Mechanism for RTP Header Extensions", RFC 5285, DOI 10.17487/RFC5285, July 2008, <https://www.rfc-editor.org/rfc/rfc5285>.

[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>.

7.2. Informative References

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <https://www.rfc-editor.org/rfc/rfc3711>.

[RFC6904] Lennox, J., "Encryption of Header Extensions in the Secure Real-time Transport Protocol (SRTP)", RFC 6904, DOI 10.17487/RFC6904, April 2013, <https://www.rfc-editor.org/rfc/rfc6904>.

[RFC7667] Westerlund, M. and S. Wenger, "RTP Topologies", RFC 7667, DOI 10.17487/RFC7667, November 2015, <https://www.rfc-editor.org/rfc/rfc7667>.

[RFC9335] Uberti, J., Jennings, C., and S. Murillo, "Completely Encrypting RTP Header Extensions and Contributing Sources", RFC 9335, DOI 10.17487/RFC9335, January 2023, <https://www.rfc-editor.org/rfc/rfc9335>.

[SVC] W3C, "Scalable Video Coding (SVC) Extension for WebRTC", n.d., <https://www.w3.org/TR/webrtc-svc>.

Acknowledgments

Fanny Linderborg and Emil Vardar for their contributions in implementing and evaluating the corruption detection mechanism.

Author’s Address

Erik Språng
Google
Email: sprang@google.com