

MOPS
Internet-Draft
Intended status: Standards Track
Expires: 10 September 2020

M.P. Sharabayko
M.A. Sharabayko
Haivision Network Video, GmbH
J. Dube
Haivision
JS. Kim
JW. Kim
SK Telecom Co., Ltd.
9 March 2020

The SRT Protocol
draft-sharabayko-mops-srt-00

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.

Note to Readers

Source for this draft and an issue tracker can be found at <https://github.com/haivision/srt-rfc> (<https://github.com/haivision/srt-rfc>).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

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

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

This Internet-Draft will expire on 10 September 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the [Trust Legal Provisions](#) and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Motivation	3
1.2. Secure Reliable Transport Protocol	4
2. Conventions and Definitions	5
3. Packet Structure	5
3.1. Data Packets	6
3.2. Control Packets	8
3.2.1. Handshake	9
3.2.2. Keep-Alive	17
3.2.3. ACK (Acknowledgement)	18
3.2.4. NAK (Loss Report)	20
3.2.5. Shutdown	21
3.2.6. ACKACK	22
4. SRT Data Transmission and Control	23
4.1. Stream Multiplexing	23
4.2. Data Transmission Modes	23
4.2.1. Message Mode	23
4.2.2. Live Mode	24
4.2.3. Buffer Mode	24
4.3. Handshake Messages	24
4.3.1. Caller-Listener Handshake	28
4.3.2. Rendezvous Handshake	30
4.4. SRT Buffer Latency	36
4.5. Timestamp Based Packet Delivery	37
4.5.1. Packet Delivery Time	38
4.6. Too-Late Packet Drop	40
4.7. Drift Management	41
4.8. Acknowledgement and Lost Packet Handling	42
4.8.1. Packet Acknowledgement (ACKs, ACKACKs)	43
4.8.2. Packet Retransmission (NAKs)	44
4.9. Bidirectional Transmission Queues	45
4.10. Round Trip Time Estimation	45

4.11. Congestion Control	45
5. Encryption	46
6. Security Considerations	46
7. IANA Considerations	46
Contributors	46
Acknowledgments	47
References	47
Normative References	47
Informative References	47
Appendix A . Packet Sequence List coding	49
Authors' Addresses	49

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 includes features which assure more reliable delivery.

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 protocol (UDT), 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 that recreates the signal characteristics on the receiver side, 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. Conventions and Definitions

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

3. Packet Structure

SRT packets are transmitted in UDP packets [[RFC0768](#)]. Every UDP packet carrying SRT traffic contains an SRT header (immediately after the UDP header).

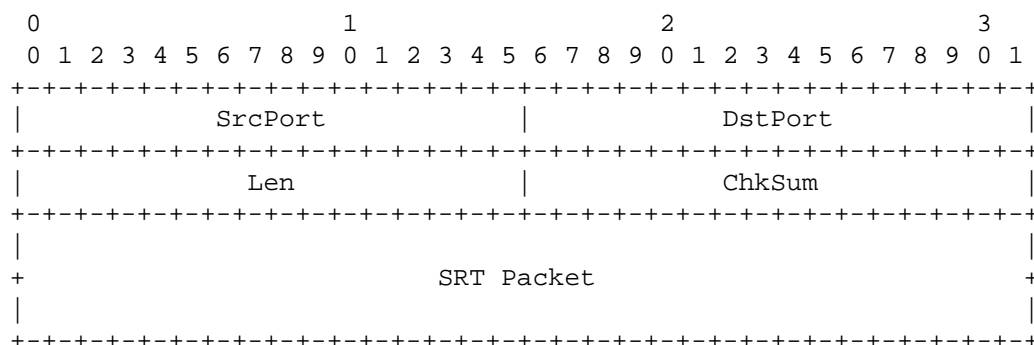


Figure 1: SRT packet as UDP payload

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.

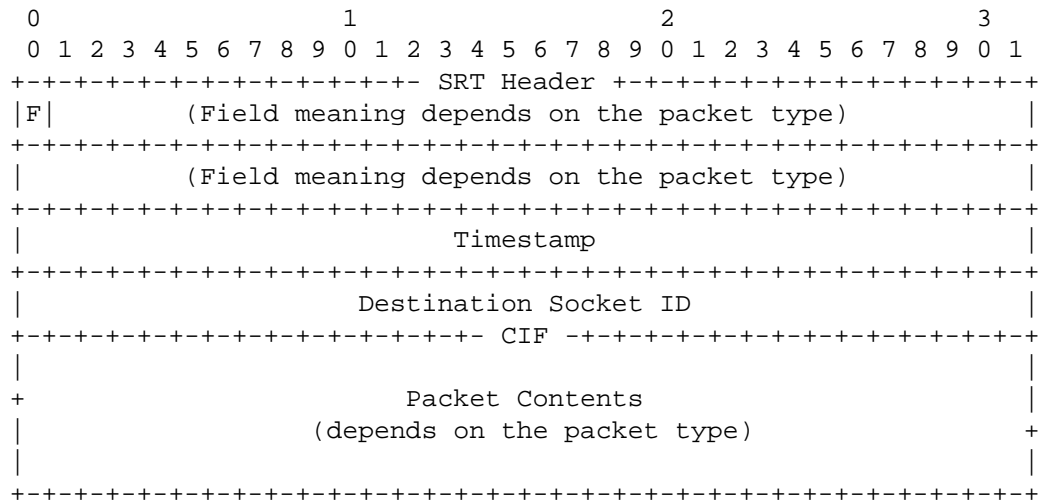


Figure 2: SRT packet structure

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 time stamp 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.

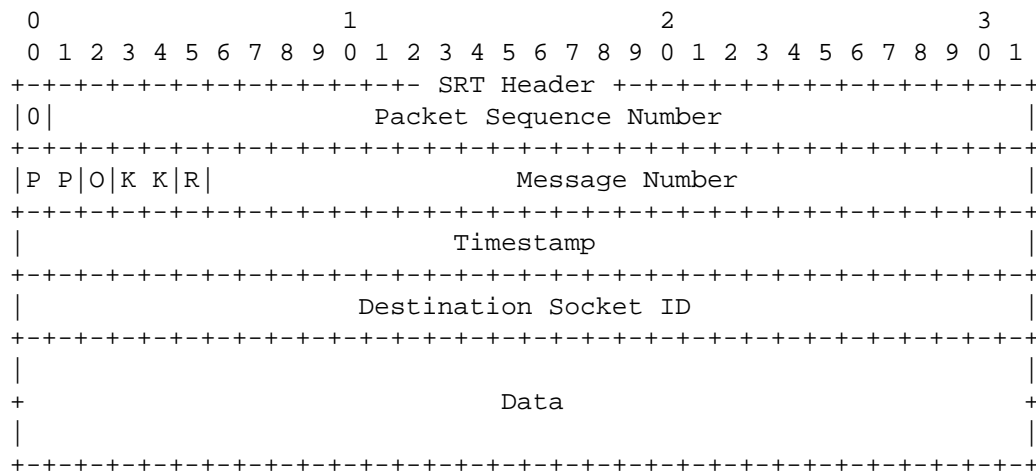


Figure 3: data packet structure

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).

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.

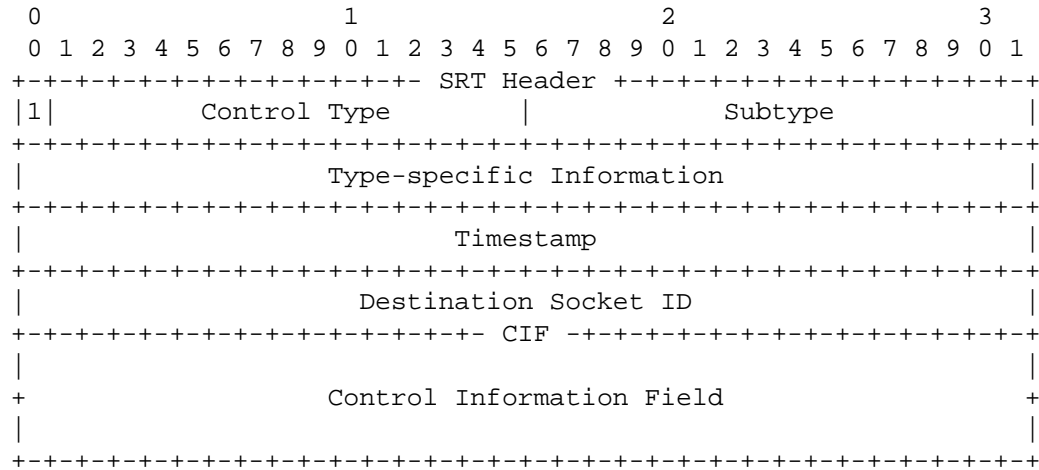


Figure 4: control packet structure

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.

Control Information Field (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 "0x7ffff" is reserved for a user-defined type.

Packet Type	Control Type	Subtype	Section
HANDSHAKE	0x0000	0x0	Section 3.2.1
KEEPALIVE	0x0001	0x0	Section 3.2.2
ACK	0x0002	0x0	Section 3.2.3
NAK (Loss Report)	0x0003	0x0	Section 3.2.4
SHUTDOWN	0x0005	0x0	Section 3.2.5
ACKACK	0x0006	0x0	Section 3.2.6
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.



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 block size. The values of this field are described in Table 2.

Value	Cipher family and block size
0	No Encryption
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
0xFFFFFFFFD	DONE
0xFFFFFFFFE	AGREEMENT
0xFFFFFFFFF	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): The sender's IPv4 or IPv6 address. The value consists of four 32-bit fields. In the case of IPv4 addresses, fields 2, 3 and 4 are padded with zeroes.

Extension Type (16 bits): The value of this field is used to process an integrated handshake. There are two extensions: Handshake Extension Message ([Section 3.2.1.1](#)) and Key Material Exchange ([Section 3.2.1.2](#)). 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.

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, and 2 for a Handshake Extension response.

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

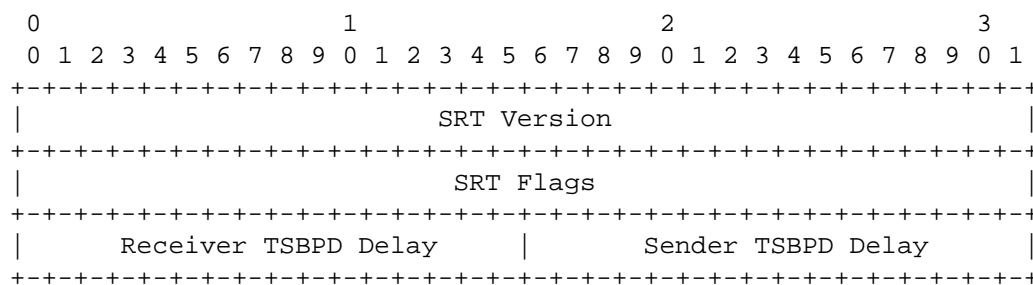


Figure 6: Handshake Extension Message structure

SRT Version (32 bits): SRT library version.

SRT Flags (32 bits): SRT configuration 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

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](#).

3.2.1.2. Key Material Exchange

The Key Material Exchange portion of a Handshake packet has both request and response type extensions. The value of a request is 3, and the response value is 4.

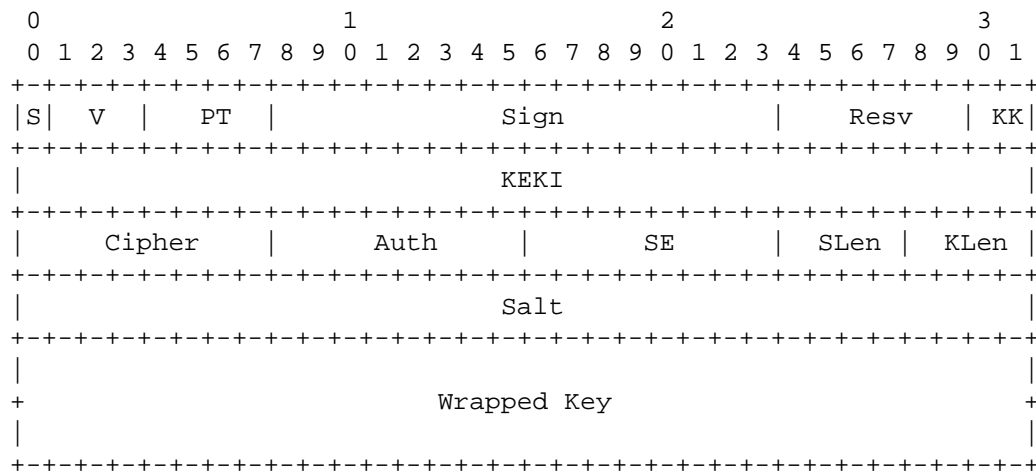


Figure 7: Key Material Extension structure

S (): 1 bit. Value: {0} This is a fixed-width field that is a remnant from the header of a previous design.

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: MSmsg - 2: KMmsg - 7: Reserved to discriminate MPEG-TS packet (0x47=sync byte)

Signature (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)

Reserved (Resv): 6 bits. Value: {0} This is a fixed-width field reserved for flag extension or other usage.

Key-based Data Encryption (KK): 2 bits. This is a fixed-width field that indicates whether or not data is encrypted: - 00b: not encrypted (data packets only) - 01b: even key - 10b: odd key

- 11b: even and odd keys

Key Encryption Key Index (KEKI): 32 bits. Value: {0} This is a fixed-width field for specifying the KEK index (big endian order) - 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 - 1: AES-ECB (not supported in SRT) - 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

Reserved (Resv1): 8 bits. Value: {0} This is a fixed-width field reserved for future use.

Reserved (Resv2): 16 bits. Value: {0} This is a fixed-width field reserved for future use.

Slen/4 (): 4 bits. Value: {0..255} This is a fixed-width field for specifying salt length in bytes divided by 4. Can be zero if no salt/IV present

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.

Salt (Slen): Slen*8 bits. Value: { } This is a variable-width field for specifying a salt key

Wrap (): (64+n * Klen * 8) bits. Value: { } This is a variable-width field for specifying Wrapped key(s), where n = 1 or 2 NOTE 1: n = (KK + 1)/2 NOTE 2: size in bytes = (((KK+1)/2) * Klen) + 8)

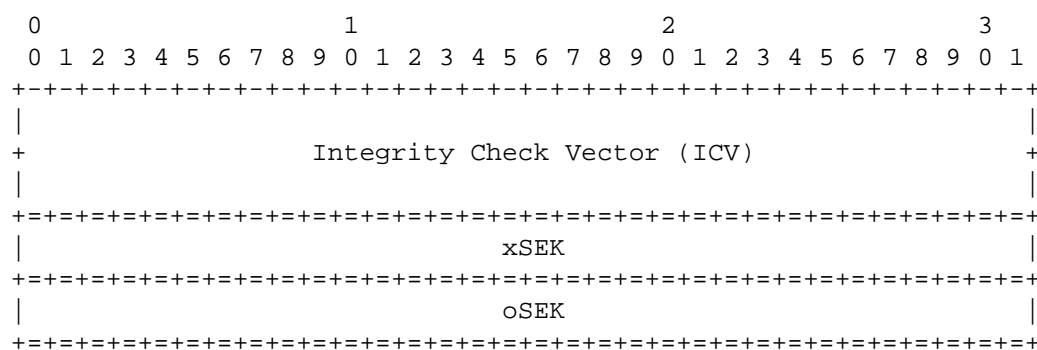


Figure 8: Unwrapped key structure

ICV (64 bits): 64-bit Integrity Check Vector(AES key wrap integrity).

xSEK (variable length): This field identifies an odd or even SEK. If both keys are present, then this field is eSEK (even key) and the next one is the odd key. The length of this field is calculated by $KLen * 4 * 8$.

oSEK (variable length): This field is present only when the message carries the two SEKs.

3.2.2. 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:

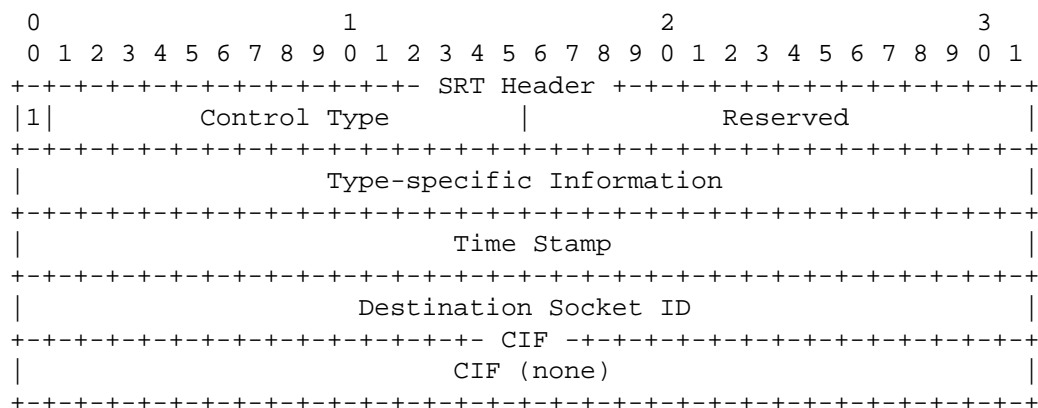


Figure 9: Keep-Alive structure

Packet Type (): 1 bit. Value: 1 The type value of a Keep-Alive control packet is "1".

Control Type (): 15 bits. Value: KEEPALIVE{1} This is a fixed-width field used to indicate message type

Reserved (): 16 bits. Value: ??? This is a fixed-width field reserved for future use.

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

Time Stamp (TS): 32 bits. Value: ??? This is a fixed-width field usually containing the time (in microseconds) when a packet was sent, although the real interpretation may vary depending on the type.

Destination Socket ID (DestSockID): 32 bits. Value: ??? This is a fixed-width field providing the socket ID to which a packet should be dispatched, although it may have the special value 0 when the packet is a connection request.

Control Information Field (CIF): n bits. Value: {none} This field must not appear in Keep-Alive control packets.

3.2.3. ACK (Acknowledgement)

Acknowledgement control packets are used to provide delivery status of data packets. These 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:

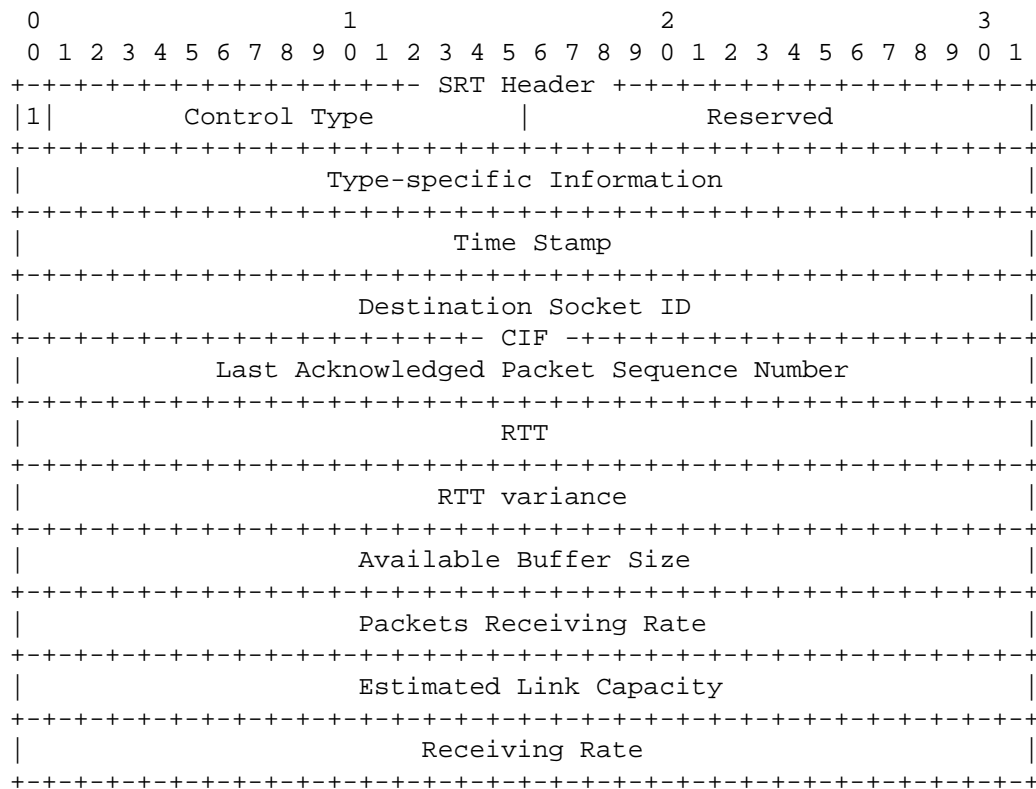


Figure 10: ACK control packet

Type-specific Information (32 bits): The time-specific Information (Figure 4) of the ACK packet stores the sequential number of the full ACK packet starting from 1.

Last Acknowledged Packet Sequence Number (32 bits): The sequence number of the last acknowledged data packet +1.

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 10.
- * 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).

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.

3.2.4. NAK (Loss Report)

Negative acknowledgement (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:

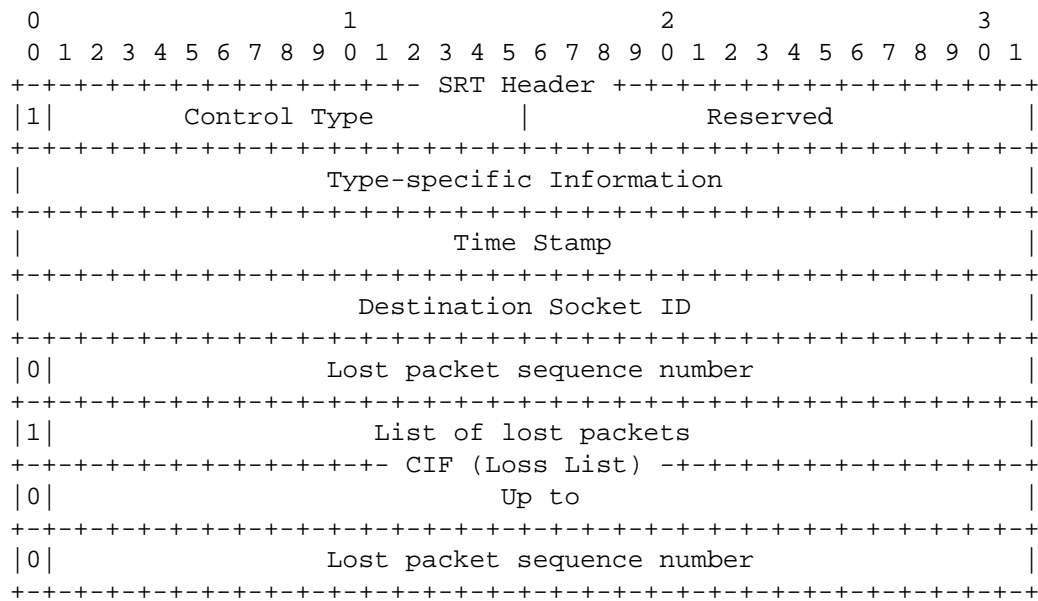


Figure 11: NAK control packet

Control Type: The type value of a NAK control packet is "3".

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

Control Information Field (CIF): A single value or a list of lost packets sequence numbers. See packet sequence number coding in [Appendix A](#).

3.2.5. Shutdown

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

An SRT SHUTDOWN Control packet is formatted as follows:

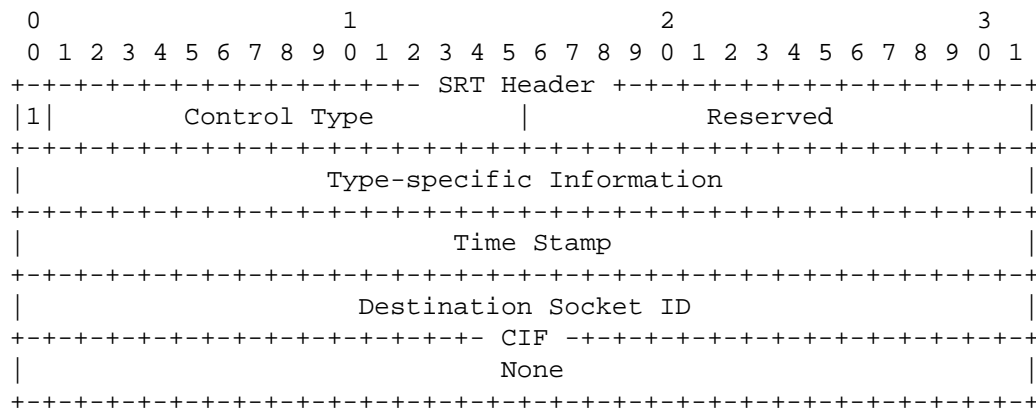


Figure 12: SHUTDOWN control packet

Control Type: The type value of Shutdown control packet is "5".

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

Control Information Field: This field must not appear in shutdown control packets.

3.2.6. 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:

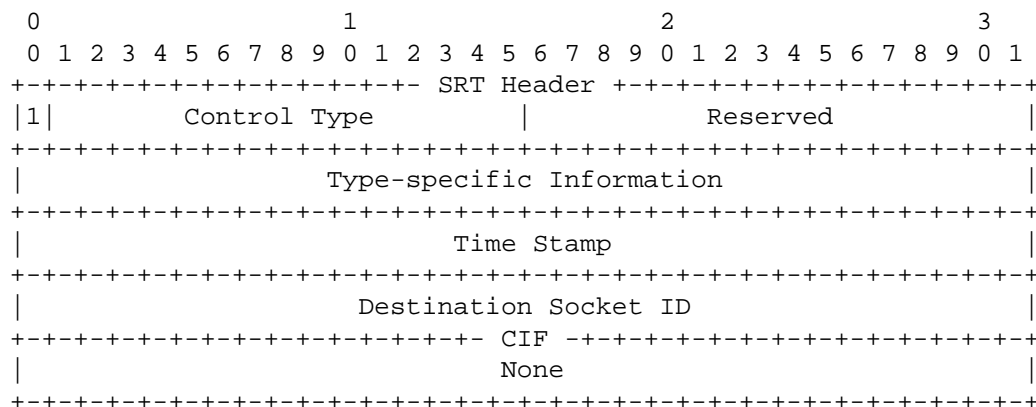


Figure 13: ACKACK control packet

Control Type: The type value of ACKACK control packet is "6".

Type-specific Information: ACK Sequence Number. This field is used for the sequence number of the ACK packet being acknowledged.

Control Information Field: This field must not appear in ACKACK control packets.

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 the SRT socket 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 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 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 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.

This 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 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

At this point the Listener still doesn't 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 doesn't 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 = URQ_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 behavior 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 URQ_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 URQ_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 doesn't 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 doesn't 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 acknowledgement (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 sections [Section 4.5](#), [Section 4.6](#) for details.

Latency is a value (specified 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 14).

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

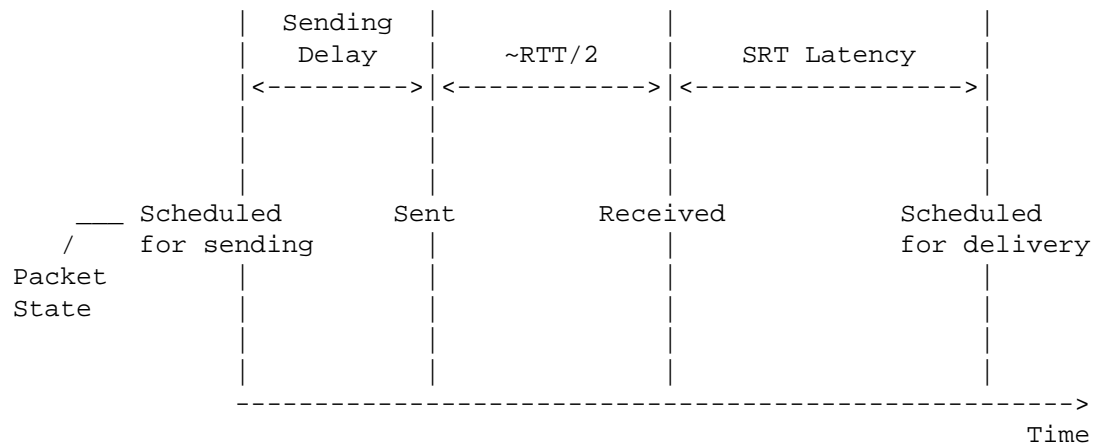


Figure 14: Key Latency Points during the Packet Transmission

The main packet states shown in Figure 14 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 $(RTT_0/2 + \text{SRT Latency})$ 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:

$$\text{PktTsbpdTime} = \text{TsbpdTimeBase} + \text{PKT_TIMESTAMP} + \text{TsbpdDelay} + \text{Drift}$$

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:

$$\text{TsbpdTimeBase} = \text{T_NOW} - \text{HSREQ_TIMESTAMP}$$

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 $\text{RTT}_0/2$, 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:

$$\text{TsbpdTimeBase} = \text{TsbpdTimeBase} + \text{MAX_TIMESTAMP} + 1$$

1. 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 acknowledgements (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 acknowledgement (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 acknowledgement (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 doesn't receive an ACK, it doesn't stop transmitting.

There are two conditions for sending an acknowledgement. A full ACK is based on a timer of 10 milliseconds (the ACK period). For high bit rate 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 doesn't 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 (header + 1 x 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 from the 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 doesn't 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 [sections 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.

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 CIF, 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

The round-trip time is estimated during the transmission of SRT data packets based on the time difference between the ACK packet is sent and the corresponding ACKACK is received by the data receiver.

[4.11.](#) Congestion Control

SRT provides certain mechanisms for the sender to get some feedback from the receiving side through the ACK packets ([Section 3.2.3](#)). 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 3.2.4](#)) 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 and BBR can apply, including the congestion control mechanism proposed in UDT [[GHG04b](#)]. 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

SRT supports encryption based on a pre-shared secret. Please refer to [[SRTTO](#)] for more information.

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 Dioussine, 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

- [AV1] Rivaz, P.d. and J. Haughton, "AV1 Bitstream & Decoding Process Specification", March 2020, <<https://aomediacodec.github.io/av1-spec/av1-spec.pdf>>.
- [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>>.
- [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-27](http://www.ietf.org/internet-drafts/draft-ietf-quic-http-27), 21 February 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-quic-http-27.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-27](http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-27), 21 February 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-27.txt>>.

[ISO13818-1]

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

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

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

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in [RFC 2119](https://www.rfc-editor.org/info/rfc2119) Key Words", [BCP 14](https://www.rfc-editor.org/info/rfc8174), [RFC 8174](https://www.rfc-editor.org/info/rfc8174), 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](https://www.rfc-editor.org/info/rfc8216), DOI 10.17487/RFC8216, August 2017, <<https://www.rfc-editor.org/info/rfc8216>>.

[RTMP] "Real-Time Messaging Protocol", March 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", March 2020, <<https://github.com/Haivision/srt>>.
- [SRTTO] Dube, J. and S. Matthews, "SRT Protocol Technical Overview", December 2019.
- [VP9] WebM, "VP9 Video Codec", March 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".

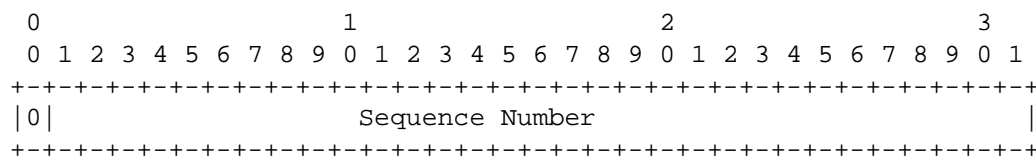


Figure 15: single sequence numbers coding

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".

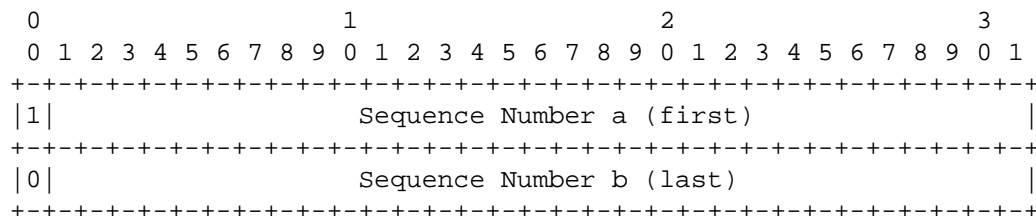


Figure 16: list of sequence numbers coding

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