MOPS
Internet-Draft
Intended status: Standards Track
Expires: 13 March 2021

M.P. Sharabayko
M.A. Sharabayko
Haivision Network Video, GmbH
J. Dube
Haivision
JS. Kim
JW. Kim
SK Telecom Co., Ltd.

9 September 2020

The SRT Protocol
draft-sharabayko-mops-srt-01

Abstract

This document specifies Secure Reliable Transport (SRT) protocol. SRT is a user-level protocol over User Datagram Protocol and provides reliability and security optimized for low latency live video streaming, as well as generic bulk data transfer. For this, SRT introduces control packet extension, improved flow control, enhanced congestion control and a mechanism for data encryption.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

1.1. Motivation

The demand for live video streaming has been increasing steadily for many years. With the emergence of cloud technologies, many video processing pipeline components have transitioned from on-premises appliances to software running on cloud instances. While real-time streaming over TCP-based protocols like RTMP [RTMP] is possible at low bitrates and on a small scale, the exponential growth of the streaming market has created a need for more powerful solutions.

To improve scalability on the delivery side, content delivery networks (CDNs) at one point transitioned to segmentation-based technologies like HLS (HTTP Live Streaming) [RFC8216] and DASH (Dynamic Adaptive Streaming over HTTP) [ISO23009]. This move increased the end-to-end latency of live streaming to over 30 seconds, which makes it unattractive for many use cases. Over time, the industry optimized these delivery methods, bringing the latency down to 3 seconds.

While the delivery side scaled up, improvements to video transcoding became a necessity. Viewers watch video streams on a variety of different devices, connected over different types of networks. Since upload bandwidth from on-premises locations is often limited, video transcoding moved to the cloud.

RTMP became the de facto standard for contribution over the public Internet. But there are limitations for the payload to be transmitted, since RTMP as a media specific protocol only supports two audio channels and a restricted set of audio and video codecs, lacking support for newer formats such as HEVC [H.265], VP9 [VP9], or AV1 [AV1].

Since RTMP, HLS and DASH rely on TCP, these protocols can only guarantee acceptable reliability over connections with low RTTs, and can not use the bandwidth of network connections to their full extent due to limitations imposed by congestion control. Notably, QUIC [I-D.ietf-quic-transport] has been designed to address these problems with HTTP-based delivery protocols in HTTP/3 [I-D.ietf-quic-http]. Like QUIC, SRT [SRTSRC] uses UDP instead of the TCP transport protocol, but assures more reliable delivery using Automatic Repeat Request (ARQ), packet acknowledgments, end-to-end latency management, etc.

1.2. Secure Reliable Transport Protocol

Low latency video transmissions across reliable (usually local) IP based networks typically take the form of MPEG-TS [ISO13818-1] unicast or multicast streams using the UDP/RTP protocol, where any packet loss can be mitigated by enabling forward error correction (FEC). Achieving the same low latency between sites in different cities, countries or even continents is more challenging. While it is possible with satellite links or dedicated MPLS [RFC3031] networks, these are expensive solutions. The use of public Internet connectivity, while less expensive, imposes significant bandwidth overhead to achieve the necessary level of packet loss recovery. Introducing selective packet retransmission (reliable UDP) to recover from packet loss removes those limitations.

Derived from the UDP-based Data Transfer (UDT) protocol [GHG04b], SRT is a user-level protocol that retains most of the core concepts and mechanisms while introducing several refinements and enhancements, including control packet modifications, improved flow control for handling live streaming, enhanced congestion control, and a mechanism for encrypting packets.

SRT is a transport protocol that enables the secure, reliable transport of data across unpredictable networks, such as the Internet. While any data type can be transferred via SRT, it is ideal for low latency (sub-second) video streaming. SRT provides improved bandwidth utilization compared to RTMP, allowing much higher contribution bitrates over long distance connections.

As packets are streamed from source to destination, SRT detects and adapts to the real-time network conditions between the two endpoints, and helps compensate for jitter and bandwidth fluctuations due to congestion over noisy networks. Its error recovery mechanism minimizes the packet loss typical of Internet connections.

To achieve low latency streaming, SRT had to address timing issues. The characteristics of a stream from a source network are completely changed by transmission over the public Internet, which introduces delays, jitter, and packet loss. This, in turn, leads to problems with decoding, as the audio and video decoders do not receive packets at the expected times. The use of large buffers helps, but latency is increased. SRT includes a mechanism to keep a constant end-to-end latency, thus recreating the signal characteristics on the receiver side, and reducing the need for buffering.

Like TCP, SRT employs a listener/caller model. The data flow is bi-directional and independent of the connection initiation - either the sender or receiver can operate as listener or caller to initiate a connection. The protocol provides an internal multiplexing mechanism, allowing multiple SRT connections to share the same UDP port, providing access control functionality to identify the caller on the listener side.

Supporting forward error correction (FEC) and selective packet retransmission (ARQ), SRT provides the flexibility to use either of the two mechanisms or both combined, allowing for use cases ranging from the lowest possible latency to the highest possible reliability.

SRT maintains the ability for fast file transfers introduced in UDT, and adds support for AES encryption.

2. Terms and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

SRT: The Secure Reliable Transport protocol described by this document.

PRNG: Pseudo-Random Number Generator.

3. Packet Structure

SRT packets are transmitted as UDP payload [RFC0768]. Every UDP packet carrying SRT traffic contains an SRT header immediately after the UDP header (Figure 1).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            SrcPort            |            DstPort            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Len              |            ChkSum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                          SRT Packet                           +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: SRT packet as UDP payload

SRT has two types of packets distinguished by the Packet Type Flag: data packet and control packet.

The structure of the SRT packet is shown in Figure 2.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- SRT Header +-+-+-+-+-+-+-+-+-+-+-+-+-+
|F|        (Field meaning depends on the packet type)           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          (Field meaning depends on the packet type)           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination Socket ID                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                        Packet Contents                        |
|                  (depends on the packet type)                 +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: SRT packet structure

F: 1 bit. Packet Type Flag. The control packet has this flag set to "1". The data packet has this flag set to "0".

Timestamp: 32 bits. The timestamp of the packet, in microseconds. The value is relative to the time the SRT connection was established. Depending on the transmission mode (Section 4.2), the field stores the packet send time or the packet origin time.

Destination Socket ID: 32 bits. A fixed-width field providing the SRT socket ID to which a packet should be dispatched. The field may have the special value "0" when the packet is a connection request.

3.1. Data Packets

The structure of the SRT data packet is shown in Figure 3.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+- SRT Header +-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|                    Packet Sequence Number                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|P P|O|K K|R|                   Message Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination Socket ID                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                              Data                             +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3: Data packet structure

Packet Sequence Number: 31 bits. The sequential number of the data packet.

PP: 2 bits. Packet Position Flag. This field indicates the position of the data packet in the message. The value "10b" (binary) means the first packet of the message. "00b" indicates a packet in the middle. "01b" designates the last packet. If a single data packet forms the whole message, the value is "11b".

O: 1 bit. Order Flag. Indicates whether the message should be delivered by the receiver in order (1) or not (0). Certain restrictions apply depending on the data transmission mode used (Section 4.2).

KK: 2 bits. Key-based Encryption Flag. The flag bits indicate whether or not data is encrypted. The value "00b" (binary) means data is not encrypted. "01b" indicates that data is encrypted with an even key, and "10b" is used for odd key encryption. Refer to Section 5. The value "11b" is only used in control packets.

R: 1 bit. Retransmitted Packet Flag. This flag is clear when a packet is transmitted the first time. The flag is set to "1" when a packet is retransmitted.

Message Number: 26 bits. The sequential number of consecutive data packets that form a message (see PP field).

Timestamp: 32 bits. See Section 3.

Destination Socket ID: 32 bits. See Section 3.

Data: variable length. The payload of the data packet. The length of the data is the remaining length of the UDP packet.

3.2. Control Packets

An SRT control packet has the following structure.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+- SRT Header +-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|         Control Type        |            Subtype            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Type-specific Information                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination Socket ID                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- CIF -+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                   Control Information Field                   +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4: Control packet structure

Control Type: 15 bits. Control Packet Type. The use of these bits is determined by the control packet type definition. See Table 1.

Subtype: 16 bits. This field specifies an additional subtype for specific packets. See Table 1.

Type-specific Information: 32 bits. The use of this field depends on the particular control packet type. Handshake packets do not use this field.

Timestamp: 32 bits. See Section 3.

Destination Socket ID: 32 bits. See Section 3.

Control Information Field (CIF): variable length. The use of this field is defined by the Control Type field of the control packet.

The types of SRT control packets are shown in Table 1. The value "0x7FFF" is reserved for a user-defined type.

Packet Type	Control Type	Subtype	Section
HANDSHAKE	0x0000	0x0	Section 3.2.1
KEEPALIVE	0x0001	0x0	Section 3.2.3
ACK	0x0002	0x0	Section 3.2.4
NAK (Loss Report)	0x0003	0x0	Section 3.2.5
SHUTDOWN	0x0005	0x0	Section 3.2.6
ACKACK	0x0006	0x0	Section 3.2.7
User-Defined Type	0x7FFF	-	N/A

Table 1: SRT Control Packet Types

3.2.1. Handshake

Handshake control packets (Control Type = 0x0000) are used to exchange peer configurations, to agree on connection parameters, and to establish a connection.

The Control Information Field (CIF) of a handshake control packet is shown in Figure 5.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                            Version                            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |        Encryption Field       |        Extension Field        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                 Initial Packet Sequence Number                |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                 Maximum Transmission Unit Size                |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Maximum Flow Window Size                   |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                         Handshake Type                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                         SRT Socket ID                         |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                           SYN Cookie                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                               |
   +                                                               +
   |                                                               |
   +                        Peer IP Address                        +
   |                                                               |
   +                                                               +
   |                                                               |
   +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
   |         Extension Type        |        Extension Length       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                               |
   +                       Extension Contents                      +
   |                                                               |
   +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure 5: Handshake packet structure

Version: 32 bits. A base protocol version number. Currently used values are 4 and 5. Values greater than 5 are reserved for future use.

Encryption Field: 16 bits. Block cipher family and key size. The values of this field are described in Table 2. The default value is AES-128.

Value	Cipher family and key size
0	No Encryption Advertised
2	AES-128
3	AES-192
4	AES-256

Table 2: Handshake Encryption Field Values

Extension Field: 16 bits. This field is message specific extension related to Handshake Type field. The value MUST be set to 0 except for the following cases. (1) If the handshake control packet is the INDUCTION message, this field is sent back by the Listener. (2) In the case of a CONCLUSION message, this field value should contain a combination of Extension Type values. For more details, see Section 4.3.1.

Bitmask	Flag
0x00000001	HSREQ
0x00000002	KMREQ
0x00000004	CONFIG

Table 3: Handshake Extension Flags

Initial Packet Sequence Number: 32 bits. The sequence number of the very first data packet to be sent.

Maximum Transmission Unit Size: 32 bits. This value is typically set to 1500, which is the default Maximum Transmission Unit (MTU) size for Ethernet, but can be less.

Maximum Flow Window Size: 32 bits. The value of this field is the maximum number of data packets allowed to be "in flight"

(i.e. the number of sent packets for which an ACK control packet has not yet been received).

Handshake Type: 32 bits. This field indicates the handshake packet type. The possible values are described in Table 4. For more details refer to Section 4.3.

Value	Handshake type
0xFFFFFFFD	DONE
0xFFFFFFFE	AGREEMENT
0xFFFFFFFF	CONCLUSION
0x00000000	WAVEHAND
0x00000001	INDUCTION

Table 4: Handshake Type

SRT Socket ID: 32 bits. This field holds the ID of the source SRT socket from which a handshake packet is issued.

SYN Cookie: 32 bits. Randomized value for processing a handshake. The value of this field is specified by the handshake message type. See Section 4.3.

Peer IP Address: 128 bits. IPv4 or IPv6 address of the packet's sender. The value consists of four 32-bit fields. In the case of IPv4 addresses, fields 2, 3 and 4 are filled with zeroes.

Extension Type: 16 bits. The value of this field is used to process an integrated handshake. Each extension can have a pair of request and response types.

Value	Extension Type	HS Extension Flag
1	SRT_CMD_HSREQ	HSREQ
2	SRT_CMD_HSRSP	HSREQ
3	SRT_CMD_KMREQ	KMREQ
4	SRT_CMD_KMRSP	KMREQ
5	SRT_CMD_SID	CONFIG
6	SRT_CMD_CONGESTION	CONFIG
7	SRT_CMD_FILTER	CONFIG
8	SRT_CMD_GROUP	CONFIG

Table 5: Handshake Extension Type values

Extension Length: 16 bits. The length of the Extension Contents field in four-byte blocks.

Extension Contents: variable length. The payload of the extension.

3.2.1.1. Handshake Extension Message

In a Handshake Extension, the value of the Extension Field of the handshake control packet is defined as 1 for a Handshake Extension request (SRT_CMD_HSREQ in Table 5), and 2 for a Handshake Extension response (SRT_CMD_HSRSP in Table 5).

The Extension Contents field of a Handshake Extension Message is structured as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          SRT Version                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           SRT Flags                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Receiver TSBPD Delay     |       Sender TSBPD Delay      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 6: Handshake Extension Message structure

SRT Version: 32 bits. SRT library version MUST be formed as major * 0x10000 + minor * 0x100 + patch.

SRT Flags: 32 bits. SRT configuration flags (see Section 3.2.1.1.1).

Receiver TSBPD Delay: 16 bits. Timestamp-Based Packet Delivery (TSBPD) Delay of the receiver. Refer to Section 4.5.

Sender TSBPD Delay: 16 bits. TSBPD of the sender. Refer to Section 4.5.

3.2.1.1.1. Handshake Extension Message Flags

Bitmask	Flag
0x00000001	TSBPDSND
0x00000002	TSBPDRCV
0x00000004	CRYPT
0x00000008	TLPKTDROP
0x00000010	PERIODICNAK
0x00000020	REXMITFLG
0x00000040	STREAM
0x00000080	PACKET_FILTER

Table 6: Handshake Extension Message Flags

* TSBPDSND flag defines if the TSBPD mechanism (Section 4.5) will be used for sending.

* TSBPDRCV flag defines if the TSBPD mechanism (Section 4.5) will be used for receiving.

* CRYPT flag MUST be set. It is a legacy flag that indicates the party understands KK field of the SRT Packet (Figure 3).

* TLPKTDROP flag should be set if too-late packet drop mechanism will be used during transmission. See Section 4.6.

* PERIODICNAK flag set indicates the peer will send periodic NAK packets. See Section 4.8.2.

* REXMITFLG flag MUST be set. It is a legacy flag that indicates the peer understands the R field of the SRT DATA Packet (Figure 3).

* STREAM flag identifies the transmission mode (Section 4.2) to be used in the connection. If the flag is set the buffer mode (Section 4.2.3) will be used. Otherwise, message mode (Section 4.2.1) is to be used.

* PACKET_FILTER flag indicates if the peer supports packet filter.

3.2.1.2. Key Material Extension Message

If an encrypted connection is being established, the Key Material (KM) is first transmitted as a Handshake Extension message. This extension is not supplied for unprotected connections. The purpose of the extension is to let peers exchange and negotiate encryption-related information to be used to encrypt and decrypt the payload of the stream.

The extension can be supplied with the Handshake Extension Type field set to either SRT_CMD_KMREQ or SRT_CMD_HSRSP (see Table 5 in Section 3.2.1). For more details refer to Section 4.3.

The KM message is placed in the Extension Contents. See Section 3.2.2 for the structure of the KM message.

3.2.1.3. Stream ID Extension Message

The Stream ID handshake extension message can be used to identify the stream content. The Stream ID value can be free-form, but there is also a recommended convention that can be used to achieve interoperability.

The Stream ID handshake extension message has SRT_CMD_SID extension type (see Table 5. The extension contents are a sequence of UTF-8 characters. The maximum allowed size of the StreamID extension is 512 bytes.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                           Stream ID                           |
                               ...
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 7: Stream ID Extension Message

The Extension Contents field holds a sequence of UTF-8 characters (see Figure 7). The maximum allowed size of the StreamID extension is 512 bytes. The actual size is determined by the Extension Length field (Figure 5), which defines the length in four byte blocks. If the actual payload is less than the declared length, the remaining bytes are set to zeros.

The content is stored as 32-bit little endian words.

3.2.1.4. Group Membership Extension

The Group Membership handshake extension is used to distinguish single SRT connections and bonded SRT connections (group connections).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Group ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Flags     |             Weight            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 8: Group Membership Extension Message

GroupID: 32 bits. The identifier of a group whose members include the sender socket that is making a connection. The target socket that should interpret it should belong to the corresponding group on its side (or should create one, if it doesn't exist).

Type: 8 bits. Group type, as per SRT_GTYPE_ enumeration.

* 0: undefined group type,

* 1: broadcast group type,

* 2: main/backup group type

* 3: balancing group type (reserved for future use)

* 4: multicast group type (reserved for future use)

Flags: 8 bits. Special flags mostly reserved for the future. See Figure 9.

Weight: 16 bits. Special value with interpretation depending on the Type field value.

* Not used with broadcast groups.

* Defines the link priority in backup groups.

* Not yet defined (reserved for future) for any other cases.

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|   (zero)   |M|
+-+-+-+-+-+-+-+-+
```

Figure 9: Group Membership Extension Flags

M: 1 bit. When set, defines synchronization on message numbers, otherwise transmission is synchronized on sequence numbers.

3.2.2. Key Material

The purpose of the Key Material Message is to let peers exchange encryption-related information to be used to encrypt and decrypt the payload of the stream.

This message can be supplied in two possible ways:

* as a Handshake Extension, see Section 3.2.1.2,

* in the Content Information Field of the User-Defined control packet (described below).

When the Key Material is transmitted as a control packet, the Control Type field of the SRT packet header is set to User-Defined Type (see Table 1), the Subtype field of the header is set to SRT_CMD_KMREQ for key-refresh request and SRT_CMD_KMRSP for key-refresh response (Table 5). The KM Refresh mechanism is described in Section 5.1.6.

The structure of the Key Material message is illustrated in Figure 10.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |S|  V  |   PT  |              Sign             |   Resv1   | KK|
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                              KEKI                             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |     Cipher    |      Auth     |       SE      |     Resv2     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |             Resv3             |     SLen/4    |     KLen/4    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                              Salt                             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                               |
   +                          Wrapped Key                          +
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 10: Key Material Message structure

S: 1 bit, value = {0}. This is a fixed-width field that is reserved for future usage.

Version (V): 3 bits, value = {1}. This is a fixed-width field that indicates the SRT version:

* 1: initial version

Packet Type (PT): 4 bits, value = {2}. This is a fixed-width field that indicates the Packet Type:

* 0: Reserved

* 1: Media Stream Message (MSmsg)

* 2: Keying Material Message (KMmsg)

* 7: Reserved to discriminate MPEG-TS packet (0x47=sync byte)

Sign: 16 bits, value = {0x2029}. This is a fixed-width field that contains the signature 'HAI' encoded as a PnP Vendor ID ([PNPID]) (in big-endian order)

Resv1: 6 bits, value = {0}. This is a fixed-width field reserved for flag extension or other usage.

Key-based Encryption (KK): 2 bits. This is a fixed-width field that indicates which SEKs (odd and/or even) are provided in the extension:

* 00b: no SEK is provided (invalid extension format)

* 01b: even key is provided

* 10b: odd key is provided

* 11b: both even and odd keys are provided

Key Encryption Key Index (KEKI): 32 bits, value = {0}. This is a fixed-width field for specifying the KEK index (big-endian order) was used to wrap (and optionally authenticate) the SEK(s). The value 0 is used to indicate the default key of the current stream. Other values are reserved for the possible use of a key management system in the future to retrieve a cryptographic context.

* 0: Default stream associated key (stream/system default)

* 1..255: Reserved for manually indexed keys

Cipher: 8 bits, value = {0..2}. This is a fixed-width field for specifying encryption cipher and mode:

* 0: None or KEKI indexed crypto context

* 2: AES-CTR [SP800-38A]

Authentication (Auth): 8 bits, value = {0}. This is a fixed-width field for specifying a message authentication code algorithm:

* 0: None or KEKI indexed crypto context

Stream Encapsulation (SE): 8 bits, value = {2}. This is a fixed-width field for describing the stream encapsulation:

* 0: Unspecified or KEKI indexed crypto context

* 1: MPEG-TS/UDP

* 2: MPEG-TS/SRT

Resv2: 8 bits, value = {0}. This is a fixed-width field reserved for future use.

Resv3: 16 bits, value = {0}. This is a fixed-width field reserved for future use.

SLen/4: 8 bits, value = {4}. This is a fixed-width field for specifying salt length SLen in bytes divided by 4. Can be zero if no salt/IV present. The only valid length of salt defined is 128 bits.

KLen/4: 8 bits, value = {4,6,8}. This is a fixed-width field for specifying SEK length in bytes divided by 4. Size of one key even if two keys present. MUST match the key size specified in the Encryption Field of the handshake packet Table 2.

Salt (SLen): SLen * 8 bits, value = { }. This is a variable-width field that complements the keying material by specifying a salt key.

Wrap: (64 + n * KLen * 8) bits, value = { }. This is a variable-width field for specifying Wrapped key(s), where n = (KK + 1)/2 and the size of the wrap field is ((n * KLen) + 8) bytes.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                  Integrity Check Vector (ICV)                 +
|                                                               |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                              xSEK                             |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                              oSEK                             |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure 11: Unwrapped key structure

ICV: 64 bits. 64-bit Integrity Check Vector(AES key wrap integrity). This field is used to detect if the keys were unwrapped properly. If the KEK in hand is invalid, validation fails and unwrapped keys are discarded.

xSEK: variable width. This field identifies an odd or even SEK. If only one key is present, the bit set in the KK field tells which SEK is provided. If both keys are present, then this field is eSEK (even key) and it is followed by odd key oSEK. The length of this field is calculated as KLen * 8.

oSEK: variable width. This field with the odd key is present only when the message carries the two SEKs (identified by he KK field).

3.2.3. Keep-Alive

Keep-alive control packets are sent after a certain timeout from the last time any packet (Control or Data) was sent. The purpose of this control packet is to notify the peer to keep the connection open when no data exchange is taking place.

The default timeout for a keep-alive packet to be sent is 1 second.

An SRT keep-alive packet is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+- SRT Header +-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|         Control Type        |           Reserved            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Type-specific Information                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination Socket ID                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 12: Keep-Alive control packet

Packet Type: 1 bit, value = 1. The packet type value of a keep-alive control packet is "1".

Control Type: 15 bits, value = KEEPALIVE{0x0001}. The control type value of a keep-alive control packet is "1".

Reserved: 16 bits, value = 0. This is a fixed-width field reserved for future use.

Type-specific Information. This field is reserved for future definition.

Timestamp: 32 bits. See Section 3.

Destination Socket ID: 32 bits. See Section 3.

Keep-alive controls packet do not contain Control Information Field (CIF).

3.2.4. ACK (Acknowledgment)

Acknowledgment control packets are used to provide delivery status of data packets. By acknowled reception of data packets up to the acknowledged packet sequence number the receiver notifies the sender that all prior packets were received or, in case of live transmission mode (Section 4.2.2), preceeeding missing packets if any were dropped as too late to be delivered.

ACK packets may also carry some additional information from the receiver like RTT, bandwidth, receiving speed, etc. The CIF portion of the ACK control packet is expanded as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+- SRT Header +-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|        Control Type         |           Reserved            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgement Number                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination Socket ID                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- CIF -+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Last Acknowledged Packet Sequence Number           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              RTT                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          RTT Variance                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Available Buffer Size                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Packets Receiving Rate                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Estimated Link Capacity                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Receiving Rate                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 13: ACK control packet

Packet Type: 1 bit, value = 1. The packet type value of an ACK control packet is "1".

Control Type: 15 bits, value = ACK{0x0002}. The control type value of an ACK control packet is "2".

Reserved: 16 bits, value = 0. This is a fixed-width field reserved for future use.

Acknowledgement Number: 32 bits. This field contains the sequential number of the full acknowledgment packet starting from 1.

Timestamp: 32 bits. See Section 3.

Destination Socket ID: 32 bits. See Section 3.

Last Acknowledged Packet Sequence Number: 32 bits. This field contains the sequence number of the last data packet being acknowledged plus one. In other words, if it the sequence number of the first unacknowledged packet.

RTT: 32 bits. RTT value, in microseconds, estimated by the receiver based on the previous ACK-ACKACK packet exchange.

RTT Variance: 32 bits. The variance of the RTT estimation, in microseconds.

Available Buffer Size: 32 bits. Available size of the receiver's buffer, in packets.

Packets Receiving Rate: 32 bits. The rate at which packets are being received, in packets per second.

Estimated Link Capacity: 32 bits. Estimated bandwidth of the link, in packets per second.

Receiving Rate: 32 bits. Estimated receiving rate, in bytes per second.

There are several types of ACK packets:

* A Full ACK control packet is sent every 10 ms and has all the fields of Figure 13.

* A Lite ACK control packet includes only the Last Acknowledged Packet Sequence Number field. The Type-specific Information field should be set to 0.

* A Small ACK includes the fields up to and including the Available Buffer Size field. The Type-specific Information field should be set to 0.

The sender only acknowledges the receipt of Full ACK packets (see ACKACK Section Section 3.2.7).

The Lite ACK and Small ACK packets are used in cases when the receiver should acknowledge received data packets more often than every 10 ms. This is usually needed at high data rates. It is up to the receiver to decide the condition and the type of ACK packet to send (Lite or Small). The recommendation is to send a Lite ACK for every 64 packets received.

3.2.5. NAK (Loss Report)

Negative acknowledgment (NAK) control packets are used to signal failed data packet deliveries. The receiver notifies the sender about lost data packets by sending a NAK packet that contains a list of sequence numbers for those lost packets.

An SRT NAK packet is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+- SRT Header +-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|         Control Type        |            Reserved             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Type-specific Information                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination Socket ID                       |
+-+-+-+-+-+-+-+-+-+-+-+-+- CIF (Loss List) -+-+-+-+-+-+-+-+-+-+-+-+
|0|                 Lost packet sequence number                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|         Range of lost packets from sequence number            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|                    Up to sequence number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|                 Lost packet sequence number                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 14: NAK control packet

Packet Type: 1 bit, value = 1. The packet type value of a NAK control packet is "1".

Control Type: 15 bits, value = NAK{0x0003}. The control type value of a NAK control packet is "3".

Reserved: 16 bits, value = 0. This is a fixed-width field reserved for future use.

Type-specific Information: 32 bits. This field is reserved for future definition.

Timestamp: 32 bits. See Section 3.

Destination Socket ID: 32 bits. See Section 3.

Control Information Field (CIF). A single value or a range of lost packets sequence numbers. See packet sequence number coding in Appendix A.

3.2.6. Shutdown

Shutdown control packets are used to initiate the closing of an SRT connection.

An SRT shutdown control packet is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+- SRT Header +-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|        Control Type         |           Reserved            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Type-specific Information                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination Socket ID                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 15: Shutdown control packet

Packet Type: 1 bit, value = 1. The packet type value of a shutdown control packet is "1".

Control Type: 15 bits, value = SHUTDOWN{0x0005}. The control type value of a shutdown control packet is "5".

Timestamp: 32 bits. See Section 3.

Destination Socket ID: 32 bits. See Section 3.

Type-specific Information. This field is reserved for future definition.

Shutdown control packets do not contain Control Information Field (CIF).

3.2.7. ACKACK

ACKACK control packets are sent to acknowledge the reception of a Full ACK, and are used in the calculation of RTT by the receiver.

An SRT ACKACK Control packet is formatted as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+- SRT Header +-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|         Control Type        |           Reserved            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Acknowledgement Number                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination Socket ID                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 16: ACKACK control packet

Packet Type: 1 bit, value = 1. The packet type value of an ACKACK control packet is "1".

Control Type: 15 bits, value = ACKACK{0x0006}. The control type value of an ACKACK control packet is "6".

Acknowledgement Number. This field contains the Acknowledgement Number of the full ACK packet the reception of which is being acknowledged by this ACKACK packet.

Timestamp: 32 bits. See Section 3.

Destination Socket ID: 32 bits. See Section 3.

ACKACK control packets do not contain Control Information Field (CIF).

4. SRT Data Transmission and Control

This section describes key concepts related to the handling of control and data packets during the transmission process.

After the handshake and exchange of capabilities is completed, packet data can be sent and received over the established connection. To fully utilize the features of low latency and error recovery provided by SRT, the sender and receiver must handle control packets, timers, and buffers for the connection as specified in this section.

4.1. Stream Multiplexing

Multiple SRT sockets may share the same UDP socket so that the packets received to this UDP socket will be correctly dispatched to those SRT sockets they are currently destined.

During the handshake, the parties exchange their SRT Socket IDs. These IDs are then used in the Destination Socket ID field of every control and data packet (see Section 3).

4.2. Data Transmission Modes

SRT has been mainly created for Live Streaming and therefore its main and default transmission mode is "live". SRT supports, however, the modes that the original UDT library supported, that is, buffer and message transmission.

4.2.1. Message Mode

When the STREAM flag of the handshake Extension Message Section 3.2.1.1 is set to 0, the protocol operates in Message mode, characterized as follows:

* Every packet has its own Packet Sequence Number.

* One or several consecutive SRT Data packets can form a message.

* All the packets belonging to the same message have a similar message number set in the Message Number field.

The first packet of a message has the first bit of the Packet Position Flags (Section 3.1) set to 1. The last packet of the message has the second bit of the Packet Position Flags set to 1. Thus, a PP equal to "11b" indicates a packet that forms the whole message. A PP equal to "00b" indicates a packet that belongs to the inner part of the message.

The concept of the message in SRT comes from UDT ([GHG04b]). In this mode a single sending instruction passes exactly one piece of data that has boundaries (a message). This message may span across multiple UDP packets (and multiple SRT data packets). The only size limitation is that it shall fit as a whole in the buffers of the sender and the receiver. Although internally all operations (e.g. ACK, NAK) on data packets are performed independently, an application must send and receive the whole message. Until the message is complete (all packets are received) the application will not be allowed to read it.

When the Order Flag of a Data packet is set to 1, this imposes a sequential reading order on messages. An Order Flag set to 0 allows an application to read messages that are already fully available, before any preceding messages that may have some packets missing.

4.2.2. Live Mode

Live mode is a special type of message mode where only data packets with their PP field set to "11b" are allowed.

Additionally Timestamp-Based Packet Delivery (TSBPD) (Section 4.5) and Too-Late Packet Drop (Section 4.6) mechanisms are used in this mode.

4.2.3. Buffer Mode

Buffer mode is negotiated during the Handshake by setting the STREAM flag of the handshake Extension Message Flags to 1.

In this mode consecutive packets form one continuous stream that can be read, with portions of any size.

4.3. Handshake Messages

SRT is a connection-oriented protocol. It embraces the concepts of "connection" and "session". The UDP system protocol is used by SRT for sending data and control packets.

An SRT connection is characterized by the fact that it is:

* first engaged by a handshake process;

* maintained as long as any packets are being exchanged in a timely manner;

* considered closed when a party receives the appropriate close command from its peer (connection closed by the foreign host), or when it receives no packets at all for some predefined time (connection broken on timeout).

SRT supports two connection configurations:

1. Caller-Listener, where one side waits for the other to initiate a connection

2. Rendezvous, where both sides attempt to initiate a connection

The handshake is performed between two parties: "Initiator" and "Responder":

* Initiator starts the extended SRT handshake process and sends appropriate SRT extended handshake requests.

* Responder expects the SRT extended handshake requests to be sent by the Initiator and sends SRT extended handshake responses back.

There are two basic types of SRT handshake extensions that are exchanged in the handshake:

* Handshake Extension Message exchanges the basic SRT information;

* Key Material Exchange exchanges the wrapped stream encryption key (used only if encryption is requested).

* Stream ID extension exchanges some stream-specific information that can be used by the application to identify the incoming stream connection.

The Initiator and Responder roles are assigned depending on the connection mode.

For Caller-Listener connections: the Caller is the Initiator, the Listener is the Responder. For Rendezvous connections: the Initiator and Responder roles are assigned based on the initial data interchange during the handshake.

The Handshake Type field in the Handshake Structure (see Figure 5) indicates the handshake message type.

Caller-Listener handshake exchange has the following order of Handshake Types:

1. Caller to Listener: INDUCTION

2. Listener to Caller: INDUCTION (reports cookie)

3. Caller to Listener: CONCLUSION (uses previously returned cookie)

4. Listener to Caller: CONCLUSION (confirms connection established)

Rendezvous handshake exchange has the following order of Handshake Types:

1. After starting the connection: WAVEAHAND.

2. After receiving the above message from the peer: CONCLUSION.

3. After receiving the above message from the peer: AGREEMENT.

When a connection process has failed before either party can send the CONCLUSION handshake, the Handshake Type field will contain the appropriate error value for the rejected connection. See the list of error codes in Table 7.

Code	Error	Description
1000	REJ_UNKNOWN	Unknown reason
1001	REJ_SYSTEM	System function error
1002	REJ_PEER	Rejected by peer
1003	REJ_RESOURCE	Resource allocation problem
1004	REJ_ROGUE	incorrect data in handshake
1005	REJ_BACKLOG	listener's backlog exceeded
1006	REJ_IPE	internal program error
1007	REJ_CLOSE	socket is closing
1008	REJ_VERSION	peer is older version than agent's min
1009	REJ_RDVCOOKIE	rendezvous cookie collision
1010	REJ_BADSECRET	wrong password
1011	REJ_UNSECURE	password required or unexpected
1012	REJ_MESSAGEAPI	Stream flag collision
1013	REJ_CONGESTION	incompatible congestion-controller type
1014	REJ_FILTER	incompatible packet filter
1015	REJ_GROUP	incompatible group

Table 7: Handshake Rejection Reason Codes

The specification of the cipher family and block size is decided by the data Sender. When the transmission is bidirectional, this value MUST be agreed upon at the outset because when both are set the Responder wins. For Caller-Listener connections it is reasonable to set this value on the Listener only. In the case of Rendezvous the only reasonable approach is to decide upon the correct value from the different sources and to set it on both parties (note that *AES-128* is the default).

4.3.1. Caller-Listener Handshake

This section describes the handshaking process where a Listener is waiting for an incoming Handshake request on a bound UDP port from a Caller. The process has two phases: induction and conclusion.

4.3.1.1. The Induction Phase

The INDUCTION phase serves only to set a cookie on the Listener so that it doesn't allocate resources, thus mitigating a potential DoS attack that might be perpetrated by flooding the Listener with handshake commands.

The Caller begins by sending the INDUCTION handshake, which contains the following (significant) fields:

* Version: MUST always be 4

* Encryption Field: 0

* Extension Field: 2

* Handshake Type: INDUCTION

* SRT Socket ID: SRT Socket ID of the Caller

* SYN Cookie: 0

The Destination Socket ID of the SRT packet header in this message is 0, which is interpreted as a connection request.

The handshake version number is set to 4 in this initial handshake. This is due to the initial design of SRT that was to be compliant with the UDT protocol ([GHG04b]) on which it is based.

The Listener responds with the following:

* Version: 5

* Encryption Field: Advertised cipher family and block size.

* Extension Field: SRT magic code 0x4A17

* Handshake Type: INDUCTION

* SRT Socket ID: Socket ID of the Listener

* SYN Cookie: a cookie that is crafted based on host, port and current time with 1 minute accuracy to avoid SYN flooding attack [RFC4987]

At this point the Listener still does not know if the Caller is SRT or UDT, and it responds with the same set of values regardless of whether the Caller is SRT or UDT.

If the party is SRT, it does interpret the values in Version and Extension Field. If it receives the value 5 in Version, it understands that it comes from an SRT party, so it knows that it should prepare the proper handshake messages phase. It also checks the following:

* whether the Extension Flags contains the magic value 0x4A17; otherwise the connection is rejected. This is a contingency for the case where someone who, in an attempt to extend UDT independently, increases the Version value to 5 and tries to test it against SRT.

* whether the Encryption Flags contain a non-zero value, which is interpreted as an advertised cipher family and block size.

A legacy UDT party completely ignores the values reported in Version and Handshake Type. It is, however, interested in the SYN Cookie value, as this must be passed to the next phase. It does interpret these fields, but only in the "conclusion" message.

4.3.1.2. The Conclusion Phase

Once the Caller gets the SYN cookie from the Listener, it sends the CONCLUSION handshake to the Listener.

The following values are set by the compliant caller:

* Version: 5

* Handshake Type: CONCLUSION

* SRT Socket ID: Socket ID of the Caller

* SYN Cookie: the cookie previously received in the induction phase

The Destination Socket ID in this message is the socket ID that was previously received in the induction phase in the SRT Socket ID field of the handshake structure.

* Encryption Flags: advertised cipher family and block size.

* Extension Flags: A set of flags that define the extensions provided in the handshake.

The Listener responds with the same values shown above, without the cookie (which is not needed here), as well as the extensions for HS Version 5 (which will probably be exactly the same).

There is not any "negotiation" here. If the values passed in the handshake are in any way not acceptable by the other side, the connection will be rejected. The only case when the Listener can have precedence over the Caller is the advertised Cipher Family and Block Size (Table 2) in the Encryption Field of the Handshake.

The value for latency is always agreed to be the greater of those reported by each party.

4.3.2. Rendezvous Handshake

The Rendezvous process uses a state machine. It is slightly different from UDT Rendezvous handshake [GHG04b], although it is still based on the same message request types.

Both parties start with WAVEAHAND and use the Version value of 5. Legacy Version 4 clients do not look at the Version value, whereas Version 5 clients can detect version 5. The parties only continue with the Version 5 Rendezvous process when Version is set to 5 for both. Otherwise the process continues exclusively according to Version 4 rules [GHG04b].

With Version 5 Rendezvous, both parties create a cookie for a process called the "cookie contest". This is necessary for the assignment of Initiator and Responder roles. Each party generates a cookie value (a 32-bit number) based on the host, port, and current time with 1 minute accuracy. This value is scrambled using an MD5 sum calculation. The cookie values are then compared with one another.

Since it is impossible to have two sockets on the same machine bound to the same NIC and port and operating independently, it is virtually impossible that the parties will generate identical cookies. However, this situation may occur if an application tries to "connect to itself" - that is, either connects to a local IP address, when the socket is bound to INADDR_ANY, or to the same IP address to which the socket was bound. If the cookies are identical (for any reason), the connection will not be made until new, unique cookies are generated (after a delay of up to one minute). In the case of an application "connecting to itself", the cookies will always be identical, and so the connection will never be established.

When one party's cookie value is greater than its peer's, it wins the cookie contest and becomes Initiator (the other party becomes the Responder).

At this point there are two possible "handshake flows": serial and parallel.

4.3.2.1. Serial Handshake Flow

In the serial handshake flow, one party is always first, and the other follows. That is, while both parties are repeatedly sending WAVEAHAND messages, at some point one party - let's say Alice - will find she has received a WAVEAHAND message before she can send her next one, so she sends a CONCLUSION message in response. Meantime, Bob (Alice's peer) has missed Alice's WAVEAHAND messages, so that Alice's CONCLUSION is the first message Bob has received from her.

This process can be described easily as a series of exchanges between the first and following parties (Alice and Bob, respectively):

1. Initially, both parties are in the waving state. Alice sends a handshake message to Bob:

    * Version: 5

    * Type: Extension field: 0, Encryption field: advertised "PBKEYLEN".

    * Handshake Type: WAVEAHAND

    * SRT Socket ID: Alice's socket ID

    * SYN Cookie: Created based on host/port and current time.

While Alice does not yet know if she is sending this message to a Version 4 or Version 5 peer, the values from these fields would not be interpreted by the Version 4 peer when the Handshake Type is WAVEAHAND.

1. Bob receives Alice’s WAVEAHAND message, switches to the "attention" state. Since Bob now knows Alice’s cookie, he performs a "cookie contest" (compares both cookie values). If Bob’s cookie is greater than Alice’s, he will become the Initiator. Otherwise, he will become the Responder.

The resolution of the Handshake Role (Initiator or Responder) is essential for further processing.

Then Bob responds:

* Version: 5

* Extension field: appropriate flags if Initiator, otherwise 0

* Encryption field: advertised PBKEYLEN

* Handshake Type: CONCLUSION

If Bob is the Initiator and encryption is on, he will use either his own cipher family and block size or the one received from Alice (if she has advertised those values).

1. Alice receives Bob’s CONCLUSION message. While at this point she also performs the "cookie contest", the outcome will be the same. She switches to the "fine" state, and sends:

    * Version: 5

    * Appropriate extension flags and encryption flags

    * Handshake Type: CONCLUSION

Both parties always send extension flags at this point, which will contain HSREQ if the message comes from an Initiator, or HSRSP if it comes from a Responder. If the Initiator has received a previous message from the Responder containing an advertised cipher family and block size in the encryption flags field, it will be used as the key length for key generation sent next in the KMREQ extension.

1. Bob receives Alice’s CONCLUSION message, and then does one of the following (depending on Bob’s role):

    * If Bob is the Initiator (Alice’s message contains HSRSP), he:

        - switches to the "connected" state

    - sends Alice a message with Handshake Type AGREEMENT, but containing no SRT extensions (Extension Flags field should be 0)

* If Bob is the Responder (Alice's message contains HSREQ), he:

    - switches to "initiated" state

    - sends Alice a message with Handshake Type CONCLUSION that also contains extensions with HSRSP

        o awaits a confirmation from Alice that she is also connected (preferably by AGREEMENT message)

2. Alice receives the above message, enters into the "connected" state, and then does one of the following (depending on Alice's role):

    * If Alice is the Initiator (received CONCLUSION with HSRSP), she sends Bob a message with Handshake Type = AGREEMENT.

    * If Alice is the Responder, the received message has Handshake Type AGREEMENT and in response she does nothing.

3. At this point, if Bob was Initiator, he is connected already. If he was a Responder, he should receive the above AGREEMENT message, after which he switches to the "connected" state. In the case where the UDP packet with the agreement message gets lost, Bob will still enter the "connected" state once he receives anything else from Alice. If Bob is going to send, however, he has to continue sending the same CONCLUSION until he gets the confirmation from Alice.

4.3.2.2. Parallel Handshake Flow

The chances of the parallel handshake flow are very low, but still it may occur if the handshake messages with WAVEAHAND are sent and received by both peers at precisely the same time.

The resulting flow is very much like Bob's behaviour in the serial handshake flow, but for both parties. Alice and Bob will go through the same state transitions:

Waving -> Attention -> Initiated -> Connected

In the Attention state they know each other's cookies, so they can assign roles. In contrast to serial flows, which are mostly based on request-response cycles, here everything happens completely asynchronously: the state switches upon reception of a particular handshake message with appropriate contents (the Initiator MUST attach the HSREQ extension, and Responder MUST attach the "HSRSP" extension).

Here's how the parallel handshake flow works, based on roles:

Initiator:

1. Waving

    * Receives WAVEAHAND message

    * Switches to Attention

    * Sends CONCLUSION + HSREQ

2. Attention

    * Receives CONCLUSION message, which:

        - contains no extensions:

            o switches to Initiated, still sends CONCLUSION + HSREQ

        - contains "HSRSP" extension:

            o switches to Connected, sends AGREEMENT

3. Initiated

    * Receives CONCLUSION message, which:

        - Contains no extensions:

            o REMAINS IN THIS STATE, still sends CONCLUSION + HSREQ

        - contains "HSRSP" extension:

            o switches to Connected, sends AGREEMENT

4. Connected

    * May receive CONCLUSION and respond with AGREEMENT, but normally by now it should already have received payload packets.

Responder:

1. Waving

   * Receives WAVEAHAND message

   * Switches to Attention

   * Sends CONCLUSION message (with no extensions)

2. Attention

   * Receives CONCLUSION message with HSREQ. This message might contain no extensions, in which case the party shall simply send the empty CONCLUSION message, as before, and remain in this state.

   * Switches to Initiated and sends CONCLUSION message with HSRSP

3. Initiated

   * Receives:

     - CONCLUSION message with HSREQ

       o responds with CONCLUSION with HSRSP and remains in this state

     - AGREEMENT message

       o responds with AGREEMENT and switches to Connected

     - Payload packet

       o responds with AGREEMENT and switches to Connected

4. Connected

   * Is not expecting to receive any handshake messages anymore. The AGREEMENT message is always sent only once or per every final CONCLUSION message.

Note that any of these packets may be missing, and the sending party will never become aware. The missing packet problem is resolved this way:

1. If the Responder misses the CONCLUSION + HSREQ message, it simply continues sending empty CONCLUSION messages. Only upon reception of CONCLUSION + HSREQ does it respond with CONCLUSION + HSRSP.

2. If the Initiator misses the CONCLUSION + HSRSP response from the Responder, it continues sending CONCLUSION + HSREQ. The Responder MUST always respond with CONCLUSION + HSRSP when the Initiator sends CONCLUSION + HSREQ, even if it has already received and interpreted it.

3. When the Initiator switches to the Connected state it responds with a AGREEMENT message, which may be missed by the Responder. Nonetheless, the Initiator may start sending data packets because it considers itself connected - it does not know that the Responder has not yet switched to the Connected state. Therefore it is exceptionally allowed that when the Responder is in the Initiated state and receives a data packet (or any control packet that is normally sent only between connected parties) over this connection, it may switch to the Connected state just as if it had received a AGREEMENT message.

4. If the the Initiator has already switched to the Connected state it will not bother the Responder with any more handshake messages. But the Responder may be completely unaware of that (having missed the AGREEMENT message from the Initiator). Therefore it does not exit the connecting state, which means that it continues sending CONCLUSION + HSRSP messages until it receives any packet that will make it switch to the Connected state (normally AGREEMENT). Only then does it exit the connecting state and the application can start transmission.

4.4. SRT Buffer Latency

The SRT sender and receiver have buffers to store packets.

On the sender, latency is the time that SRT holds a packet to give it a chance to be delivered successfully while maintaining the rate of the sender at the receiver. If an acknowledgment (ACK) is missing or late for more than the configured latency, the packet is dropped from the sender buffer. A packet can be retransmitted as long as it remains in the buffer for the duration of the latency window. On the receiver, packets are delivered to an application from a buffer after the latency interval has passed. This helps to recover from potential packet losses. See Section 4.5, Section 4.6 for details.

Latency is a value, in milliseconds, that can cover the time to transmit hundreds or even thousands of packets at high bitrate. Latency can be thought of as a window that slides over time, during which a number of activities take place, such as the reporting of acknowledged packets (ACKs) (Section 4.8.1) and unacknowledged packets (NAKs)(Section 4.8.2).

Latency is configured through the exchange of capabilities during the extended handshake process between initiator and responder. The Handshake Extension Message (Section 3.2.1.1) has TSBPD delay information, in milliseconds, from the SRT receiver and sender. The latency for a connection will be established as the maximum value of latencies proposed by the initiator and responder.

4.5. Timestamp-Based Packet Delivery

The goal of the SRT Timestamp-Based Packet Delivery (TSBPD) mechanism is to reproduce the output of the sending application (e.g., encoder) at the input of the receiving application (e.g., decoder) in live data transmission mode (see Section 4.2). It attempts to reproduce the timing of packets committed by the sending application to the SRT sender. This allows packets to be scheduled for delivery by the SRT receiver, making them ready to be read by the receiving application (see Figure 17).

The SRT receiver, using the timestamp of the SRT data packet header, delivers packets to a receiving application with a fixed minimum delay from the time the packet was scheduled for sending on the SRT sender side. Basically, the sender timestamp in the received packet is adjusted to the receiver’s local time (compensating for the time drift or different time zones) before releasing the packet to the application. Packets can be withheld by the SRT receiver for a configured receiver delay. A higher delay can accommodate a larger uniform packet drop rate, or a larger packet burst drop. Packets received after their "play time" are dropped if the Too-Late Packet Drop feature is enabled (see Section 4.6).

The packet timestamp, in microseconds, is relative to the SRT connection creation time. Packets are inserted based on the sequence number in the header field. The origin time, in microseconds, of the packet is already sampled when a packet is first submitted by the application to the SRT sender unless explicitly provided. The TSBPD feature uses this time to stamp the packet for first transmission and any subsequent retransmission. This timestamp and the configured SRT latency (Section 4.4) control the recovery buffer size and the instant that packets are delivered at the destination (the aforementioned "play time" which is decided by adding the timestamp to the configured latency).

It is worth mentioning that the use of the packet sending time to stamp the packets is inappropriate for the TSBPD feature, since a new time (current sending time) is used for retransmitted packets, putting them out of order when inserted at their proper place in the stream.

Figure 17 illustrates the key latency points during the packet transmission with the TSBPD feature enabled.

```
                 |  Sending  |              |                   |
                 |   Delay   |    ~RTT/2    |    SRT Latency    |
                 |<--------->|<------------>|<----------------->|
                 |           |              |                   |
                 |           |              |                   |
                 |           |              |                   |
       ___ Scheduled       Sent         Received           Scheduled
      /    for sending       |              |              for delivery
   Packet        |           |              |                   |
   State         |           |              |                   |
                 |           |              |                   |
                 |           |              |                   |
                 ----------------------------------------------------->
                                                                   Time
```

Figure 17: Key Latency Points during the Packet Transmission

The main packet states shown in Figure 17 are the following:

* "Scheduled for sending": the packet is committed by the sending application, stamped and ready to be sent;

* "Sent": the packet is passed to the UDP socket and sent;

* "Received": the packet is received and read from the UDP socket;

* "Scheduled for delivery": the packet is scheduled for the delivery and ready to be read by the receiving application.

It is worth noting that the round-trip time (RTT) of an SRT link may vary in time. However the actual end-to-end latency on the link becomes fixed and is approximately equal to (RTT_0/2 + SRT Latency) once the SRT handshake exchange happens, where RTT_0 is the actual value of the round-trip time during the SRT handshake exchange (the value of the round-trip time once the SRT connection has been established).

The value of sending delay depends on the hardware performance. Usually it is relatively small (several microseconds) in contrast to RTT_0/2 and SRT latency which are measured in milliseconds.

4.5.1. Packet Delivery Time

Packet delivery time is the moment, estimated by the receiver, when a packet should be delivered to the upstream application. The calculation of packet delivery time (PktTsbpdTime) is performed upon receiving a data packet according to the following formula:

PktTsbpdTime = TsbpdTimeBase + PKT_TIMESTAMP + TsbpdDelay + Drift

where

* TsbpdTimeBase is the time base that reflects the time difference between local clock of the receiver and the clock used by the sender to timestamp packets being sent (see Section 4.5.1.1);

* PKT_TIMESTAMP is the data packet timestamp, in microseconds;

* TsbpdDelay is the receiver's buffer delay (or receiver's buffer latency, or SRT Latency). This is the time, in milliseconds, that SRT holds a packet from the moment it has been received till the time it should be delivered to the upstream application;

* Drift is the time drift used to adjust the fluctuations between sender and receiver clock, in microseconds.

SRT Latency (TsbpdDelay) should be a buffer time large enough to cover the unexpectedly extended RTT time, and the time needed to retransmit the lost packet. The value of minimum TsbpdDelay is negotiated during the SRT handshake exchange and is equal to 120 milliseconds. The recommended value of TsbpdDelay is 3-4 times RTT.

It is worth noting that TsbpdDelay limits the number of packet retransmissions to a certain extent making impossible to retransmit packets endlessly. This is important for live data transmission.

4.5.1.1. TSBPD Time Base Calculation

The initial value of TSBPD time base (TsbpdTimeBase) is calculated at the moment of the second handshake request is received as follows:

TsbpdTimeBase = T_NOW - HSREQ_TIMESTAMP

where T_NOW is the current time according to the receiver clock; HSREQ_TIMESTAMP is the handshake packet timestamp, in microseconds.

The value of TsbpdTimeBase is approximately equal to the initial one-way delay of the link RTT_0/2, where RTT_0 is the actual value of the round-trip time during the SRT handshake exchange.

During the transmission process, the value of TSBPD time base may be adjusted in two cases:

1. During the TSBPD wrapping period. The TSBPD wrapping period happens every 01:11:35 hours. This time corresponds to the maximum timestamp value of a packet (MAX_TIMESTAMP). MAX_TIMESTAMP is equal to 0xFFFFFFFF, or the maximum value of 32-bit unsigned integer, in microseconds (Section 3). The TSBPD wrapping period starts 30 seconds before reaching the maximum timestamp value of a packet and ends once the packet with timestamp within (30, 60) seconds interval is delivered (read from the buffer). The updated value of TsbpdTimeBase will be recalculated as follows:

```
TsbpdTimeBase = TsbpdTimeBase + MAX_TIMESTAMP + 1
```

2. By drift tracer. See Section 4.7 for details.

4.6. Too-Late Packet Drop

The Too-Late Packet Drop (TLPKTDROP) mechanism allows the sender to drop packets that have no chance to be delivered in time, and allows the receiver to skip missing packets that have not been delivered in time. The timeout of dropping a packet is based on the TSBPD mechanism (see Section 4.5).

In the SRT, when Too-Late Packet Drop is enabled, and a packet timestamp is older than 125% of the SRT latency, it is considered too late to be delivered and may be dropped by the sender. However, the sender keeps packets for at least 1 second in case the SRT latency is not enough for a large RTT (that is, if 125% of the SRT latency is less than 1 second).

When enabled on the receiver, the receiver drops packets that have not been delivered or retransmitted in time, and delivers the subsequent packets to the application when it is their time to play.

In pseudo-code, the algorithm of reading from the receiver buffer is the following:

```
<CODE BEGINS>
pos = 0;  /* Current receiver buffer position */
i = 0;    /* Position of the next available in the receiver buffer
             packet relatively to the current buffer position pos */

while(True) {
    // Get the position i of the next available packet
    // in the receiver buffer
    i = next_avail();
    // Calculate packet delivery time PktTsbpdTime
    // for the next available packet
    PktTsbpdTime = delivery_time(i);

    if T_NOW < PktTsbpdTime:
        continue;

    Drop packets which buffer position number is less than i;

    Deliver packet with the buffer position i;

    pos = i + 1;
}
<CODE ENDS>
```

where T_NOW is the current time according to the receiver clock.

The TLPKTDROP mechanism can be turned off to always ensure a clean delivery. However, a lost packet can simply pause a delivery for some longer, potentially undefined time, and cause even worse tearing for the player. Setting higher SRT latency will help much more in the case when TLPKTDROP causes packet drops too often.

4.7. Drift Management

When the sender enters "connected" status it tells the application there is a socket interface that is transmitter-ready. At this point the application can start sending data packets. It adds packets to the SRT sender’s buffer at a certain input rate, from which they are transmitted to the receiver at scheduled times.

A synchronized time is required to keep proper sender/receiver buffer levels, taking into account the time zone and round-trip time (up to 2 seconds for satellite links). Considering addition/subtraction round-off, and possibly unsynchronized system times, an agreed-upon time base drifts by a few microseconds every minute. The drift may accumulate over many days to a point where the sender or receiver buffers will overflow or deplete, seriously affecting the quality of the video. SRT has a time management mechanism to compensate for this drift.

When a packet is received, SRT determines the difference between the time it was expected and its timestamp. The timestamp is calculated on the receiver side. The RTT tells the receiver how much time it was supposed to take. SRT maintains a reference between the time at the leading edge of the send buffer’s latency window and the corresponding time on the receiver (the present time). This allows to convert packet timestamp to the local receiver time. Based on this time, various events (packet delivery, etc.) can be scheduled.

The receiver samples time drift data and periodically calculates a packet timestamp correction factor, which is applied to each data packet received by adjusting the inter-packet interval. When a packet is received it is not given right away to the application. As time advances, the receiver knows the expected time for any missing or dropped packet, and can use this information to fill any "holes" in the receive queue with another packet (see Section 4.5).

It is worth noting that the period of sampling time drift data is based on a number of packets rather than time duration to ensure enough samples, independently of the media stream packet rate. The effect of network jitter on the estimated time drift is attenuated by using a large number of samples. The actual time drift being very slow (affecting a stream only after many hours) does not require a fast reaction.

The receiver uses local time to be able to schedule events -- to determine, for example, if it is time to deliver a certain packet right away. The timestamps in the packets themselves are just references to the beginning of the session. When a packet is received (with a timestamp from the sender), the receiver makes a reference to the beginning of the session to recalculate its timestamp. The start time is derived from the local time at the moment that the session is connected. A packet timestamp equals "now" minus "StartTime", where the latter is the point in time when the socket was created.

4.8. Acknowledgement and Lost Packet Handling

To enable the Automatic Repeat reQuest of data packet retransmissions, a sender stores all sent data packets in its buffer.

The SRT receiver periodically sends acknowledgments (ACKs) for the received data packets so that the SRT sender can remove the acknowledged packets from its buffer (Section 4.8.1). Once the acknowledged packets are removed, their retransmission is no longer possible and presumably not needed.

Upon receiving the full acknowledgment (ACK) control packet, the SRT sender should acknowledge its reception to the receiver by sending an ACKACK control packet with the sequence number of the full ACK packet being acknowledged.

The SRT receiver also sends NAK control packets to notify the sender about the missing packets (Section 4.8.2). The sending of a NAK packet can be triggered immediately after a gap in sequence numbers of data packets is detected. In addition, a Periodic NAK report mechanism can be used to send NAK reports periodically. The NAK packet in that case will list all the packets that the receiver considers being lost up to the moment the Periodic NAK report is sent.

Upon reception of the NAK packet, the SRT sender prioritizes retransmissions of lost packets over the regular data packets to be transmitted for the first time.

The retransmission of the missing packet is repeated until the receiver acknowledges its receipt, or if both peers agree to drop this packet (see Section 4.6).

4.8.1. Packet Acknowledgement (ACKs, ACKACKs)

At certain intervals (see below), the SRT receiver sends an acknowledgment (ACK) that causes the acknowledged packets to be removed from the SRT sender's buffer.

An ACK control packet contains the sequence number of the packet immediately following the latest in the list of received packets. Where no packet loss has occurred up to the packet with sequence number n, an ACK would include the sequence number (n + 1).

An ACK (from a receiver) will trigger the transmission of an ACKACK (by the sender), with almost no delay. The time it takes for an ACK to be sent and an ACKACK to be received is the RTT. The ACKACK tells the receiver to stop sending the ACK position because the sender already knows it. Otherwise, ACKs (with outdated information) would continue to be sent regularly. Similarly, if the sender does not receive an ACK, it does not stop transmitting.

There are two conditions for sending an acknowledgment. A full ACK is based on a timer of 10 milliseconds (the ACK period or synchronization time interval SYN). For high bitrate transmissions, a "light ACK" can be sent, which is an ACK for a sequence of packets. In a 10 milliseconds interval, there are often so many packets being sent and received that the ACK position on the sender does not advance quickly enough. To mitigate this, after 64 packets (even if the ACK period has not fully elapsed) the receiver sends a light ACK. A light ACK is a shorter ACK (SRT header and one 32-bit field). It does not trigger an ACKACK.

When a receiver encounters the situation where the next packet to be played was not successfully received from the sender, it will "skip" this packet (see Section 4.6) and send a fake ACK. To the sender, this fake ACK is a real ACK, and so it just behaves as if the packet had been received. This facilitates the synchronization between SRT sender and receiver. The fact that a packet was skipped remains unknown by the sender. Skipped packets are recorded in the statistics on the SRT receiver.

4.8.2. Packet Retransmission (NAKs)

The SRT receiver sends NAK control packets to notify the sender about the missing packets. The NAK packet sending can be triggered immediately after a gap in sequence numbers of data packets is detected.

Upon reception of the NAK packet, the SRT sender prioritizes retransmissions of lost packets over the regular data packets to be transmitted for the first time.

The SRT sender maintains a list of lost packets (loss list) that is built from NAK reports. When scheduling packet transmission, it looks to see if a packet in the loss list has priority and sends it if so. Otherwise, it sends the next packet scheduled for the first transmission list. Note that when a packet is transmitted, it stays in the buffer in case it is not received by the SRT receiver.

NAK packets are processed to fill in the loss list. As the latency window advances and packets are dropped from the sending queue, a check is performed to see if any of the dropped or resent packets are in the loss list, to determine if they can be removed from there as well so that they are not retransmitted unnecessarily.

There is a counter for the packets that are resent. If there is no ACK for a packet, it will stay in the loss list and can be resent more than once. Packets in the loss list are prioritized.

If packets in the loss list continue to block the send queue, at some point this will cause the send queue to fill. When the send queue is full, the sender will begin to drop packets without even sending them the first time. An encoder (or other application) may continue to provide packets, but there's no place for them, so they will end up being thrown away.

This condition where packets are unsent does not happen often. There is a maximum number of packets held in the send buffer based on the configured latency. Older packets that have no chance to be retransmitted and played in time are dropped, making room for newer real-time packets produced by the sending application. See Section 4.5, Section 4.6 for details.

In addition to the regular NAKs, the Periodic NAK report mechanism can be used to send NAK reports periodically. The NAK packet in that case will have all the packets that the receiver considers being lost at the time of sending the Periodic NAK report.

SRT Periodic NAK reports are sent with a period of (RTT + 4 * RTTVar) / 2 (so called NAKInterval), with a 20 milliseconds floor, where RTT and RTTVar are defined in section Section 4.10. A NAK control packet contains a compressed list of the lost packets. Therefore, only lost packets are retransmitted. By using NAKInterval for the NAK reports period, it may happen that lost packets are retransmitted more than once, but it helps maintain low latency in the case where NAK packets are lost.

An ACKACK tells the receiver to stop sending the ACK position because the sender already knows it. Otherwise, ACKs (with outdated information) would continue to be sent regularly.

An ACK serves as a ping, with a corresponding ACKACK pong, to measure RTT. The time it takes for an ACK to be sent and an ACKACK to be received is the RTT. Each ACK has a number. A corresponding ACKACK has that same number. The receiver keeps a list of all ACKs in a queue to match them. Unlike a full ACK, which contains the current RTT and several other values in the Control Information Field (CIF) (Section 3.2.4), a light ACK just contains the sequence number. All control messages are sent directly and processed upon reception, but ACKACK processing time is negligible (the time this takes is included in the round-trip time).

4.9. Bidirectional Transmission Queues

Once an SRT connection is established, both peers can send data packets simultaneously.

4.10. Round-Trip Time Estimation

Round-trip time (RTT) in SRT is estimated during the transmission of data packets based on a difference in time between an ACK packet is sent out and a corresponding ACKACK packet is received back by the SRT receiver.

An ACK sent by the receiver triggers an ACKACK from the sender with minimal processing delay. The ACKACK response is expected to arrive at the receiver roughly one RTT after the corresponding ACK was sent.

The SRT receiver records the time when an ACK is sent out. The ACK carries a unique sequence number (independent of the data packet sequence number). The corresponding ACKACK also carries the same sequence number. Upon receiving the ACKACK, SRT calculates the RTT by comparing the difference between the ACKACK arrival time and the ACK departure time. In the following formula, RTT is the current value that the receiver maintains and rtt is the recent value that was just calculated from an ACK/ACKACK pair:

RTT = RTT * 0.875 + rtt * 0.125

RTT variance RTTVar is obtained as follows:

RTTVar = RTTVar * 0.75 + abs(RTT - rtt) * 0.25

where abs() means an absolute value.

Both RTT and RTTVar are measured in microseconds. The initial value of RTT is 100 milliseconds, RTTVar is 50 milliseconds.

The smoothed RTT calculated by the receiver as well as the RTT variance RTTVar are sent with the next full acknowledgement packet (see Section 3.2.4). Note that the first ACK in an SRT session might contain an initial RTT value of 100 milliseconds, because the early calculations may not be precise.

The sender always gets the RTT from the receiver. It does not have an analog to the ACK/ACKACK mechanism, i.e. it can not send a message that guarantees an immediate return without processing. Upon an ACK reception, the SRT sender updates its own RTT and RTTVar values using the same formulas as above, in which case rtt is the most recent value it receives, i.e., carried by an incoming ACK.

Note that an SRT socket can both send and receive data packets. RTT and RTTVar are updated by the socket based on algorithms for the sender (using ACK packets) and for the receiver (using ACK-ACKACK pairs). When an SRT socket receives data, it updates its local RTT and RTTVar, which can be used for its own sender as well.

4.11. Congestion Control

SRT provides certain mechanisms for the sender to get some feedback from the receiving side through the ACK packets (Section 3.2.4). Every 10 ms the sender receives the latest values of RTT and RTT variance, Available Buffer Size, Packets Receiving Rate and Estimated Link Capacity. Upon reception of the NAK packet (Section 3.2.5) the sender can detect packet losses during the transmission. These mechanisms provide a solid background for various congestion control algorithms.

Given that SRT can operate in live and file transfer modes, there are two groups of congestion control algorithms possible.

For live transmission mode (Section 4.2.2) the congestion control algorithm does not need to control the sending pace of the data packets, as the sending timing is provided by the live input. Although certain limitations on the minimal inter-sending time of consecutive packets can be applied in order to avoid congestion during fluctuations of the source bitrate. Also it is allowed to drop those packets that can not be delivered in time.

For file transfer, any known File Congestion Control algorithms like CUBIC [RFC8312] and BBR [BBR] can apply, including the congestion control mechanism proposed in UDT [GHG04b], [GuAnAO]. The UDT congestion control relies on the available link capacity, packet loss reports (NAK) and packet acknowledgements (ACKs). It then slows down the output of packets as needed by adjusting the packet sending pace. In periods of congestion, it can block the main stream and focus on the lost packets.

5. Encryption

This section describes the encryption mechanism that protects the payload of SRT streams. Based on standard cryptographic algorithms, the mechanism allows an efficient stream cipher with a key establishment method.

5.1. Overview

SRT implements encryption using AES [AES] in counter mode (AES-CTR) [SP800-38A] with a short-lived key to encrypt and decrypt the media stream. The AES-CTR cipher is suitable for continuous stream encryption that permits decryption from any point, without access to start of the stream (random access), and for the same reason tolerates packet loss. It also offers strong confidentiality when the counter is managed properly.

5.1.1. Encryption Scope

SRT encrypts only the payload of SRT data packets (Section 3.1), while the header is left unencrypted. The unencrypted header contains the Packet Sequence Number field used to keep the synchronization of the cipher counter between the encrypting sender and the decrypting receiver. No constraints apply to the payload of SRT data packets as no padding of the payload is required by counter mode ciphers.

5.1.2. AES Counter

The counter for AES-CTR is the size of the cipher's block, i.e. 128 bits. It is derived from a 128-bit sequence consisting of

* a block counter in the least significant 16 bits, which counts the blocks in a packet,

* a packet index - based on the packet sequence number in the SRT header - in the next 32 bits,

* eighty zeroed bits.

The upper 112 bits of this sequence are XORed with an Initialization Vector (IV) to produce a unique counter for each crypto block. The IV is derived from the Salt provided in the Keying Material (Section 3.2.2):

IV = MSB(112, Salt): Most significant 112 bits of the salt.

5.1.3. Stream Encrypting Key (SEK)

The key used for AES-CTR encryption is called the "Stream Encrypting Key" (SEK). It is used for up to 2^25 packets with further rekeying. The short-lived SEK is generated by the sender using a pseudo-random number generator (PRNG), and transmitted within the stream, wrapped with another longer-term key, the Key Encrypting Key (KEK), using a known AES key wrap protocol.

For connection-oriented transport such as SRT, there is no need to periodically transmit the short-lived key since no additional party can join a stream in progress. The keying material is transmitted within the connection handshake packets, and for a short period when rekeying occurs.

5.1.4. Key Encrypting Key (KEK)

The Key Encrypting Key (KEK) is derived from a secret (passphrase) shared between the sender and the receiver. The KEK provides access to the Stream Encrypting Key, which in turn provides access to the protected payload of SRT data packets. The KEK has to be at least as long as the SEK.

The KEK is generated by a password-based key generation function (PBKDF2) [RFC2898], using the passphrase, a number of iterations (2048), a keyed-hash (HMAC-SHA1) [RFC2104], and a key length value (KLen). The PBKDF2 function hashes the passphrase to make a long string, by repetition or padding. The number of iterations is based on how much time can be given to the process without it becoming disruptive.

5.1.5. Key Material Exchange

The KEK is used to generate a wrap [RFC3394] that is put in a key material (KM) message by the initiator of a connection (i.e. caller in caller-listener handshake and initiator in the rendezvous handshake, see Section 4.3) to send to the responder (listener). The KM message contains the key length, the salt (one of the arguments provided to the PBKDF2 function), the protocol being used (e.g. AES-256) and the AES counter (which will eventually change, see Section 5.1.6).

On the other side, the responder attempts to decode the wrap to obtain the Stream Encrypting Key. In the protocol for the wrap there is a padding, which is a known template, so the responder knows from the KM that it has the right KEK to decode the SEK. The SEK (generated and transmitted by the initiator) is random, and cannot be known in advance. The KEK formula is calculated on both sides, with the difference that the responder gets the key length (KLen) from the initiator via the key material (KM). It is the initiator who decides on the configured length. The responder obtains it from the material sent by the initiator.

The responder returns the same KM message to show that it has the same information as the initiator, and that the encoded material will be decrypted. If the responder does not return this status, this means that it does not have the SEK. All incoming encrypted packets received by the responder will be lost (undecrypted). Even if they are transmitted successfully, the receiver will be unable to decrypt them, and so packets will be dropped. All data packets coming from responder will be unencrypted.

5.1.6. KM Refresh

The short lived SEK is regenerated for cryptographic reasons when a pre-determined number of packets has been encrypted. The KM refresh period is determined by the implementation. The receiver knows which SEK (odd or even) was used to encrypt the packet by means of the KK field of the SRT Data Packet (Section 3.1).

There are two variables used to determine the KM Refresh timing:

* KM Refresh Period specifies the number of packets to be sent before switching to the new SEK,

* KM Pre-Announcement Period specifies when a new key is announced in a number of packets before key switchover. The same value is used to determine when to decommission the old key after switchover.

The recommended KM Refresh Period is after 2^25 packets encrypted with the same SEK are sent. The recommended KM Pre-Announcement Period is 4000 packets (i.e. a new key is generated, wrapped, and sent at 2^25 minus 4000 packets; the old key is decommissioned at 2^25 plus 4000 packets).

Even and odd keys are alternated during transmission the following way. The packets with the earlier key #1 (let it be the odd key) will continue to be sent. The receiver will receive the new key #2 (even), then decrypt and unwrap it. The receiver will reply to the sender if it is able to understand. Once the sender gets to the 2^25th packet using the odd key (key #1), it will then start to send packets with the even key (key #2), knowing that the receiver has what it needs to decrypt them. This happens transparently, from one packet to the next. At 2^25 plus 4000 packets the first key will be decommissioned automatically.

Both keys live in parallel for two times the Pre-Announcement Period (e.g. 4000 packets before the key switch, and 4000 packets after). This is to allow for packet retransmission. It is possible for packets with the older key to arrive at the receiver a bit late. Each packet contains a description of which key it requires, so the receiver will still have the ability to decrypt it.

5.2. Encryption Process

5.2.1. Generating the Stream Encrypting Key

On the sending side SEK, Salt and KEK are generated the following way:

```
SEK  = PRNG(KLen)
Salt = PRNG(128)
KEK = PBKDF2(passphrase, LSB(64,Salt), Iter, Klen)
```

where

* PBKDF2 is the PKCS#5 Password Based Key Derivation Function [RFC2898],

* passphrase is the pre-shared passphrase,

* Salt is the field of the KM message,

* LSB(n, v) is the function taking n least significant bits of v,

* Iter=2048 defines the number of iterations for PBKDF2,

* KLen is the field of the KM message.

```
Wrap = AESkw(KEK, SEK)
```

where AESkw(KEK, SEK) is the key wrapping function [RFC3394].

5.2.2. Encrypting the Payload

The encryption of the payload of the SRT DATA packet is done with AES-CTR

```
EncryptedPayload = AES_CTR_Encrypt(SEK, IV, UnencryptedPayload)
```

where the Initialization Vector is derived as

```
IV = (MSB(112, Salt) << 2) XOR (PktSeqNo)
```

* PktSeqNo is the value of the Packet Sequence Number field of the SRT data packet.

5.3. Decryption Process

5.3.1. Restoring the Stream Encrypting Key

For the receiver to be able to decrypt the incoming stream it has to know the stream encrypting key (SEK) used by the sender. The receiver must know the passphrase used by the sender. The remaining information can be extracted from the Keying Material message.

The Keying Material message contains the AES-wrapped [RFC3394] SEK used by the encoder. The Key-Encryption Key (KEK) required to unwrap the SEK is calculated as:

KEK = PBKDF2(passphrase, LSB(64,Salt), Iter, KLen)

where

* PBKDF2 is the PKCS#5 Password Based Key Derivation Function [RFC2898],

* passphrase is the pre-shared passphrase,

* Salt is the field of the KM message,

* LSB(n, v) is the function taking n least significant bits of v,

* Iter=2048 defines the number of iterations for PBKDF2,

* KLen is the field of the KM message.

SEK = AESkuw(KEK, Wrap)

where AESkuw(KEK, Wrap) is the key unwrapping function.

5.3.2. Decrypting the Payload

The decryption of the payload of the SRT data packet is done with AES-CTR

DecryptedPayload = AES_CTR_Encrypt(SEK, IV, EncryptedPayload)

where the Initialization Vector is derived as

IV = (MSB(112, Salt) << 2) XOR (PktSeqNo)

* PktSeqNo is the value of the Packet Sequence Number field of the SRT data packet.

6. Security Considerations

SRT supports confidentiality of user data using stream ciphering based on AES. Session keys for ciphering are delivered through control packets during handshake, with the protection by Key Encryption Key, which is generated by a sender and receiver with pre-shared secret such as passphrase. As in UDT, careful uses of SYN Cookies may help to deter denial of service attacks. Appropriate security policy including key size, key refresh period, as well as passphrase should be managed by security officers, which is out of scope of the present document.

7. IANA Considerations

This document makes no requests of the IANA.

Contributors

This specification is heavily based on the SRT Protocol Technical Overview [SRTTO] written by Jean Dube and Steve Matthews.

In alphabetical order, the contributors to the pre-IETF SRT project and specification at Haivision are: Marc Cymontkowski, Roman Diouskine, Jean Dube, Mikolaj Malecki, Steve Matthews, Maria Sharabayko, Maxim Sharabayko, Adam Yellen.

The contributors to this specification at SK Telecom are Jeongseok Kim and Joonwoong Kim.

We cannot list all the contributors to the open-sourced implementation of SRT on GitHub. But we appreciate the help, contribution, integrations and feedback of the SRT and SRT Alliances community.

Acknowledgments

The basis of the SRT protocol and its implementation was the UDP-based Data Transfer Protocol [GHG04b]. The authors thank Yunhong Gu and Robert Grossman, the authors of the UDP-based Data Transfer Protocol [GHG04b].

TODO acknowledge.

References

Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/info/rfc768>.

[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/info/rfc2119>.

Informative References

[AES] National Institute of Standards and Technology, "FIPS Pub 197: Advanced Encryption Standard (AES)", November 2001, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[AV1] Rivaz, P.d. and J. Haughton, "AV1 Bitstream & Decoding Process Specification", September 2020, <https://aomediacodec.github.io/av1-spec/av1-spec.pdf>.

[BBR] Cardwell, N., Cheng, Y., Gunn, C.S., Yeganeh, S.H., and V. Jacobson, "BBR: Congestion-Based Congestion Control", October 2016.

[GHG04b] Gu, Y., Hong, X., and R.L. Grossman, "Experiences in Design and Implementation of a High Performance Transport Protocol", DOI 10.1109/SC.2004.24, December 2004, <https://doi.org/10.1109/SC.2004.24>.

[GuAnAO] Gu, Y., Hong, X., and R.L. Grossman, "An Analysis of AIMD Algorithm with Decreasing Increases", October 2004.

[H.265] International Telecommunications Union, "H.265 : High efficiency video coding", ITU-T Recommendation H.265, 2019.

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-29, 9 June 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-http-29.txt>.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-29, 9 June 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-29.txt>.

[ISO13818-1]
ISO, "Information technology -- Generic coding of moving pictures and associated audio information: Systems", ISO/IEC 13818-1, September 2020.

[ISO23009] ISO, "Information technology -- Dynamic adaptive streaming over HTTP (DASH)", ISO/IEC 23009:2019, September 2020.

[PNPID] "PNP ID AND ACPI ID REGISTRY", September 2020, <https://uefi.org/PNP_ACPI_Registry>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <https://www.rfc-editor.org/info/rfc2898>.

[RFC3031] Rosen, E., Viswanathan, A., and R. Callon, "Multiprotocol Label Switching Architecture", RFC 3031, DOI 10.17487/RFC3031, January 2001, <https://www.rfc-editor.org/info/rfc3031>.

[RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, DOI 10.17487/RFC3394, September 2002, <https://www.rfc-editor.org/info/rfc3394>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <https://www.rfc-editor.org/info/rfc4987>.

[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/info/rfc8174>.

[RFC8216] Pantos, R., Ed. and W. May, "HTTP Live Streaming", RFC 8216, DOI 10.17487/RFC8216, August 2017, <https://www.rfc-editor.org/info/rfc8216>.

[RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", RFC 8312, DOI 10.17487/RFC8312, February 2018, <https://www.rfc-editor.org/info/rfc8312>.

[RTMP] "Real-Time Messaging Protocol", September 2020, <https://www.adobe.com/devnet/rtmp.html>.

[SP800-38A] Dworkin, M., "Recommendation for Block Cipher Modes of Operation", December 2001.

[SRTSRC] "SRT fully functional reference implementation", September 2020, <https://github.com/Haivision/srt>.

[SRTTO] Dube, J. and S. Matthews, "SRT Protocol Technical Overview", December 2019.

[VP9] WebM, "VP9 Video Codec", September 2020, <https://www.webmproject.org/vp9>.

Appendix A. Packet Sequence List Coding

For any single packet sequence number, it uses the original sequence number in the field. The first bit MUST start with "0".

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|                   Sequence Number                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 18: Single sequence numbers coding

For any consecutive packet sequence numbers that the difference between the last and first is more than 1, only record the first (a) and the the last (b) sequence numbers in the list field, and modify the the first bit of a to "1".

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|                   Sequence Number a (first)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|                   Sequence Number b (last)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 19: Range of sequence numbers coding

Appendix B. SRT Access Control

One type of information that can be interchanged when a connection is being established in SRT is the Stream ID, which can be used in a caller-listener connection layout. This is a string of maximum 512 characters set on the caller side. It can be retrieved at the listener side on the newly accepted connection.

SRT listener can notify an upstream application about the connection attempt when a HS conclusion arrives, exposing the contents of the Stream ID extension message. Based on this information, the application can accept or reject the connection, select the desired data stream, or set an appropriate passphrase for the connection.

The Stream ID value can be used as free-form, but there is a recommended convention so that all SRT users speak the same language. The intent of the convention is to:

* promote readability and consistency among free-form names,

* interpret some typical data in the key-value style.

B.1. General Syntax

This recommended syntax starts with the characters known as an executable specification in POSIX: #!.

The next two characters are:

```
: - this marks the YAML format, the only one currently used
The content format, which is either:
    : - the comma-separated keys with no nesting
    { - like above, but nesting is allowed and must end with }
```

(Nesting means that you can have multiple level brace-enclosed parts inside.)

The form of the key-value pair is:

```
key1=value1,key2=value2...
```

B.2. Standard Keys

Beside the general syntax, there are several top-level keys treated as standard keys. All single letter key definitions, including those not listed in this section, are reserved for future use. Users can additionally use custom key definitions with user_* or companyname_* prefixes, where user and companyname are to be replaced with an actual user or company name.

The existing key values MUST not be extended, and MUST not differ from those described in this section.

The following keys are standard:

* u: User Name, or authorization name, that is expected to control which password should be used for the connection. The application should interpret it to distinguish which user should be used by the listener party to set up the password.

* r: Resource Name identifies the name of the resource and facilitates selection should the listener party be able to serve multiple resources.

* h: Host Name identifies the hostname of the resource. For example, to request a stream with the URI somehost.com/videos/querry.php?vid=366 the hostname field should have somehost.com, and the resource name can have videos/querry.php?vid=366 or simply 366. Note that this is still a key to be specified explicitly. Support tools that apply simplifications and URI extraction are expected to insert only the host portion of the URI here.

* s: Session ID is a temporary resource identifier negotiated with the server, used just for verification. This is a one-shot identifier, invalidated after the first use. The expected usage is when details for the resource and authorization are negotiated over a separate connection first, and then the session ID is used here alone.

* t: Type specifies the purpose of the connection. Several standard types are defined, but users may extend the use:

  - stream (default, if not specified): for exchanging the user-specified payload for an application-defined purpose,

  - file: for transmitting a file, where r is the filename,

- auth: for exchanging sensible data. The r value states its purpose. No specific possible values for that are known so far (FUTURE USE).

* m: Mode expected for this connection:

  - request (default): the caller wants to receive the stream,

  - publish: the caller wants to send the stream data,

  - bidirectional: bidirectional data exchange is expected.

Note that "m" is not required in the case where Stream ID is not used to distinguish authorization or resources, and the caller is expected to send the data. This is only for cases where the listener can handle various purposes of the connection and is therefore required to know what the caller is attempting to do.

B.3. Examples

The example content of the StreamID is:

```
#!::u=admin,r=bluesbrothers1_hi
```

It specifies the username and the resource name of the stream to be served to the caller.

```
#!::u=johnny,t=file,m=publish,r=results.csv
```

This specifies that the file is expected to be transmitted from the caller to the listener and its name is results.csv.

Appendix C. Changelog

C.1. Since Version 00

* Improved and extended the description of "Encryption" section,

* Improved and extended the description of "Round-Trip Time Estimation" section,

* Extended the description of "Handshake" section with "Stream ID Extension Message", "Group Membership Extension" subsections,

* Extended "Handshake Messages" section with the detailed description of handshake procedure,

* Improved "Key Material" section description,

* Changed packet structure formatting for "Packet Structure" section,

* Did minor additions to the "Acknowledgement and Lost Packet Handling" section,

* Fixed broken links,

* Extended the list of references.

Authors' Addresses

Maxim Sharabayko
Haivision Network Video, GmbH

Email: maxsharabayko@haivision.com

Maria Sharabayko
Haivision Network Video, GmbH

Email: msharabayko@haivision.com

Jean Dube
Haivision

Email: jdube@haivision.com

Jeongseok Kim
SK Telecom Co., Ltd.

Email: jeongseok.kim@sk.com

Joonwoong Kim
SK Telecom Co., Ltd.

Email: joonwoong.kim@sk.com