

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 5, 2020

M. Nottingham
March 4, 2020

HTTP Link Hints
draft-nottingham-link-hint-02

Abstract

This memo specifies "HTTP Link Hints", a mechanism for annotating Web links to HTTP(S) resources with information that otherwise might be discovered by interacting with them.

Note to Readers

RFC EDITOR: please remove this section before publication

The issues list for this draft can be found at
<https://github.com/mnot/I-D/labels/link-hint> [1].

The most recent (often, unpublished) draft is at
<https://mnot.github.io/I-D/link-hint/> [2].

Recent changes are listed at <https://github.com/mnot/I-D/commits/gh-pages/link-hint> [3].

See also the draft's current status in the IETF datatracker, at
<https://datatracker.ietf.org/doc/draft-nottingham-link-hint/> [4].

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 September 5, 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	2
1.1. Notational Conventions	3
2. HTTP Link Hints	4
3. Pre-Defined HTTP Link Hints	4
3.1. allow	5
3.2. formats	5
3.3. links	5
3.4. accept-post	6
3.5. accept-patch	7
3.6. accept-ranges	7
3.7. accept-prefer	7
3.8. precondition-req	8
3.9. auth-schemes	8
3.10. status	9
4. Security Considerations	9
5. IANA Considerations	9
5.1. HTTP Link Hint Registry	9
6. References	10
6.1. Normative References	10
6.2. Informative References	11
6.3. URIs	12
Appendix A. Representing Link Hints in Link Headers	12
Appendix B. Acknowledgements	13
Author's Address	13

1. Introduction

HTTP [RFC7230] clients can discover a variety of information about a resource by interacting with it. For example, the methods supported can be learned through the Allow response header field, and the need

for authentication is conveyed with a 401 Authentication Required status code.

Often, it can be beneficial to know this information before interacting with the resource; not only can such knowledge save time (through reduced round trips), but it can also affect the choices available to the code or user driving the interaction.

For example, a user interface that presents the data from an HTTP-based API might need to know which resources the user has write access to, so that it can present the appropriate interface.

This specification defines a vocabulary of "HTTP link hints" that allow such metadata about HTTP resources to be attached to Web links [RFC8288], thereby making it available before the link is followed. It also establishes a registry for future hints.

Hints are just that - they are not a "contract", and are to only be taken as advisory. The runtime behaviour of the resource always overrides hinted information.

For example, a client might receive a hint that the PUT method is allowed on all "widget" resources. This means that generally, the client can PUT to them, but a specific resource might reject a PUT based upon access control or other considerations.

More fine-grained information might also be gathered by interacting with the resource (e.g., via a GET), or by another resource "containing" it (such as a "widgets" collection) or describing it (e.g., one linked to it with a "describedby" link relation).

There is not a single way to carry hints in a link; rather, it is expected that this will be done by individual link serialisations (see [RFC8288], Section 3.4.1). However, Appendix A does recommend how to include link hints in the existing Link HTTP header field.

1.1. Notational Conventions

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.

2. HTTP Link Hints

A HTTP link hint is a (key, value) tuple that describes the target resource of a Web link [RFC8288], or the link itself. The value's canonical form is a JSON [RFC8259] data structure specific to that hint.

Typically, link hints are serialised in links as target attributes ([RFC8288], Section 3.4.1).

In JSON-based formats, this can be achieved by simply serialising link hints as an object; for example:

```
{
  "_links": {
    "self": {
      "href": "/orders/523",
      "hints": {
        "allow": ["GET", "POST"],
        "accept-post": {
          "application/example+json":
            {}
        }
      }
    }
  }
}
```

In other link formats, this requires a mapping from the canonical JSON data model. One such mapping is described in Appendix A for the Link HTTP header field.

The information in a link hint SHOULD NOT be considered valid for longer than the freshness lifetime ([RFC7234], Section 4.2) of the representation that the link occurred within, and in some cases, it might be valid for a considerably shorter period.

Likewise, the information in a link hint is specific to the link it is attached to. This means that if a representation is specific to a particular user, the hints on links in that representation are also specific to that user.

3. Pre-Defined HTTP Link Hints

3.1. allow

- o Hint Name: allow
- o Description: Hints the HTTP methods that can be used to interact with the target resource; equivalent to the Allow HTTP response header.
- o Content Model: array (of strings)
- o Specification: [this document]

Content MUST be an array of strings, containing HTTP methods ([RFC7231], Section 4).

3.2. formats

- o Hint Name: formats
- o Description: Hints the representation type(s) that the target resource can produce and consume, using the GET and PUT (if allowed) methods respectively.
- o Content Model: object
- o Specification: [this document]

Content MUST be an object, whose keys are media types ([RFC7231], Section 3.1.1.1), and values are objects.

The object MAY have a "links" member, whose value is an object representing links (in the sense of [RFC8288]) whose context is any document that uses that format. Generally, this will be schema or profile ([RFC6906]) information. The "links" member has the same format as the "links" hint.

Furthermore, the object MAY have a "deprecated" member, whose value is either true or false, indicating whether support for the format might be removed in the near future.

All other members of the object are under control of the corresponding media type's definition.

3.3. links

- o Hint Name: links
- o Description: Hints at links whose context is the target resource.

- o Content Model: object
- o Specification: [this document]

The "links" hint contains links (in the sense of [RFC8288]) whose context is the hinted target resource, which are stable for the lifetime of the hint.

Content MUST be an object, whose member names are link relations ([RFC8288]) and values are objects that MUST have an "href" member whose value is a URI-reference ([RFC3986], using the original link as the base for resolution) for the link hint's target resource, and MAY itself contain link hints, serialised as the value for a "hints" member.

For example:

```
"links": {
  "edit-form": {
    "href": "./edit",
    "hints": {
      formats: {
        "application/json": {}
      }
    }
  }
}
```

3.4. accept-post

- o Hint Name: accept-post
- o Description: Hints the POST request format(s) that the target resource can consume.
- o Content Model: object
- o Specification: [this document]

Content MUST be an object, with the same constraints as for "formats".

When this hint is present, "POST" SHOULD be listed in the "allow" hint.

3.5. accept-patch

- o Hint Name: accept-patch
- o Description: Hints the PATCH [RFC5789] request format(s) that the target resource can consume; equivalent to the Accept-Patch HTTP response header.
- o Content Model: array (of strings)
- o Specification: [this document]

Content MUST be an array of strings, containing media types ([RFC7231], Section 3.1.1.1).

When this hint is present, "PATCH" SHOULD be listed in the "allow" hint.

3.6. accept-ranges

- o Hint Name: accept-ranges
- o Description: Hints the range-specifier(s) available for the target resource; equivalent to the Accept-Ranges HTTP response header [RFC7233].
- o Content Model: array (of strings)
- o Specification: [this document]

Content MUST be an array of strings, containing HTTP range-specifiers ([RFC7233], Section 3.1).

3.7. accept-prefer

- o Hint Name: accept-prefer
- o Description: Hints the preference(s) [RFC7240] that the target resource understands (and might act upon) in requests.
- o Content Model: array (of strings)
- o Specification: [this document]

Content MUST be an array of strings, contain preferences ([RFC7240], Section 2). Note that, by its nature, a preference can be ignored by the server.

3.8. precondition-req

- o Hint Name: precondition-req
- o Description: Hints that the target resource requires state-changing requests (e.g., PUT, PATCH) to include a precondition, as per [RFC7232], to avoid conflicts due to concurrent updates.
- o Content Model: array (of strings)
- o Specification: [this document]

Content MUST be an array of strings, with possible values "etag" and "last-modified" indicating type of precondition expected.

See also the 428 Precondition Required status code ([RFC6585]).

3.9. auth-schemes

- o Hint Name: auth-schemes
- o Description: Hints that the target resource requires authentication using the HTTP Authentication Framework [RFC7235].
- o Content Model: array (of objects)
- o Specification: [this document]

Content MUST be an array of objects, each with a "scheme" member containing a string that corresponds to a HTTP authentication scheme ([RFC7235]), and optionally a "realms" member containing an array of zero to many strings that identify protection spaces that the resource is a member of.

For example:

```
{
  "auth-req": [
    {
      "scheme": "Basic",
      "realms": ["private"]
    }
  ]
}
```


3.10. status

- o Hint Name: status
- o Description: Hints the status of the target resource.
- o Content Model: string
- o Specification: [this document]

Content MUST be a string; possible values are:

- o "deprecated" - indicates that use of the resource is not recommended, but it is still available.
- o "gone" - indicates that the resource is no longer available; i.e., it will return a 410 Gone HTTP status code if accessed.

4. Security Considerations

Clients need to exercise care when using hints. For example, a naive client might send credentials to a server that uses the auth-req hint, without checking to see if those credentials are appropriate for that server.

5. IANA Considerations

5.1. HTTP Link Hint Registry

This specification defines the HTTP Link Hint Registry. See Section 2 for a general description of the function of link hints.

Link hints are generic; that is, they are potentially applicable to any HTTP resource, not specific to one application of HTTP, nor to one particular format. Generally, they ought to be information that would otherwise be discoverable by interacting with the resource.

Hint names MUST be composed of the lowercase letters (a-z), digits (0-9), underscores ("_") and hyphens ("-"), and MUST begin with a lowercase letter.

Hint content MUST be described in terms of JSON values ([RFC8259], Section 3).

Hint semantics SHOULD be described in terms of the framework defined in [RFC8288].

New hints are registered using the Expert Review process described in [RFC8126] to enforce the criteria above. Requests for registration of new resource hints are to use the following template:

- o Hint Name: [hint name]
- o Description: [a short description of the hint's semantics]
- o Content Model: [valid JSON value types; see RFC627 Section 2.1]
- o Specification: [reference to specification document]

Initial registrations are enumerated in Section 3. The "rel", "rev", "hreflang", "media", "title", and "type" hint names are reserved, so as to avoid potential clashes with link serialisations.

6. References

6.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/RFC5789, March 2010, <<https://www.rfc-editor.org/info/rfc5789>>.
- [RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <<https://www.rfc-editor.org/info/rfc6585>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.

- [RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <<https://www.rfc-editor.org/info/rfc7232>>.
- [RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, DOI 10.17487/RFC7233, June 2014, <<https://www.rfc-editor.org/info/rfc7233>>.
- [RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <<https://www.rfc-editor.org/info/rfc7234>>.
- [RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <<https://www.rfc-editor.org/info/rfc7235>>.
- [RFC7240] Snell, J., "Prefer Header for HTTP", RFC 7240, DOI 10.17487/RFC7240, June 2014, <<https://www.rfc-editor.org/info/rfc7240>>.
- [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>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <<https://www.rfc-editor.org/info/rfc8288>>.

6.2. Informative References

- [RFC6906] Wilde, E., "The 'profile' Link Relation Type", RFC 6906, DOI 10.17487/RFC6906, March 2013, <<https://www.rfc-editor.org/info/rfc6906>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.

6.3. URIs

- [1] <https://github.com/mnot/I-D/labels/link-hint>
- [2] <https://mnot.github.io/I-D/link-hint/>
- [3] <https://github.com/mnot/I-D/commits/gh-pages/link-hint>
- [4] <https://datatracker.ietf.org/doc/draft-nottingham-link-hint/>

Appendix A. Representing Link Hints in Link Headers

A link hint can be represented in a Link header ([RFC8288], Section 3) as a link-extension.

When doing so, the JSON of the hint's content SHOULD be normalised to reduce extraneous spaces (%x20), and MUST NOT contain horizontal tabs (%x09), line feeds (%x0A) or carriage returns (%x0D). When they are part of a string value, these characters MUST be escaped as described in [RFC8259] Section 7; otherwise, they MUST be discarded.

Furthermore, if the content is an array or an object, the surrounding delimiters MUST be removed before serialisation. In other words, the outermost object or array is represented without the braces ("{}") or brackets ("[]") respectively, but this does not apply to inner objects or arrays.

For example, the two JSON values below are those of the fictitious "example" and "example1" hints, respectively:

```
"The Example Value"  
1.2
```

In a Link header, they would be serialised as:

```
Link: </>; rel="sample"; example="The Example Value";  
      example1=1.2
```

A more complex, single value for "example":

```
[  
  "foo",  
  -1.23,  
  true,  
  ["charlie", "bennet"],  
  {"cat": "thor"},  
  false  
]
```

would be serialised as:

```
Link: </>; rel="sample"; example="\foo", -1.23, true,  
      ["charlie", "bennet"], {"cat": "thor"}, false"
```

Appendix B. Acknowledgements

Thanks to Jan Algermissen, Mike Amundsen, Bill Burke, Graham Klyne, Leif Hedstrom, Jeni Tennison, Erik Wilde and Jorge Williams for their suggestions and feedback.

Author's Address

Mark Nottingham

Email: mnot@mnot.net

URI: <https://www.mnot.net/>

dispatch
Internet-Draft
Updates: 4122 (if approved)
Intended status: Standards Track
Expires: August 27, 2020

BGP. Peabody
February 24, 2020

UUID Format Update
draft-peabody-dispatch-new-uuid-format-00

Abstract

This document presents a new UUID format (version 6) which is suited for use as a database key.

A common case for modern applications is to create a unique identifier to be used as a primary key in a database table that is ordered by creation time, difficult to guess and has a compact text format. None of the existing UUID versions fulfill each of these requirements. This document is a proposal to update RFC4122 with a new UUID version that addresses these concerns.

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 August 27, 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	2
2. Background	2
3. Summary of Changes	5
3.1. Version 6	5
3.2. Timestamp	5
3.3. Clock Sequence and Node Parts	5
3.4. Alternate Text Formats	6
3.4.1. Base64 Text (Variant A)	7
3.4.2. Base32 Text	7
4. Uniqueness Service	7
5. Acknowledgements	8
6. IANA Considerations	8
7. Security Considerations	8
8. Normative References	8
Author's Address	8

1. Introduction

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Background

A lot of things have changed in the time since UUIDs were originally created. Modern applications have a need to use (and many have already implemented) UUIDs as database primary keys. However some properties of the existing specification are not well suited to this task.

The motivation for using UUIDs as database keys stems primarily from the fact that applications are increasingly distributed in nature. Simplistic "auto increment" schemes with integers in sequence do not work well in a distributed system since the effort required to synchronize such numbers across a network can easily become not worth it. The fact that UUIDs can be used to create unique and reasonably short values in distributed systems without requiring synchronization makes them a good candidate for use as a database key in such environments.

However, most of the existing UUID versions have poor database index locality. Meaning new values created in succession are not close to each other in the index and thus require inserts to be performed at random locations. The negative performance effects of which on common structures used for this (B-tree and its variants) can be dramatic. Newly inserted values should be time-ordered to address this. Version 1 UUIDs are time-ordered, but have other issues (see next).

A point of convenience and simplicity of implementation is that custom sort ordering logic should not be needed to put time ordered values in sequence. It is possible to sort Version 1 UUIDs by time but it requires breaking the bytes of the UUID into various pieces to determine the order (from the timestamp). Implementations would be simplified with a sort order where the UUID can simply be treated as an opaque sequence of bytes and ordered as such. This covers the first 64 bits of the UUID.

The latter portion (the last 64 bits) are in essence used to provide uniqueness.

Privacy and network security issues arise from using a MAC address in the node field of Version 1 UUIDs. Exposed MAC addresses can be used to locate machines to attack and can reveal various information about such machines (minimally manufacturer, potentially other details).

The use of MAC addresses in UUID Version 1, and the other hashing schemes used in the various versions, points to a more basic issue: There is no known way to guarantee "universal uniqueness". In fact, uniqueness needs are application-specific. MAC addresses in the node field might be okay for some applications. Others might be okay with using cryptographically secure random numbers (possibly with increased risk of collision). Still others might already have a predefined means to determine uniqueness for the application in question, such as a server node number. In an attempt to ensure uniqueness, the existing UUID format over-specifies exactly how this uniqueness is determined. This document posits the idea that while such mechanisms as MAC address may be okay for certain applications, it should be treated as a suggestion, not a requirement for proper implementation. Many applications will work perfectly well with more narrow and simpler uniqueness mechanisms (like using an existing node ID from whatever cluster the server is already in) and that this should be allowed as long as the uniqueness properties are clearly specified in the implementation. I.e. "using this field type as a database primary key will produce UUIDs which are unique within this database cluster" should be perfectly acceptable. Some other unnecessary requirement of global/universal uniqueness should not be needed for the implementation to be considered correct.

The property of "unguessability" is also application-specific. Some applications may desire increased security by using UUIDs which are difficult to guess (this way for example rate-limiting can be used to greatly reduce the probability of someone correctly guessing a new identifier or at least make it harder/take longer to do so). While applications should of course be using proper security measures, and relying solely on the unguessability of an identifier for security purposes is ill-advised, it is certainly not wrong to use this property as an additional layer of security. Examples of measures used to increase unguessability would be using cryptographically secure random data in the node and/or clock sequence fields (latter 64 bits), or using such random data in the subsecond portion of the timestamp (if subsecond time ordering is less important than unguessability for the application in question). The specification should indicate that such variations are acceptable as they do not change the format in an incompatible way.

Using a UUID as a database key generally requires communicating that UUID to other applications. The database server will store the value internally. It may be referenced in a query language (e.g. SQL), and/or transmitted in some database driver protocol. Other software, often written in another language, frequently then needs to store this identifier in its own memory and potentially perform its own operations like sorting and searching with it. And such identifiers are also commonly then used in protocols like HTTP where they indicate a particular resource. Sometimes they are typed in by humans. Sometimes constraints exist on which bytes may be used (such as an HTTP URL path). In most cases, shorter is better.

For these reasons, having a compact textual format is important. The existing hex format is already in wide use, so keeping it for backward compatibility makes sense. However an encoding using a base32 alphabet would be more compact and still be case-insensitive. A base64 alphabet would be even more compact (but require case-sensitivity). This document proposes both as options. This would allow applications to use a more compact text format for the situations needing textual representation (i.e. you can just put this value in URL and it is not unnecessarily long and does not require escaping). The alphabets used for base32 and base64 encoding should be in ASCII numeric value sequence so the text forms can also be sorted correctly as raw bytes. (This is not a property of the Base32 and Base64 standards from [RFC4648], however there are several variations in use so introducing a new one here for the express purpose of correct sorting would seem to be acceptable.)

3. Summary of Changes

The following is a summary of proposed changes to the UUID specification in [RFC4122]. Each is given as a statement of the problem or limitation to which it is addressed, along with a description of the proposed change.

3.1. Version 6

A UUID version 6 is proposed. It is ordered by creation time, sorts correctly as raw bytes, does not require use of a MAC address in the node section and has options for a compact text format.

3.2. Timestamp

The timestamp value from [RFC4122] (60-bit number of 100- nanosecond intervals since 00:00:00.00, 15 October 1582) is workable but the sequence in which the bytes are encoded (the lowest bytes first) results in unnecessary additional logic to sort correctly by timestamp. Ordering by timestamp is important for the use case of UUIDs as primary keys in a database since it improves locality by grouping new records close to each other (this can have major performance implications in large tables).

The proposed change is to encode the timestamp value into the same 60 bits as in [RFC4122] but in big-endian byte ordering. This way an application can sort by timestamp by simply treating the UUID as an opaque bunch of bytes.

3.3. Clock Sequence and Node Parts

The latter 64 bits of a UUID per [RFC4122] are the clock sequence and node fields. The node field is problematic as it encourages applications to use their MAC address which may present a security problem (it is not always appropriate to reveal the network address of a machine as it could make it the target of an attack or provide information about its manufacturer or other details). A lesser concern is that it also incidentally produces UUID with the same 6 bytes at the end and are visually more difficult to distinguish when looking at them in a list.

Seeing as the entire point of these last 64 bits is to ensure uniqueness, this document proposes that the strict definitions of clock sequence and node be relaxed. Instead implementations would be permitted to fill this section with random bytes and/or include an application defined value for uniqueness (such as a node number of a machine in a cluster).

Note for discussion: Another point to consider is that there is no known way to fully guarantee that that duplicate identifiers will not be created unless some per-determined outside source of uniqueness is employed. (Such as for version 1 UUIDs the MAC address.) However, applications each have their own requirements for uniqueness. Uniqueness within a single database cluster for example is acceptable in many cases. A specification that forces all UUIDs to be globally unique when it is not needed might not be a good idea. Identifiers are only as universally unique as their input, so it might be better to just clearly state this and say that it's fine if UUIDs are only guaranteed to be unique within a specific context if it makes sense for that application.

3.4. Alternate Text Formats

The existing UUID text format is hex encoded plus four hyphens. For many applications this is unnecessarily verbose. The same information can be encoded into significantly fewer bytes using a base 64 or base 32 alphabet.

Many applications have a need to use the unique identifier of a database record in a URL (e.g. in an HTTP request either in the path or a query parameter). It can also be useful as a file name. Being able to use a UUID for this purpose without having to escape certain characters it is a useful property.

This document proposes alternate alphabets for encoding UUIDs which are convenient for use in URLs and file names, and also sort correctly when treated as raw bytes. Some applications may not have the ability (or want) to encode and decode UUIDs from text to binary and thus having the text format also sort correctly as raw bytes is useful.

The standard Base64 and Base32 specifications in [RFC4648] do not have these properties, thus different alphabets are given for each.

Situations which require understanding the encoding SHOULD specify which encoding is used. For example, a database field which uses UUID version 6 with "b64a" encoding (see below), could be specified as type "UUID6B64A", which would result in binary storage according to UUID version 6, and otherwise read and write the value to/from applications in the b64a text format shown below. Note also that the length can be easily used to positively distinguish if a value is text or binary form. A 16-byte value will necessarily be raw unencoded bytes whereas text forms will be longer.

3.4.1. Base64 Text (Variant A)

UUIDs encoded in this form use the "url-safe base64" alphabet: "A" to "Z", "a" to "z", "0" to "9" and "-" and "_", but in ASCII value sequence. No padding characters are used.

The name "b64a" (not case sensitive) can be used by implementations to refer to this encoding.

Note: It might be useful to add another variation ("b64b") with a different alphabet. Hyphen and underscore are useful in a lot of places but there might be some others that are better for specific cases.

3.4.2. Base32 Text

Base32 can be useful if case-insensitivity is required.

UUIDs encoded in this form use digits "2" through "7" followed by "A" through "Z" (same alphabet as in [RFC4648] but in ASCII value sequence). Case is not sensitive. Implementations MAY choose to output lower case letters and doing so is also correct. Implementations which parse UUIDs encoded in this way MUST be case insensitive. No padding characters are used. Unless there is a specific reason for an implementation to do otherwise, it SHOULD output lower case base32 characters. The motivation for this it will increase the number of situations where UUIDs encoded in base32 and then used in different environments (some of which may be case sensitive, some not) are handled correctly by default. For example file names are case sensitive on some file systems and not on others. Preferring one specific (lower) case allows these to be used interchangeably with predictable results.

The name "b32a" (not case sensitive) can be used by implementations to refer to this encoding.

4. Uniqueness Service

An idea for discussion is that for applications which truly require globally unique identifiers one possible solution would be for someone to maintain a service which allocates numbers by time. In essence and for example "give me a 32-bit number that will be unique for the time range of midnight to midnight tomorrow". Such a service would be relatively easy to create. The effort required to maintain it depends largely on how much it is used. Applications using the same endpoint for this service would be guaranteed unique UUIDs. Companies could host their own too. I'm not sure if this sort of thing would be worth the effort but it's another idea for how to

address the global uniqueness issue for applications that really need it.

5. Acknowledgements

TODO: Acknowledgements for prior work and discussion.

6. IANA Considerations

TBD

7. Security Considerations

TODO: Provide additional information on "unguessability" as needed.

8. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <<https://www.rfc-editor.org/info/rfc4122>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.

Author's Address

Brad G. Peabody

Email: brad@peabody.io

dispatch
Internet-Draft
Updates: 4122 (if approved)
Intended status: Standards Track
Expires: 2 October 2022

BGP. Peabody
K. Davis
31 March 2022

New UUID Formats
draft-peabody-dispatch-new-uuid-format-03

Abstract

This document presents new Universally Unique Identifier (UUID) formats for use in modern applications and databases.

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 2 October 2022.

Copyright Notice

Copyright (c) 2022 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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
2. Terminology	4
2.1. Requirements Language	4
2.2. Abbreviations	5
3. Summary of Changes	5
3.1. changelog	5
4. Variant and Version Fields	7
5. New Formats	8
5.1. UUID Version 6	8
5.2. UUID Version 7	9
5.3. UUID Version 8	10
5.4. Max UUID	11
6. UUID Best Practices	12
6.1. Timestamp Granularity	12
6.2. Monotonicity and Counters	13
6.3. Distributed UUID Generation	16
6.4. Collision Resistance	17
6.5. Global and Local Uniqueness	18
6.6. Unguessability	18
6.7. Sorting	18
6.8. Opacity	19
6.9. DBMS and Database Considerations	19
7. IANA Considerations	19
8. Security Considerations	19
9. Acknowledgements	20
10. Normative References	20
11. Informative References	20
Appendix A. Example Code	22
A.1. Creating a UUIDv6 Value	22
A.2. Creating a UUIDv7 Value	23
A.3. Creating a UUIDv8 Value	24
Appendix B. Test Vectors	24
B.1. Example of a UUIDv6 Value	25
B.2. Example of a UUIDv7 Value	26
B.3. Example of a UUIDv8 Value	26
Appendix C. Version and Variant Tables	27
C.1. Variant 10xx Versions	27
Authors' Addresses	28

1. Introduction

Many things have changed in the time since UUIDs were originally created. Modern applications have a need to create and utilize UUIDs as the primary identifier for a variety of different items in complex computational systems, including but not limited to database keys, file names, machine or system names, and identifiers for event-driven transactions.

One area UUIDs have gained popularity is as database keys. This stems from the increasingly distributed nature of modern applications. In such cases, "auto increment" schemes often used by databases do not work well, as the effort required to coordinate unique numeric identifiers across a network can easily become a burden. The fact that UUIDs can be used to create unique, reasonably short values in distributed systems without requiring synchronization makes them a good alternative, but UUID versions 1-5 lack certain other desirable characteristics:

1. Non-time-ordered UUID versions such as UUIDv4 have poor database index locality. Meaning new values created in succession are not close to each other in the index and thus require inserts to be performed at random locations. The negative performance effects of which on common structures used for this (B-tree and its variants) can be dramatic.
2. The 100-nanosecond, Gregorian epoch used in UUIDv1 timestamps is uncommon and difficult to represent accurately using a standard number format such as [IEEE754].
3. Introspection/parsing is required to order by time sequence; as opposed to being able to perform a simple byte-by-byte comparison.
4. Privacy and network security issues arise from using a MAC address in the node field of Version 1 UUIDs. Exposed MAC addresses can be used as an attack surface to locate machines and reveal various other information about such machines (minimally manufacturer, potentially other details). Additionally, with the advent of virtual machines and containers, MAC address uniqueness is no longer guaranteed.
5. Many of the implementation details specified in [RFC4122] involve trade offs that are neither possible to specify for all applications nor necessary to produce interoperable implementations.

6. [RFC4122] does not distinguish between the requirements for generation of a UUID versus an application which simply stores one, which are often different.

Due to the aforementioned issue, many widely distributed database applications and large application vendors have sought to solve the problem of creating a better time-based, sortable unique identifier for use as a database key. This has lead to numerous implementations over the past 10+ years solving the same problem in slightly different ways.

While preparing this specification the following 16 different implementations were analyzed for trends in total ID length, bit Layout, lexical formatting/encoding, timestamp type, timestamp format, timestamp accuracy, node format/components, collision handling and multi-timestamp tick generation sequencing.

1. [ULID] by A. Feerasta
2. [LexicalUUID] by Twitter
3. [Snowflake] by Twitter
4. [Flake] by Boundary
5. [ShardingID] by Instagram
6. [KSUID] by Segment
7. [Elasticflake] by P. Percy
8. [FlakeID] by T. Pawlak
9. [Sonyflake] by Sony
10. [orderedUuid] by IT. Cabrera
11. [COMBGUID] by R. Tallent
12. [SID] by A. Chilton
13. [pushID] by Google
14. [XID] by O. Poitrey
15. [ObjectID] by MongoDB
16. [CUID] by E. Elliott

An inspection of these implementations and the issues described above has led to this document which attempts to adapt UUIDs to address these issues.

2. Terminology

2.1. Requirements Language

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.

2.2. Abbreviations

The following abbreviations are used in this document:

UUID	Universally Unique Identifier [RFC4122]
CSPRNG	Cryptographically Secure Pseudo-Random Number Generator
MAC	Media Access Control
MSB	Most Significant Bit
DBMS	Database Management System

3. Summary of Changes

The following UUIDs are hereby introduced:

UUID version 6 (UUIDv6)

A re-ordering of UUID version 1 so it is sortable as an opaque sequence of bytes. Easy to implement given an existing UUIDv1 implementation. See Section 5.1

UUID version 7 (UUIDv7)

An entirely new time-based UUID bit layout sourced from the widely implemented and well known Unix Epoch timestamp source. See Section 5.2

UUID version 8 (UUIDv8)

A free-form UUID format which has no explicit requirements except maintaining backward compatibility. See Section 5.3

Max UUID

A specialized UUID which is the inverse of [RFC4122], Section 4.1.7 See Section 5.4

3.1. changelog

RFC EDITOR PLEASE DELETE THIS SECTION.

draft-03

- Reworked the draft body to make the content more concise
- UUIDv6 section reworked to just the reorder of the timestamp
- UUIDv7 changed to simplify timestamp mechanism to just millisecond Unix timestamp
- UUIDv8 relaxed to be custom in all elements except version and variant

- Introduced Max UUID.
- Added C code samples in Appendix.
- Added test vectors in Appendix.
- Version and Variant section combined into one section.
- Changed from pseudo-random number generators to cryptographically secure pseudo-random number generator (CSPRNG).
- Combined redundant topics from all UUIDs into sections such as Timestamp granularity, Monotonicity and Counters, Collision Resistance, Sorting, and Unguessability, etc.
- Split Encoding and Storage into Opacity and DBMS and Database Considerations
- Reworked Global Uniqueness under new section Global and Local Uniqueness
- Node verbiage only used in UUIDv6 all others reference random/rand instead
- Clock sequence verbiage changed simply to counter in any section other than UUIDv6
- Added Abbreviations section
- Updated IETF Draft XML Layout
- Added information about little-endian UUIDs

draft-02

- Added Changelog
- Fixed misc. grammatical errors
- Fixed section numbering issue
- Fixed some UUIDvX reference issues
- Changed all instances of "motonic" to "monotonic"
- Changed all instances of "#-bit" to "# bit"
- Changed "proceeding" verbiage to "after" in section 7
- Added details on how to pad 32 bit Unix timestamp to 36 bits in UUIDv7
- Added details on how to truncate 64 bit Unix timestamp to 36 bits in UUIDv7
- Added forward reference and bullet to UUIDv8 if truncating 64 bit Unix Epoch is not an option.
- Fixed bad reference to non-existent "time_or_node" in section 4.5.4

draft-01

- Complete rewrite of entire document.
- The format, flow and verbiage used in the specification has been reworked to mirror the original RFC 4122 and current IETF standards.
- Removed the topics of UUID length modification, alternate UUID text formats, and alternate UUID encoding techniques.

- Research into 16 different historical and current implementations of time-based universal identifiers was completed at the end of 2020 in attempt to identify trends which have directly influenced design decisions in this draft document (<https://github.com/uuid6/uuid6-ietf-draft/tree/master/research>)
- Prototype implementation have been completed for UUIDv6, UUIDv7, and UUIDv8 in various languages by many GitHub community members. (<https://github.com/uuid6/prototypes>)

4. Variant and Version Fields

The variant bits utilized by UUIDs in this specification remain in the same octet as originally defined by [RFC4122], Section 4.1.1.

The next table details Variant 10xx (8/9/A/B) and the new versions defined by this specification. A complete guide to all versions within this variant has been includes in Appendix C.1.

Ms0	Ms1	Ms2	Ms3	Version	Description
0	1	1	0	6	Reordered Gregorian time-based UUID specified in this document.
0	1	1	1	7	Unix Epoch time-based UUID specified in this document.
1	0	0	0	8	Reserved for custom UUID formats specified in this document

Table 1: New UUID variant 10xx (8/9/A/B) versions defined by this specification

For UUID version 6, 7 and 8 the variant field placement from [RFC4122] are unchanged. An example version/variant layout for UUIDv6 follows the table where M is the version and N is the variant.

```

00000000-0000-6000-8000-000000000000
00000000-0000-6000-9000-000000000000
00000000-0000-6000-A000-000000000000
00000000-0000-6000-B000-000000000000
xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxxxx

```

Figure 1: UUIDv6 Variant Examples

5. New Formats

The UUID format is 16 octets; the variant bits in conjunction with the version bits described in the next section in determine finer structure.

5.1. UUID Version 6

UUID version 6 is a field-compatible version of UUIDv1, reordered for improved DB locality. It is expected that UUIDv6 will primarily be used in contexts where there are existing v1 UUIDs. Systems that do not involve legacy UUIDv1 SHOULD consider using UUIDv7 instead.

Instead of splitting the timestamp into the low, mid and high sections from UUIDv1, UUIDv6 changes this sequence so timestamp bytes are stored from most to least significant. That is, given a 60 bit timestamp value as specified for UUIDv1 in [RFC4122], Section 4.1.4, for UUIDv6, the first 48 most significant bits are stored first, followed by the 4 bit version (same position), followed by the remaining 12 bits of the original 60 bit timestamp.

The clock sequence bits remain unchanged from their usage and position in [RFC4122], Section 4.1.5.

The 48 bit node SHOULD be set to a pseudo-random value however implementations MAY choose to retain the old MAC address behavior from [RFC4122], Section 4.1.6 and [RFC4122], Section 4.5. For more information on MAC address usage within UUIDs see the Section 8

The format for the 16-byte, 128 bit UUIDv6 is shown in Figure 1

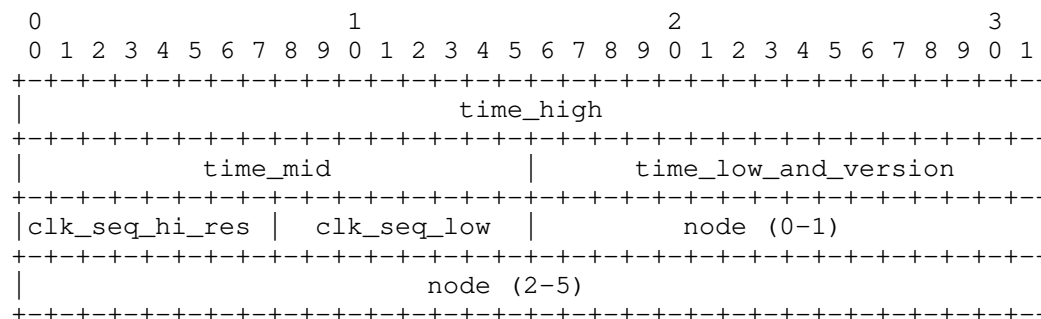


Figure 2: UUIDv6 Field and Bit Layout

time_high:

The most significant 32 bits of the 60 bit starting timestamp.
Occupies bits 0 through 31 (octets 0-3)

time_mid:

The middle 16 bits of the 60 bit starting timestamp. Occupies bits 32 through 47 (octets 4-5)

time_low_and_version:

The first four most significant bits MUST contain the UUIDv6 version (0110) while the remaining 12 bits will contain the least significant 12 bits from the 60 bit starting timestamp. Occupies bits 48 through 63 (octets 6-7)

clk_seq_hi_res:

The first two bits MUST be set to the UUID variant (10) The remaining 6 bits contain the high portion of the clock sequence. Occupies bits 64 through 71 (octet 8)

clock_seq_low:

The 8 bit low portion of the clock sequence. Occupies bits 72 through 79 (octet 9)

node:

48 bit spatially unique identifier Occupies bits 80 through 127 (octets 10-15)

With UUIDv6 the steps for splitting the timestamp into time_high and time_mid are OPTIONAL since the 48 bits of time_high and time_mid will remain in the same order. An extra step of splitting the first 48 bits of the timestamp into the most significant 32 bits and least significant 16 bits proves useful when reusing an existing UUIDv1 implementation.

5.2. UUID Version 7

UUID version 7 features a time-ordered value field derived from the widely implemented and well known Unix Epoch timestamp source, the number of milliseconds seconds since midnight 1 Jan 1970 UTC, leap seconds excluded. As well as improved entropy characteristics over versions 1 or 6.

Implementations SHOULD utilize UUID version 7 over UUID version 1 and 6 if possible.

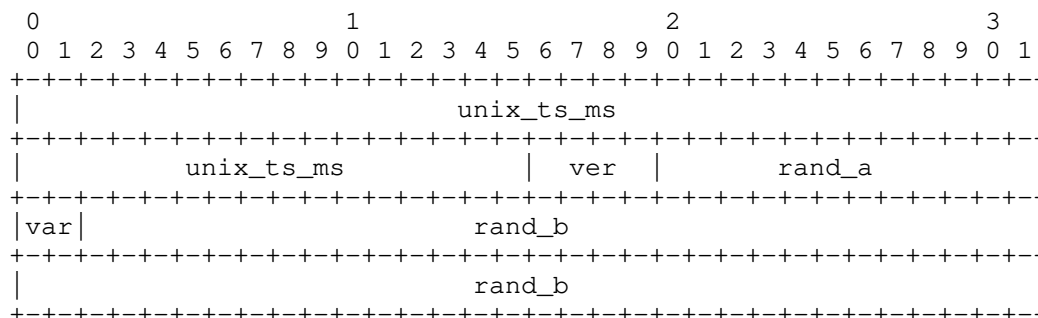


Figure 3: UUIDv7 Field and Bit Layout

unix_ts_ms:

48 bit big-endian unsigned number of Unix epoch timestamp as per Section 6.1.

ver:

4 bit UUIDv7 version set as per Section 4

rand_a:

12 bits pseudo-random data to provide uniqueness as per Section 6.2 and Section 6.6.

var:

The 2 bit variant defined by Section 4.

rand_b:

The final 62 bits of pseudo-random data to provide uniqueness as per Section 6.2 and Section 6.6.

5.3. UUID Version 8

UUID version 8 provides an RFC-compatible format for experimental or vendor-specific use cases. The only requirement is that the variant and version bits **MUST** be set as defined in Section 4. UUIDv8's uniqueness will be implementation-specific and **SHOULD NOT** be assumed.

The only explicitly defined bits are the Version and Variant leaving 120 bits for implementation specific time-based UUIDs. To be clear: UUIDv8 is not a replacement for UUIDv4 where all 122 extra bits are filled with random data.

Some example situations in which UUIDv8 usage could occur:

- * An implementation would like to embed extra information within the UUID other than what is defined in this document.

* An implementation has other application/language restrictions which inhibit the use of one of the current UUIDs.

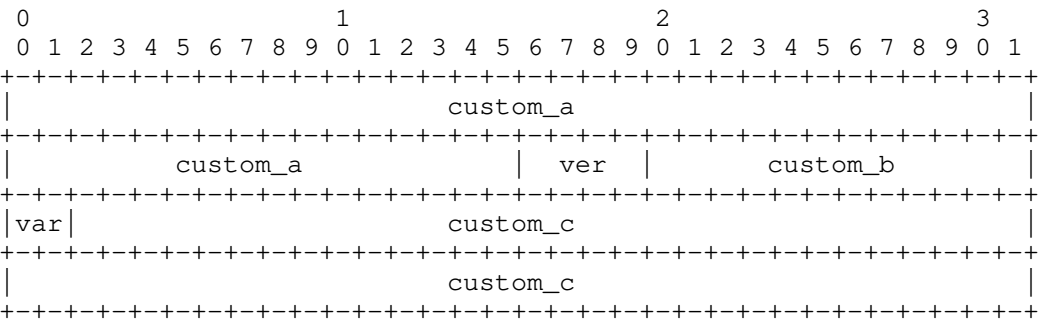


Figure 4: UUIDv8 Field and Bit Layout

custom_a:
The first 48 bits of the layout that can be filled as an implementation sees fit.

ver:
The 4 bit version field as defined by Section 4

custom_b:
12 more bits of the layout that can be filled as an implementation sees fit.

var:
The 2 bit variant field as defined by Section 4.

custom_c:
The final 62 bits of the layout immediatly following the var field to be filled as an implementation sees fit.

5.4. Max UUID

The Max UUID is special form of UUID that is specified to have all 128 bits set to 1. This UUID can be thought of as the inverse of Nil UUID defined in [RFC4122], Section 4.1.7

FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFFFF

Figure 5: Max UUID Format

6. UUID Best Practices

The minimum requirements for generating UUIDs are described in this document for each version. Everything else is an implementation detail and up to the implementer to decide what is appropriate for a given implementation. That being said, various relevant factors are covered below to help guide an implementer through the different trade-offs among differing UUID implementations.

6.1. Timestamp Granularity

UUID timestamp source, precision and length was the topic of great debate while creating this specification. As such choosing the right timestamp for your application is a very important topic. This section will detail some of the most common points on this topic.

Reliability:

Implementations SHOULD use the current timestamp from a reliable source to provide values that are time-ordered and continually increasing. Care SHOULD be taken to ensure that timestamp changes from the environment or operating system are handled in a way that is consistent with implementation requirements. For example, if it is possible for the system clock to move backward due to either manual adjustment or corrections from a time synchronization protocol, implementations must decide how to handle such cases. (See Altering, Fuzzing, or Smearing bullet below.)

Source:

UUID version 1 and 6 both utilize a Gregorian epoch timestamp while UUIDv7 utilizes a Unix Epoch timestamp. If other timestamp sources or a custom timestamp epoch are required UUIDv8 SHOULD be leveraged.

Sub-second Precision and Accuracy:

Many levels of precision exist for timestamps: milliseconds, microseconds, nanoseconds, and beyond. Additionally fractional representations of sub-second precision may be desired to mix various levels of precision in a time-ordered manner. Furthermore, system clocks themselves have an underlying granularity and it is frequently less than the precision offered by the operating system. With UUID version 1 and 6, 100-nanoseconds of precision are present while UUIDv7 features fixed millisecond level of precision within the Unix epoch that does not exceed the granularity capable in most modern systems. For other levels of precision UUIDv8 SHOULD be utilized.

Length:

The length of a given timestamp directly impacts how long a given UUID will be valid. That is, how many timestamp ticks can be contained in a UUID before the maximum value for the timestamp field is reached. Care should be given to ensure that the proper length is selected for a given timestamp. UUID version 1 and 6 utilize a 60 bit timestamp and UUIDv7 features a 48 bit timestamp.

Altering, Fuzzing, or Smearing:

Implementations MAY alter the actual timestamp. Some examples included security considerations around providing a real clock value within a UUID, to correct inaccurate clocks or to handle leap seconds. This specification makes no requirement or guarantee about how close the clock value needs to be to actual time.

Padding:

When timestamp padding is required, implementations MUST pad the most significant bits (left-most) bits with zeros. An example is padding the most significant, left-most bits of a 32 bit Unix timestamp with zero's to fill out the 48 bit timestamp in UUIDv7.

Truncating:

Similarly, when timestamps need to be truncated: the lower, least significant bits MUST be used. An example would be truncating a 64 bit Unix timestamp to the least significant, right-most 48 bits for UUIDv7.

6.2. Monotonicity and Counters

Monotonicity is the backbone of time-based sortable UUIDs. Naturally time-based UUIDs from this document will be monotonic due to an embedded timestamp however implementations can guarantee additional monotonicity via the concepts covered in this section.

Additionally, care MUST be taken to ensure UUIDs generated in batches are also monotonic. That is, if one-thousand UUIDs are generated for the same timestamp; there is sufficient logic for organizing the creation order of those one-thousand UUIDs. For batch UUID creation implementations MAY utilize a monotonic counter which SHOULD increment for each UUID created during a given timestamp.

For single-node UUID implementations that do not need to create batches of UUIDs, the embedded timestamp within UUID version 1, 6, and 7 can provide sufficient monotonicity guarantees by simply ensuring that timestamp increments before creating a new UUID. For the topic of Distributed Nodes please refer to Section 6.3

Implementations SHOULD choose one method for single-node UUID implementations that require batch UUID creation.

Fixed-Length Dedicated Counter Bits (Method 1):

This references the practice of allocating a specific number of bits in the UUID layout to the sole purpose of tallying the total number of UUIDs created during a given UUID timestamp tick. Positioning of a fixed bit-length counter SHOULD be immediately after the embedded timestamp. This promotes sortability and allows random data generation for each counter increment. With this method `rand_a` section of UUIDv7 MAY be utilized as fixed-length dedicated counter bits. In the event more counter bits are required the most significant, left-most, bits of `rand_b` MAY be leveraged as additional counter bits.

Monotonic Random (Method 2):

With this method the random data is extended to also double as a counter. This monotonic random can be thought of as a "randomly seeded counter" which MUST be incremented in the least significant position for each UUID created on a given timestamp tick. UUIDv7's `rand_b` section SHOULD be utilized with this method to handle batch UUID generation during a single timestamp tick.

The following sub-topics cover methods behind incrementing either type of counter method:

Plus One Increment (Type A):

With this increment logic the counter method is incremented by one for every UUID generation. When this increment method is utilized with Fixed-Length Dedicated Counter the trailing random generated for each new UUID can help produce unguessable UUIDs. When this increment method is utilized with Monotonic Random Counters the resulting values are easily guessable. Implementations that favor unguessability SHOULD NOT utilize this method with the monotonic random method.

Random Increment (Type B):

With this increment the actual increment of the counter MAY be a random integer of any desired length larger than zero. When this increment method is utilized with Fixed-Length Dedicated Counters the random increments MAY deplete the counter bit space (including any rollover guards) faster than the desired if a counter of adequate length is not selected. When this increment method is utilized with Monotonic Random Counters the counter ensures the UUIDs retain the required level of unguessability characters provided by the underlying entropy.

The following sub-topics cover topics related solely with creating reliable fixed-length dedicated counters:

Fixed-Length Dedicated Counter Seeding:

Implementations utilizing fixed-length counter method SHOULD randomly initialize the counter with each new timestamp tick. However, when the timestamp has not incremented; the counter SHOULD be frozen and incremented via the desired increment logic. When utilizing a randomly seeded counter alongside Method 1; the random MAY be regenerated with each counter increment without impacting sortability. The downside is that Method 1 is prone to overflows if a counter of adequate length is not selected or the random data generated leaves little room for the required number of increments. Implementations utilizing fixed-length counter method MAY also choose to randomly initialize a portion counter rather than the entire counter. For example, a 24 bit counter could have the 23 bits in least-significant, right-most, position randomly initialized. The remaining most significant, left-most counter bits are initialized as zero for the sole purpose of guarding against counter rollovers.

Fixed-Length Dedicated Counter Length:

Care MUST be taken to select a counter bit-length that can properly handle the level of timestamp precision in use. For example, millisecond precision SHOULD require a larger counter than a timestamp with nanosecond precision. General guidance is that the counter SHOULD be at least 12 bits but no longer than 42 bits. Care SHOULD also be given to ensure that the counter length selected leaves room for sufficient entropy in the random portion of the UUID after the counter. This entropy helps improve the unguessability characteristics of UUIDs created within the batch.

The following sub-topics cover rollover handling with either type of counter method:

Counter Rollover Guards:

The technique from Fixed-Length Dedicated Counter Seeding which describes allocating a segment of the fixed-length counter as a rollover guard is also recommended and SHOULD be employed to help mitigate counter rollover issues. This same technique can be leveraged with Monotonic random counter methods by ensuring the total length of a possible increment in the least significant, right most position is less than the total length of the random being incremented. As such the most significant, left-most, bits can be incremented as rollover guarding.

Counter Rollover Handling:

Counter rollovers SHOULD be handled by the application to avoid sorting issues. The general guidance is that applications that care about absolute monotonicity and sortability SHOULD freeze the counter and wait for the timestamp to advance which ensures monotonicity is not broken.

Implementations MAY use the following logic to ensure UUIDs featuring embedded counters are monotonic in nature:

1. Compare the current timestamp against the previously stored timestamp.
2. If the current timestamp is equal to the previous timestamp; increment the counter according to the desired method and type.
3. If the current timestamp is greater than the previous timestamp; re-initialize the desired counter method to the new timestamp and generate new random bytes (if the bytes were frozen or being used as the seed for a monotonic counter).

Implementations SHOULD check if the the currently generated UUID is greater than the previously generated UUID. If this is not the case then any number of things could have occurred. Such as, but not limited to, clock rollbacks, leap second handling or counter rollovers. Applications SHOULD embed sufficient logic to catch these scenarios and correct the problem ensuring the next UUID generated is greater than the previous.

6.3. Distributed UUID Generation

Some implementations MAY desire to utilize multi-node, clustered, applications which involve two or more nodes independently generating UUIDs that will be stored in a common location. While UUIDs already feature sufficient entropy to ensure that the chances of collision are low as the total number of nodes increase; so does the likelihood of a collision. This section will detail the approaches that MAY be utilized by multi-node UUID implementations in distributed environments.

Centralized Registry:

With this method all nodes tasked with creating UUIDs consult a central registry and confirm the generated value is unique. As applications scale the communication with the central registry could become a bottleneck and impact UUID generation in a negative way. Utilization of shared knowledge schemes with central/global registries is outside the scope of this specification.

Node IDs:

With this method, a pseudo-random Node ID value is placed within the UUID layout. This identifier helps ensure the bit-space for a given node is unique, resulting in UUIDs that do not conflict with any other UUID created by another node with a different node id. Implementations that choose to leverage an embedded node id SHOULD utilize UUIDv8. The node id SHOULD NOT be an IEEE 802 MAC address as per Section 8. The location and bit length are left to implementations and are outside the scope of this specification. Furthermore, the creation and negotiation of unique node ids among nodes is also out of scope for this specification.

Utilization of either a Centralized Registry or Node ID are not required for implementing UUIDs in this specification. However implementations SHOULD utilize one of the two aforementioned methods if distributed UUID generation is a requirement.

6.4. Collision Resistance

Implementations SHOULD weigh the consequences of UUID collisions within their application and when deciding between UUID versions that use entropy (random) versus the other components such as Section 6.1 and Section 6.2. This is especially true for distributed node collision resistance as defined by Section 6.3.

There are two example scenarios below which help illustrate the varying seriousness of a collision within an application.

Low Impact

A UUID collision generated a duplicate log entry which results in incorrect statistics derived from the data. Implementations that are not negatively affected by collisions may continue with the entropy and uniqueness provided by the traditional UUID format.

High Impact:

A duplicate key causes an airplane to receive the wrong course which puts people's lives at risk. In this scenario there is no margin for error. Collisions MUST be avoided and failure is unacceptable. Applications dealing with this type of scenario MUST employ as much collision resistance as possible within the given application context.

6.5. Global and Local Uniqueness

UUIDs created by this specification MAY be used to provide local uniqueness guarantees. For example, ensuring UUIDs created within a local application context are unique within a database MAY be sufficient for some implementations where global uniqueness outside of the application context, in other applications, or around the world is not required.

Although true global uniqueness is impossible to guarantee without a shared knowledge scheme; a shared knowledge scheme is not required by UUID to provide uniqueness guarantees. Implementations MAY implement a shared knowledge scheme introduced in Section 6.3 as they see fit to extend the uniqueness guaranteed this specification and [RFC4122].

6.6. Unguessability

Implementations SHOULD utilize a cryptographically secure pseudo-random number generator (CSPRNG) to provide values that are both difficult to predict ("unguessable") and have a low likelihood of collision ("unique"). CSPRNG ensures the best of Section 6.4 and Section 8 are present in modern UUIDs.

Advice on generating cryptographic-quality random numbers can be found in [RFC4086]

6.7. Sorting

UUIDv6 and UUIDv7 are designed so that implementations that require sorting (e.g. database indexes) SHOULD sort as opaque raw bytes, without need for parsing or introspection.

Time ordered monotonic UUIDs benefit from greater database index locality because the new values are near each other in the index. As a result objects are more easily clustered together for better performance. The real-world differences in this approach of index locality vs random data inserts can be quite large.

UUIDs formats created by this specification SHOULD be Lexicographically sortable while in the textual representation.

UUIDs created by this specification are crafted with big-ending byte order (network byte order) in mind. If Little-endian style is required a custom UUID format SHOULD be created using UUIDv8.

6.8. Opacity

UUIDs SHOULD be treated as opaque values and implementations SHOULD NOT examine the bits in a UUID to whatever extent is possible. However, where necessary, inspectors should refer to Section 4 for more information on determining UUID version and variant.

6.9. DBMS and Database Considerations

For many applications, such as databases, storing UUIDs as text is unnecessarily verbose, requiring 288 bits to represent 128 bit UUID values. Thus, where feasible, UUIDs SHOULD be stored within database applications as the underlying 128 bit binary value.

For other systems, UUIDs MAY be stored in binary form or as text, as appropriate. The trade-offs to both approaches are as such:

- * Storing as binary requires less space and may result in faster data access.
- * Storing as text requires more space but may require less translation if the resulting text form is to be used after retrieval and thus maybe simpler to implement.

DBMS vendors are encouraged to provide functionality to generate and store UUID formats defined by this specification for use as identifiers or left parts of identifiers such as, but not limited to, primary keys, surrogate keys for temporal databases, foreign keys included in polymorphic relationships, and keys for key-value pairs in JSON columns and key-value databases. Applications using a monolithic database may find using database-generated UUIDs (as opposed to client-generate UUIDs) provides the best UUID monotonicity. In addition to UUIDs, additional identifiers MAY be used to ensure integrity and feedback.

7. IANA Considerations

This document has no IANA actions.

8. Security Considerations

MAC addresses pose inherent security risks and SHOULD not be used within a UUID. Instead CSPRNG data SHOULD be selected from a source with sufficient entropy to ensure guaranteed uniqueness among UUID generation. See Section 6.6 for more information.

Timestamps embedded in the UUID do pose a very small attack surface. The timestamp in conjunction with an embedded counter does signal the order of creation for a given UUID and it's corresponding data but

does not define anything about the data itself or the application as a whole. If UUIDs are required for use with any security operation within an application context in any shape or form then [RFC4122] UUIDv4 SHOULD be utilized.

9. Acknowledgements

The authors gratefully acknowledge the contributions of Ben Campbell, Ben Ramsey, Fabio Lima, Gonzalo Salgueiro, Martin Thomson, Murray S. Kucherawy, Rick van Rein, Rob Wilton, Sean Leonard, Theodore Y. Ts'o., Robert Kieffer, sergeyprokhorenko, LiosK As well as all of those in the IETF community and on GitHub to who contributed to the discussions which resulted in this document.

10. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [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>>.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <<https://www.rfc-editor.org/info/rfc4122>>.
- [RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/info/rfc4122>>.

11. Informative References

- [LexicalUUID]
Twitter, "A Scala client for Cassandra", commit f6da4e0, November 2012, <<https://github.com/twitter-archive/cassie>>.
- [Snowflake]
Twitter, "Snowflake is a network service for generating unique ID numbers at high scale with some simple guarantees.", Commit b3f6a3c, May 2014, <<https://github.com/twitter-archive/snowflake/releases/tag/snowflake-2010>>.

- [Flake] Boundary, "Flake: A decentralized, k-ordered id generation service in Erlang", Commit 15c933a, February 2017, <<https://github.com/boundary/flake>>.
- [ShardingID] Instagram Engineering, "Sharding & IDs at Instagram", December 2012, <<https://instagram-engineering.com/sharding-ids-at-instagram-1cf5a71e5a5c>>.
- [KSUID] Segment, "K-Sortable Globally Unique IDs", Commit bf376a7, July 2020, <<https://github.com/segmentio/ksuid>>.
- [Elasticflake] Pearcy, P., "Sequential UUID / Flake ID generator pulled out of elasticsearch common", Commit dd71c21, January 2015, <<https://github.com/ppearcy/elasticflake>>.
- [FlakeID] Pawlak, T., "Flake ID Generator", Commit fcd6a2f, April 2020, <<https://github.com/T-PWK/flake-idgen>>.
- [Sonyflake] Sony, "A distributed unique ID generator inspired by Twitter's Snowflake", Commit 848d664, August 2020, <<https://github.com/sony/sonyflake>>.
- [orderedUuid] Cabrera, IT., "Laravel: The mysterious 'Ordered UUID'", January 2020, <<https://itnext.io/laravel-the-mysterious-ordered-uuid-29e7500b4f8>>.
- [COMBGUID] Tallent, R., "Creating sequential GUIDs in C# for MSSQL or PostgreSQL", Commit 2759820, December 2020, <<https://github.com/richardtallent/RT.Comb>>.
- [ULID] Feerasta, A., "Universally Unique Lexicographically Sortable Identifier", Commit d0c7170, May 2019, <<https://github.com/ulid/spec>>.
- [SID] Chilton, A., "sid : generate sortable identifiers", Commit 660e947, June 2019, <<https://github.com/chilts/sid>>.
- [pushID] Google, "The 2¹²⁰ Ways to Ensure Unique Identifiers", February 2015, <https://firebase.googleblog.com/2015/02/the-2120-ways-to-ensure-unique_68.html>.
- [XID] Poitrey, O., "Globally Unique ID Generator", Commit efa678f, October 2020, <<https://github.com/rs/xid>>.

- [ObjectID] MongoDB, "ObjectId - MongoDB Manual",
<<https://docs.mongodb.com/manual/reference/method/ObjectId/>>.
- [CUID] Elliott, E., "Collision-resistant ids optimized for horizontal scaling and performance.", Commit 215b27b, October 2020, <<https://github.com/ericelliott/cuid/>>.
- [IEEE754] IEEE, "Collision-resistant ids optimized for horizontal scaling and performance.", Series 754-2019, July 2019, <<https://standards.ieee.org/ieee/754/6210/>>.

Appendix A. Example Code

A.1. Creating a UUIDv6 Value

This section details a function in C which converts from a UUID version 1 to version 6:

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <arpa/inet.h>
#include <uuid/uuid.h>

/* Converts UUID version 1 to version 6 in place. */
void uuidv1tov6(uuid_t u) {

    uint64_t ut;
    unsigned char *up = (unsigned char *)u;

    // load ut with the first 64 bits of the UUID
    ut = ((uint64_t)ntohl(*(uint32_t*)up)) << 32;
    ut |= ((uint64_t)ntohl(*(uint32_t*)&up[4]));

    // dance the bit-shift...
    ut =
        ((ut >> 32) & 0x0FFF) | // 12 least significant bits
        (0x6000) | // version number
        ((ut >> 28) & 0x00000000FFFFFF0000) | // next 20 bits
        ((ut << 20) & 0x000FFFFF0000000000) | // next 16 bits
        (ut << 52); // 12 most significant bits

    // store back in UUID
    *((uint32_t*)up) = htonl((uint32_t)(ut >> 32));
    *((uint32_t*)&up[4]) = htonl((uint32_t)(ut));
}
```

Figure 6: UUIDv6 Function in C

A.2. Creating a UUIDv7 Value

```

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>

// ...

// csprng data source
FILE *rndf;
rndf = fopen("/dev/urandom", "r");
if (rndf == 0) {
    printf("fopen /dev/urandom error\n");
    return 1;
}

// ...

// generate one UUIDv7E
uint8_t u[16];
struct timespec ts;
int ret;

ret = clock_gettime(CLOCK_REALTIME, &ts);
if (ret != 0) {
    printf("clock_gettime error: %d\n", ret);
    return 1;
}

uint64_t tms;

tms = ((uint64_t)ts.tv_sec) * 1000;
tms += ((uint64_t)ts.tv_nsec) / 1000000;

memset(u, 0, 16);

fread(&u[6], 10, 1, rndf); // fill everything after the timestamp with random bytes

*((uint64_t*)(u)) |= htonll(tms << 16); // shift time into first 48 bits and OR into place

u[8] = 0x80 | (u[8] & 0x3F); // set variant field, top two bits are 1, 0
u[6] = 0x70 | (u[6] & 0x0F); // set version field, top four bits are 0, 1, 1, 1

```

Figure 7: UUIDv7 Function in C

A.3. Creating a UUIDv8 Value

UUIDv8 will vary greatly from implementation to implementation. A good candidate use case for UUIDv8 is to embed exotic timestamps like the one found in this example which employs approximately 0.25 milliseconds and approximately 5 microseconds per timestamp tick as a 48 bit value.

```
#include <stdint.h>
#include <stdio.h>
#include <time.h>

int main() {
    struct timespec tp;
    clock_gettime(CLOCK_REALTIME, &tp);
    uint64_t timestamp = (uint64_t)tp.tv_sec << 12;

    // compute 12 bit (~0.25 msec precision) fraction from nsecs
    timestamp |= ((uint64_t)tp.tv_nsec << 12) / 1000000000;

    printf("%08llx-%04llx\n", timestamp >> 16, timestamp & 0xFFFF);
    return 0;
}
```

Figure 8: UUIDv8 Function in C

Appendix B. Test Vectors

Both UUIDv1 and UUIDv6 test vectors utilize the same 60 bit timestamp: 0x1EC9414C232AB00 (138648505420000000) Tuesday, February 22, 2022 2:22:22.000000 PM GMT-05:00

Both UUIDv1 and UUIDv6 utilize the same values in `clk_seq_hi_res`, `clock_seq_low`, and `node`. All of which have been generated with random data.

```

# Unix Nanosecond precision to Gregorian 100-nanosecond intervals
gregorian_100_ns = (Unix_64_bit_nanoseconds / 100) + gregorian_Unix_offset

# Gregorian to Unix Offset:
# The number of 100-ns intervals between the
# UUID epoch 1582-10-15 00:00:00 and the Unix epoch 1970-01-01 00:00:00.
# gregorian_Unix_offset = 0x01b21dd213814000 or 1221929280000000000

# Unix 64 bit Nanosecond Timestamp:
# Unix NS: Tuesday, February 22, 2022 2:22:22 PM GMT-05:00
# Unix_64_bit_nanoseconds = 0x16D6320C3D4DCC00 or 1645557742000000000

# Work:
# gregorian_100_ns = (1645557742000000000 / 100) + 1221929280000000000
# (1386485054200000000 - 1221929280000000000) * 100 = Unix_64_bit_nanoseconds

# Final:
# gregorian_100_ns = 0x1EC9414C232AB00 or 1386485054200000000

# Original: 000111101100100101000000101001100000100011001010101011000000000
# UUIDv1:   1100001000110010101010101100000000|10010100000010100|0001|000111101100
# UUIDv6:   000111101100100101000000101001100|0010001100101010|0110|1011000000000

```

Figure 9: Test Vector Timestamp Pseudo-code

B.1. Example of a UUIDv6 Value

field	bits	value_hex
time_low	32	0xC232AB00
time_mid	16	0x9414
time_hi_and_version	16	0x11EC
clk_seq_hi_res	8	0xB3
clock_seq_low	8	0xC8
node	48	0x9E6BDECED846
total	128	
final_hex: C232AB00-9414-11EC-B3C8-9E6BDECED846		

Figure 10: UUIDv1 Example Test Vector

field	bits	value_hex
time_high	32	0x1EC9414C
time_mid	16	0x232A
time_low_and_version	16	0x6B00
clk_seq_hi_res	8	0xB3
clock_seq_low	8	0xC8
node	48	0x9E6BDECED846
total	128	
final_hex: 1EC9414C-232A-6B00-B3C8-9E6BDECED846		

Figure 11: UUIDv6 Example Test Vector

B.2. Example of a UUIDv7 Value

This example UUIDv7 test vector utilizes a well-known 32 bit Unix epoch with additional millisecond precision to fill the first 48 bits

rand_a and rand_b are filled with random data.

The timestamp is Tuesday, February 22, 2022 2:22:22.00 PM GMT-05:00 represented as 0x17F21CFD130 or 1645539742000

field	bits	value
unix_ts_ms	48	0x017F21CFD130
var	4	0x7
rand_a	12	0xCC3
var	2	b10
rand_b	62	0x18C4DC0C0C07398F
total	128	
final: 017F21CF-D130-7CC3-98C4-DC0C0C07398F		

Figure 12: UUIDv7 Example Test Vector

B.3. Example of a UUIDv8 Value

This example UUIDv8 test vector utilizes a well-known 64 bit Unix epoch with nanosecond precision, truncated to the least-significant, right-most, bits to fill the first 48 bits through version.

The next two segments of custom_b and custom_c are filled with random data.

Timestamp is Tuesday, February 22, 2022 2:22:22.000000 PM GMT-05:00 represented as 0x16D6320C3D4DCC00 or 1645557742000000000

It should be noted that this example is just to illustrate one scenario for UUIDv8. Test vectors will likely be implementation specific and vary greatly from this simple example.

field	bits	value
custom_a	48	0x320C3D4DCC00
ver	4	0x8
custom_b	12	0x75B
var	2	b10
custom_c	62	0xEC932D5F69181C0
total	128	
final: 320C3D4D-CC00-875B-8EC9-32D5F69181C0		

Figure 13: UUIDv8 Example Test Vector

Appendix C. Version and Variant Tables

C.1. Variant 10xx Versions

Msb0	Msb1	Msb2	Msb3	Version	Description
0	0	0	0	0	Unused
0	0	0	1	1	The Gregorian time-based UUID from in [RFC4122], Section 4.1.3
0	0	1	0	2	DCE Security version, with embedded POSIX UIDs from [RFC4122], Section 4.1.3
0	0	1	1	3	The name-based version specified in [RFC4122], Section 4.1.3 that uses MD5 hashing.
0	1	0	0	4	The randomly or pseudo-

					randomly generated version specified in [RFC4122], Section 4.1.3.
0	1	0	1	5	The name-based version specified in [RFC4122], Section 4.1.3 that uses SHA-1 hashing.
0	1	1	0	6	Reordered Gregorian time-based UUID specified in this document.
0	1	1	1	7	Unix Epoch time-based UUID specified in this document.
1	0	0	0	8	Reserved for custom UUID formats specified in this document.
1	0	0	1	9	Reserved for future definition.
1	0	1	0	10	Reserved for future definition.
1	0	1	1	11	Reserved for future definition.
1	1	0	0	12	Reserved for future definition.
1	1	0	1	13	Reserved for future definition.
1	1	1	0	14	Reserved for future definition.
1	1	1	1	15	Reserved for future definition.

Table 2: All UUID variant 10xx (8/9/A/B) version definitions.

Authors' Addresses

Brad G. Peabody

Email: brad@peabody.io

Kyzer R. Davis

Email: kydavis@cisco.com

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).

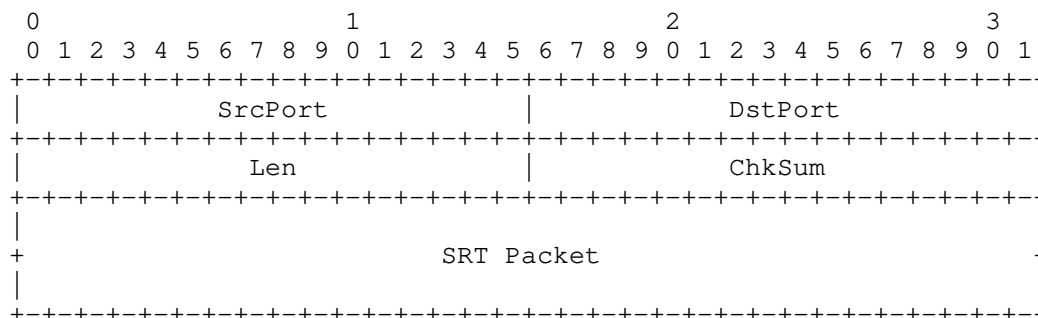


Figure 1: SRT packet as UDP payload

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.

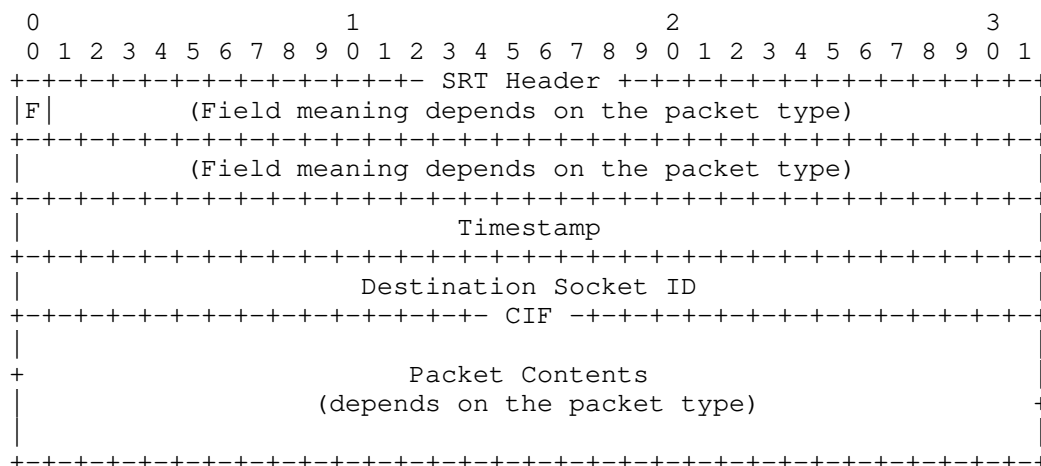


Figure 2: SRT packet structure

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.

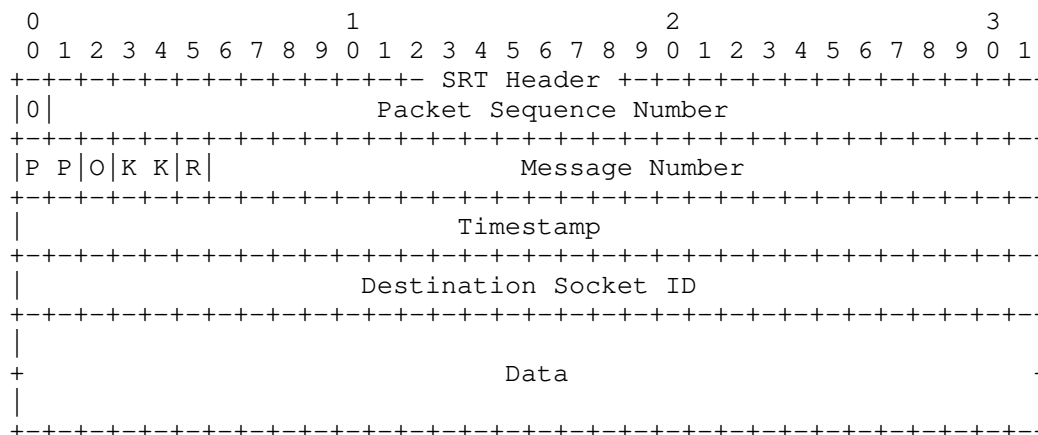


Figure 3: data packet structure

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".

0 (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.

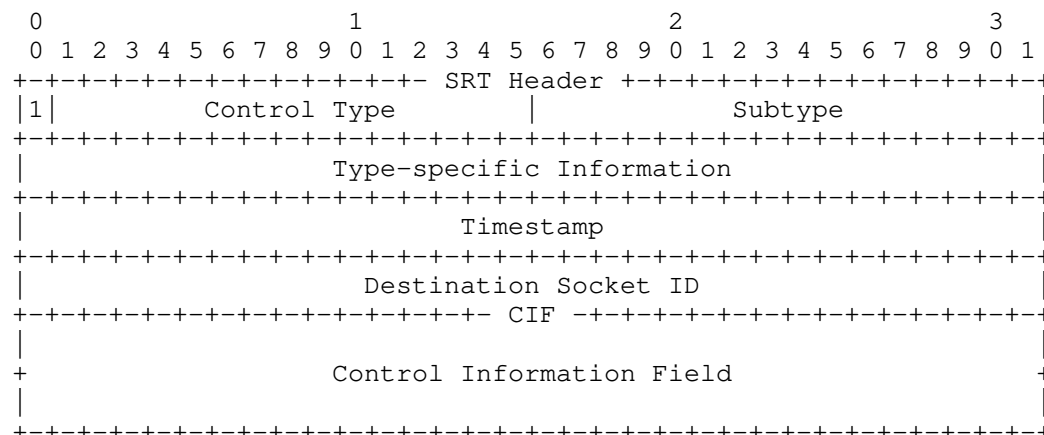


Figure 4: control packet structure

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.

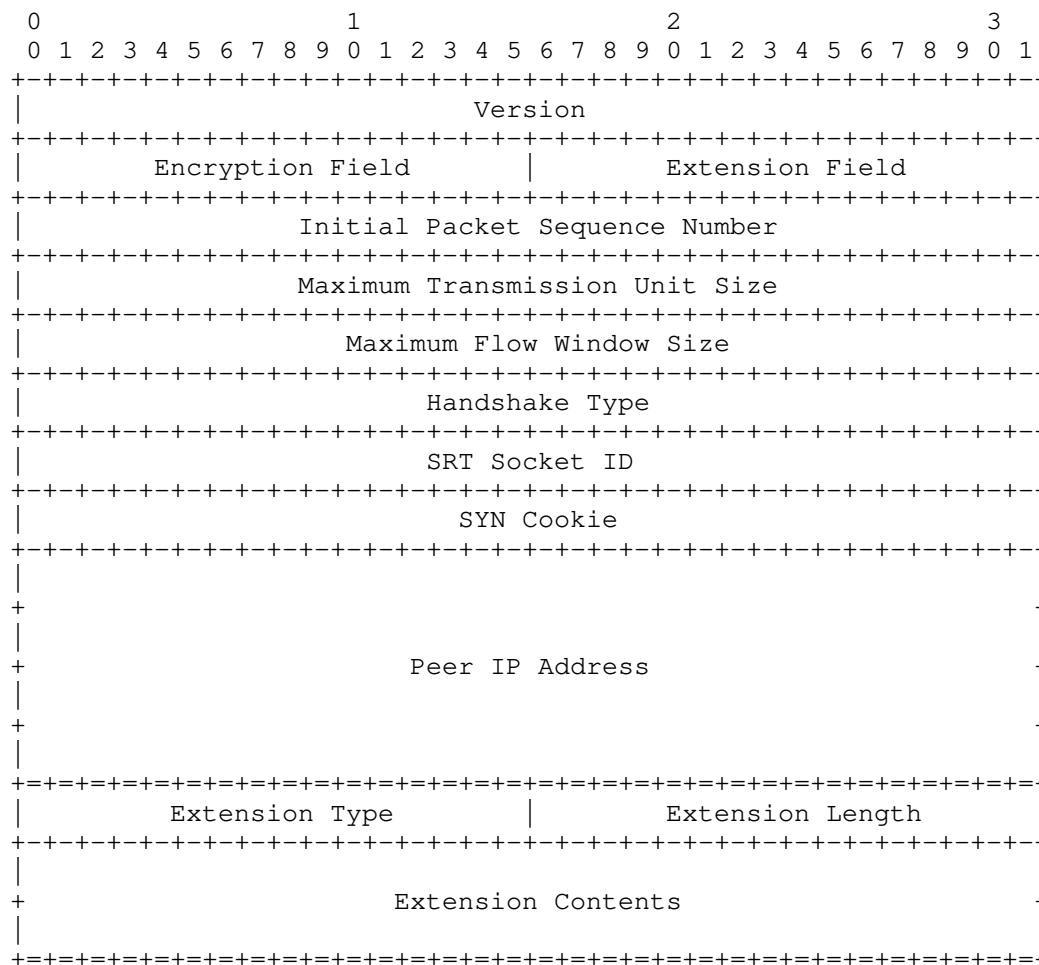


Figure 5: handshake packet structure

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:

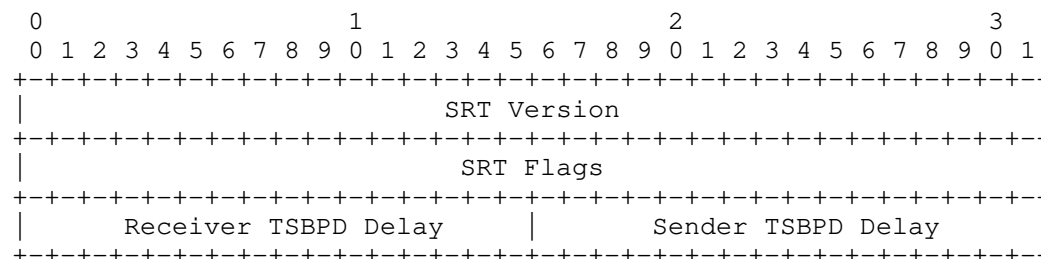


Figure 6: Handshake Extension Message structure

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.

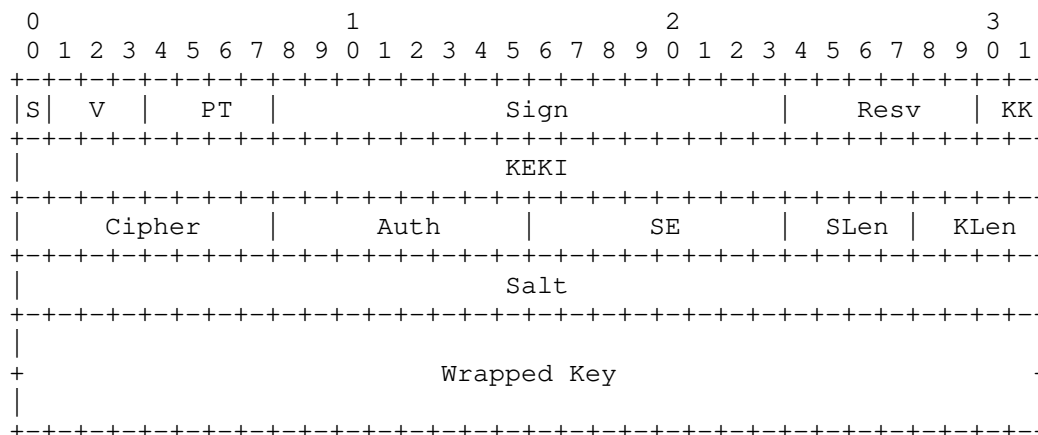


Figure 7: Key Material Extension structure

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)

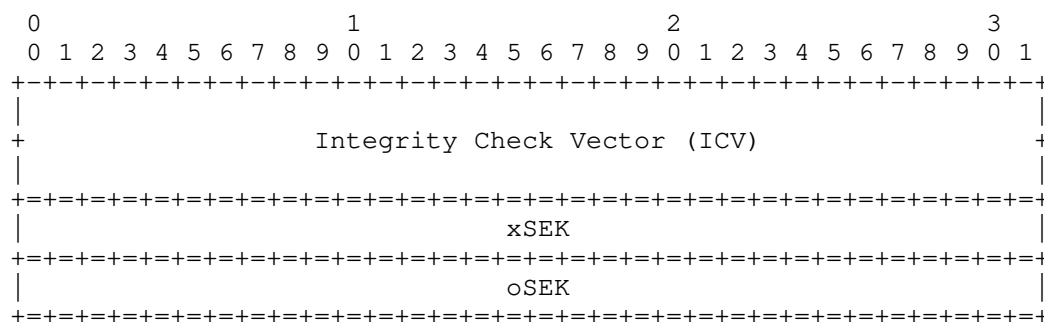


Figure 8: Unwrapped key structure

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:

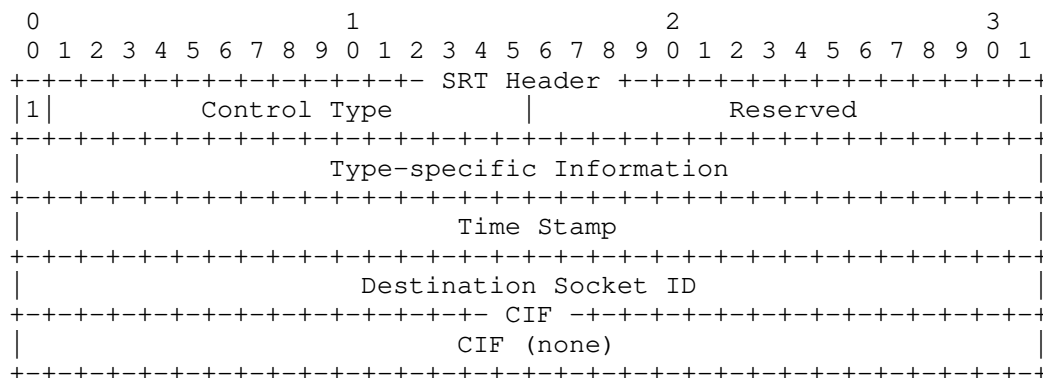


Figure 9: Keep-Alive structure

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:

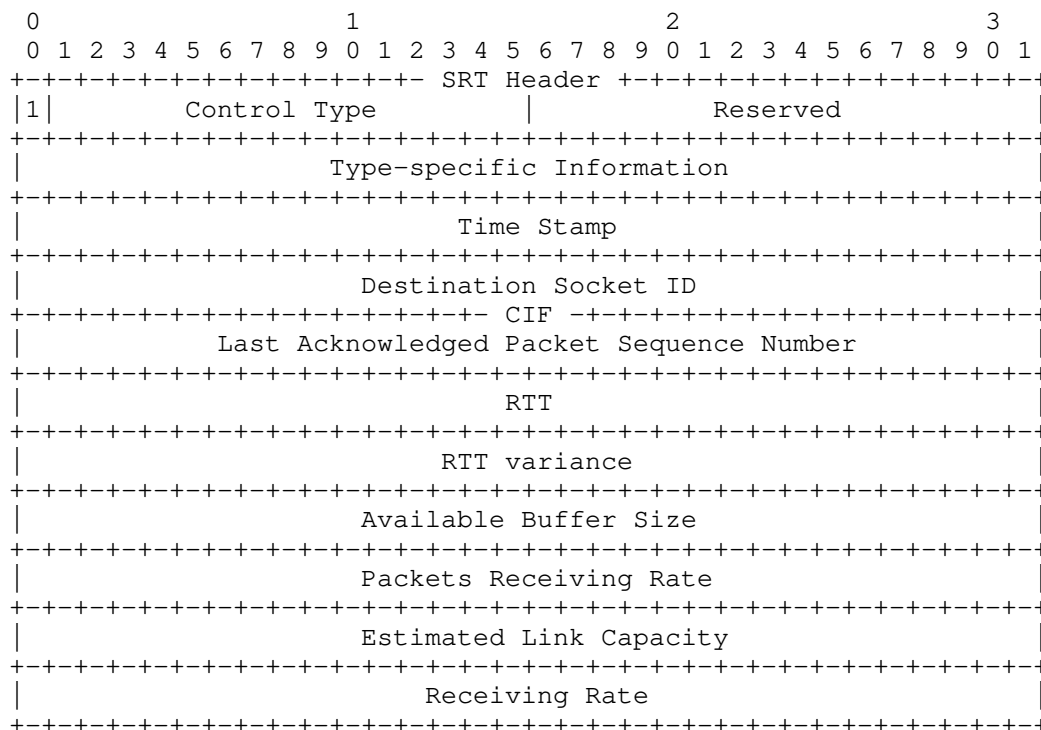


Figure 10: ACK control packet

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:

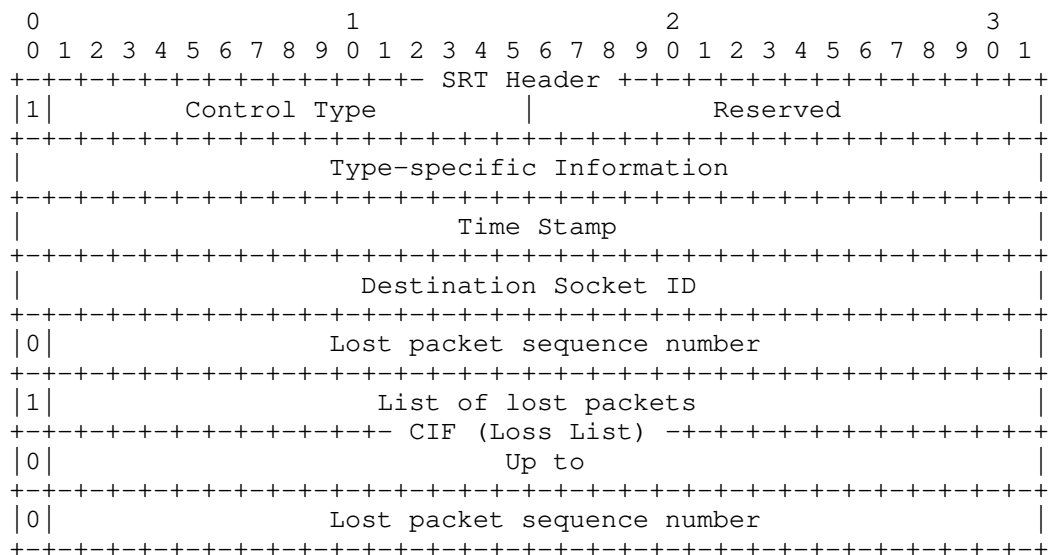


Figure 11: NAK control packet

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:

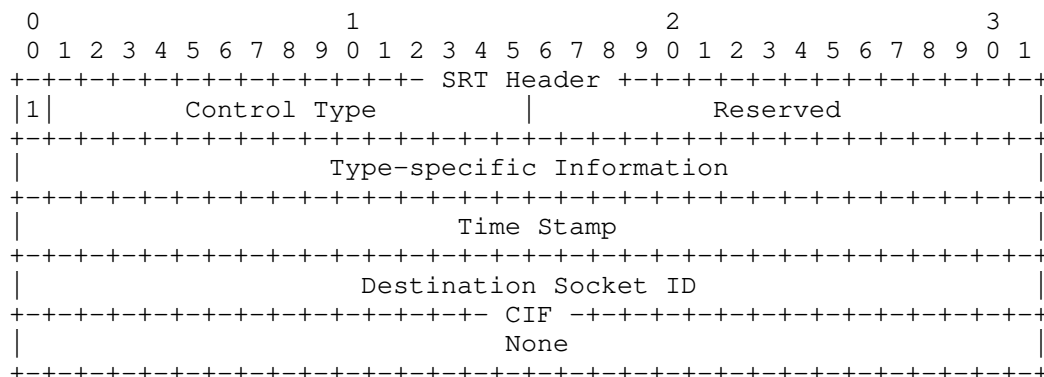


Figure 12: SHUTDOWN control packet

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:

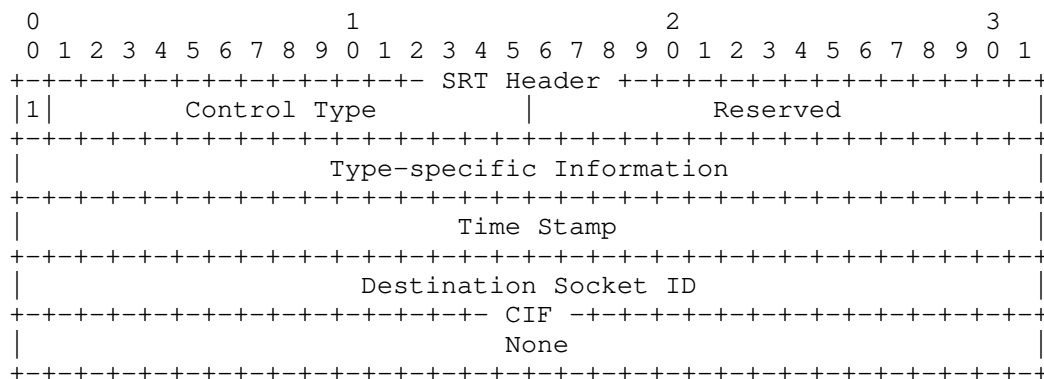


Figure 13: ACKACK control packet

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.

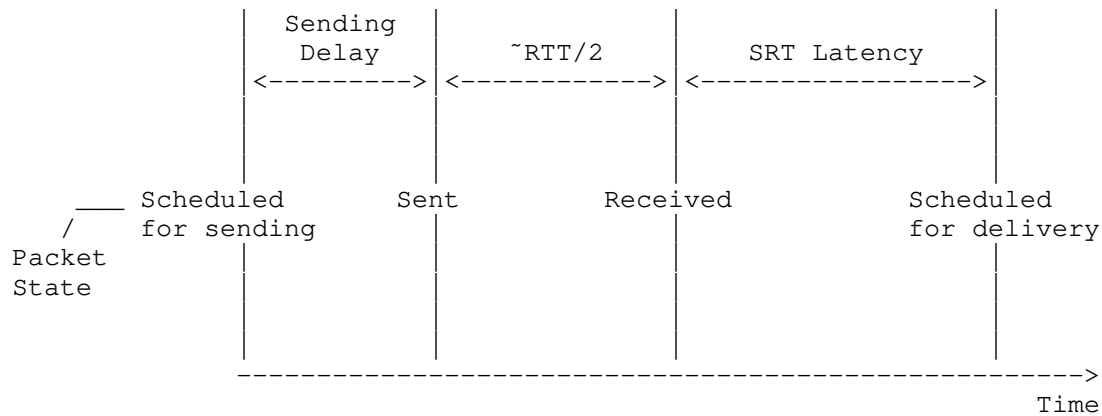


Figure 14: Key Latency Points during the Packet Transmission

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 Diousskine, 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, 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, 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, DOI 10.17487/RFC3031, January 2001, <<https://www.rfc-editor.org/info/rfc3031>>.

[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>>.

[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".

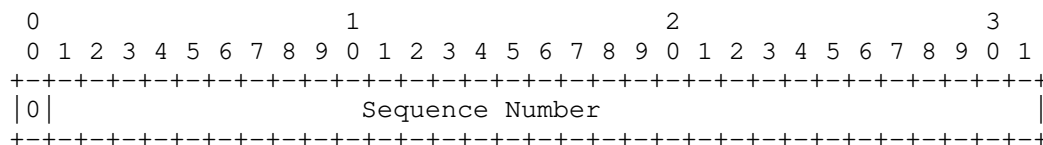


Figure 15: single sequence numbers coding

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".

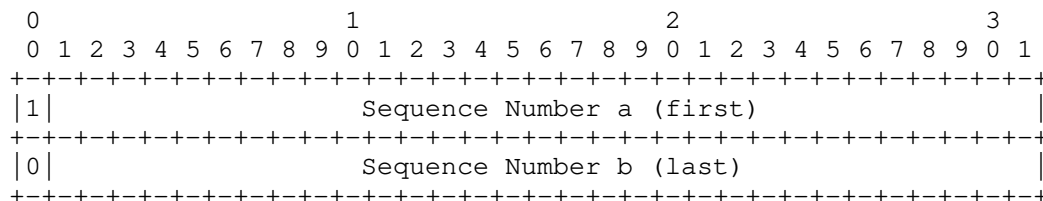


Figure 16: list of sequence numbers coding

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

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

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 13 March 2021.

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. Terms and Definitions	5
3. Packet Structure	6
3.1. Data Packets	7
3.2. Control Packets	8
3.2.1. Handshake	9
3.2.2. Key Material	17
3.2.3. Keep-Alive	21
3.2.4. ACK (Acknowledgment)	22
3.2.5. NAK (Loss Report)	24
3.2.6. Shutdown	25
3.2.7. ACKACK	26
4. SRT Data Transmission and Control	26
4.1. Stream Multiplexing	27
4.2. Data Transmission Modes	27
4.2.1. Message Mode	27
4.2.2. Live Mode	28
4.2.3. Buffer Mode	28
4.3. Handshake Messages	28
4.3.1. Caller-Listener Handshake	31
4.3.2. Rendezvous Handshake	33
4.4. SRT Buffer Latency	39
4.5. Timestamp-Based Packet Delivery	40
4.5.1. Packet Delivery Time	42
4.6. Too-Late Packet Drop	43
4.7. Drift Management	44
4.8. Acknowledgement and Lost Packet Handling	46
4.8.1. Packet Acknowledgement (ACKs, ACKACKs)	46
4.8.2. Packet Retransmission (NAKs)	47
4.9. Bidirectional Transmission Queues	49
4.10. Round-Trip Time Estimation	49
4.11. Congestion Control	50
5. Encryption	50
5.1. Overview	51
5.1.1. Encryption Scope	51
5.1.2. AES Counter	51
5.1.3. Stream Encrypting Key (SEK)	51
5.1.4. Key Encrypting Key (KEK)	52

5.1.5. Key Material Exchange	52
5.1.6. KM Refresh	53
5.2. Encryption Process	54
5.2.1. Generating the Stream Encrypting Key	54
5.2.2. Encrypting the Payload	54
5.3. Decryption Process	54
5.3.1. Restoring the Stream Encrypting Key	55
5.3.2. Decrypting the Payload	55
6. Security Considerations	56
7. IANA Considerations	56
Contributors	56
Acknowledgments	56
References	56
Normative References	56
Informative References	57
Appendix A. Packet Sequence List Coding	59
Appendix B. SRT Access Control	60
B.1. General Syntax	60
B.2. Standard Keys	61
B.3. Examples	62
Appendix C. Changelog	62
C.1. Since Version 00	62
Authors' Addresses	63

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).

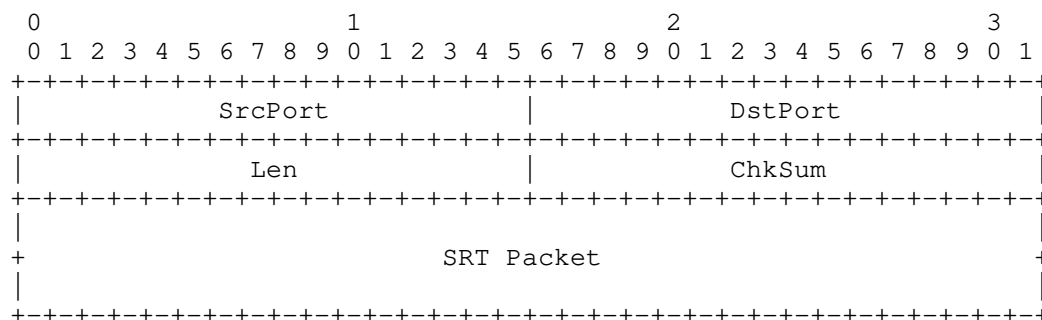


Figure 1: SRT packet as UDP payload

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.

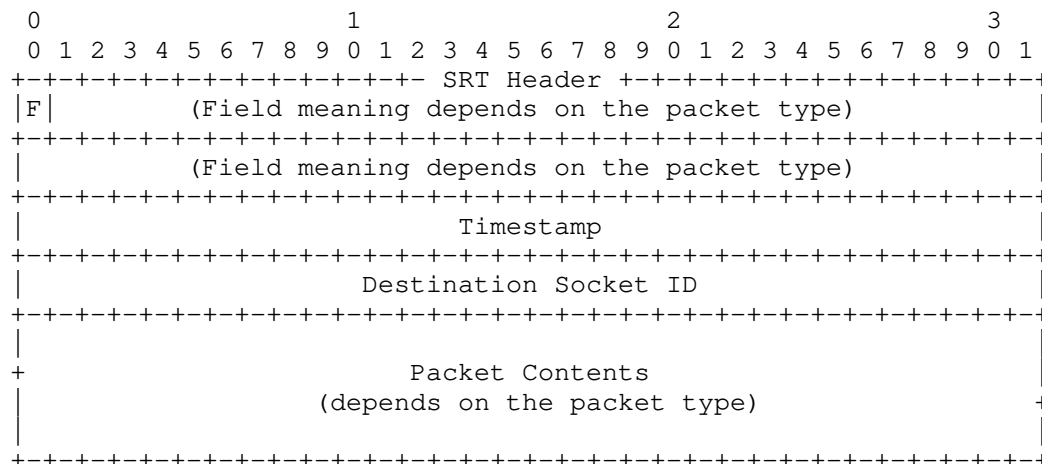


Figure 2: SRT packet structure

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 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.

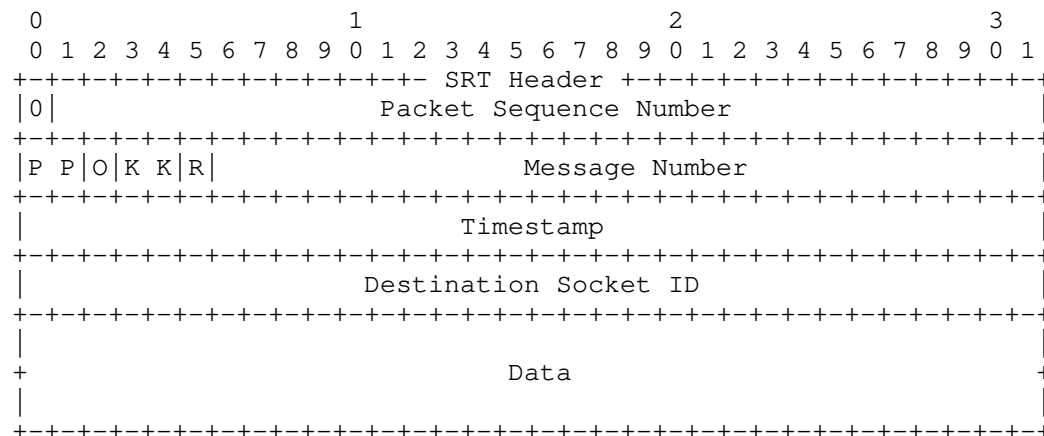


Figure 3: Data packet structure

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.

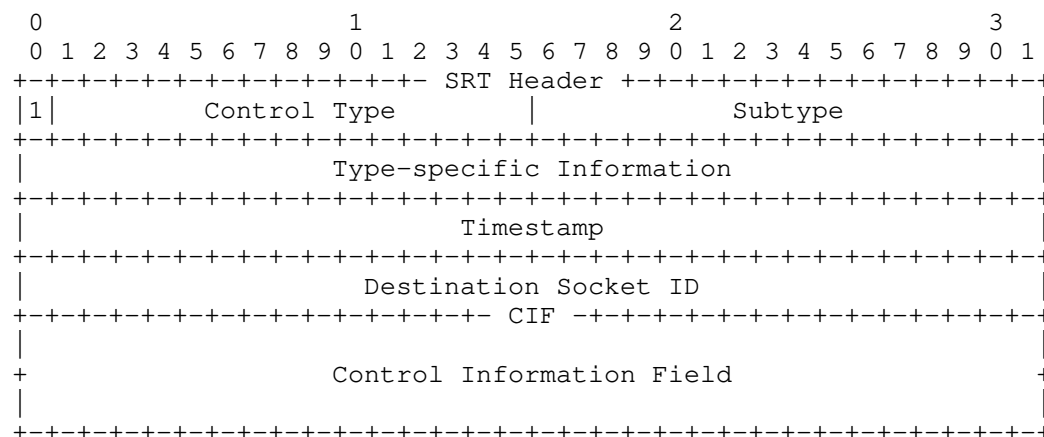


Figure 4: Control packet structure

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.

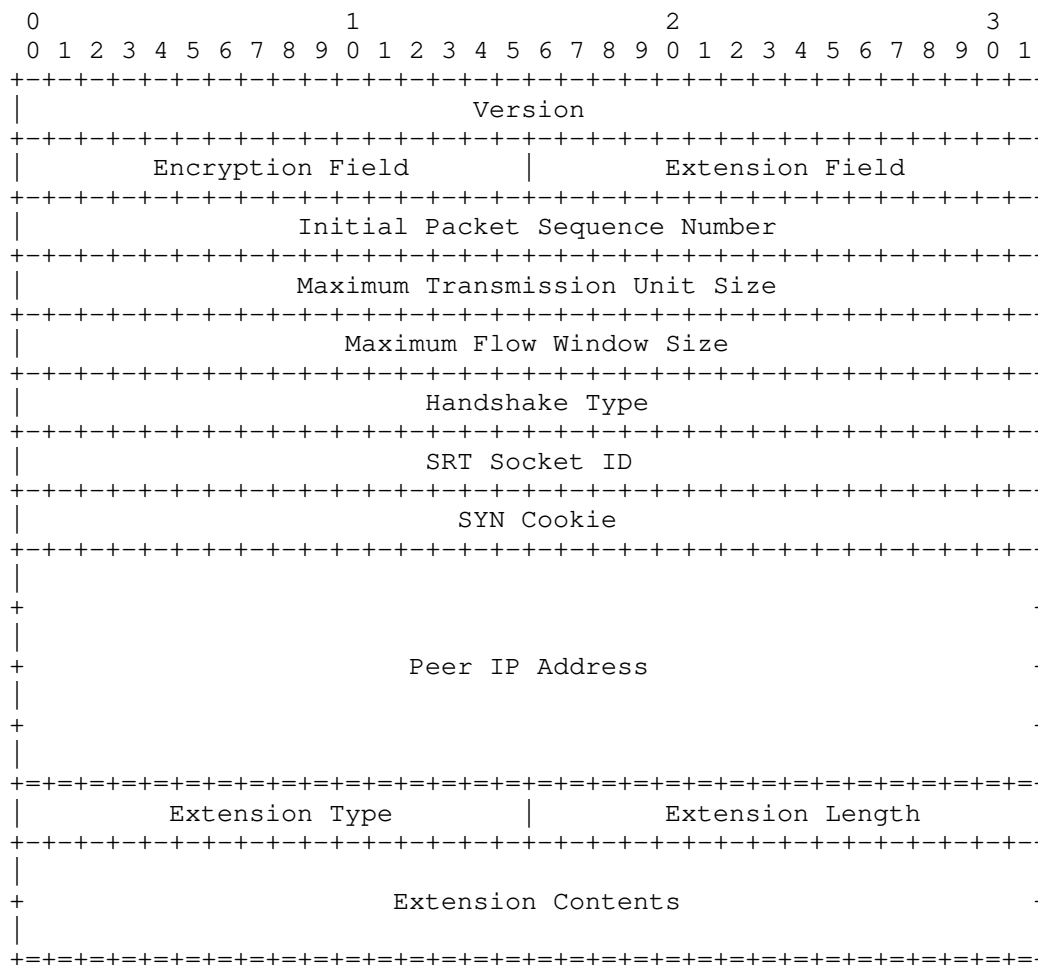


Figure 5: Handshake packet structure

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
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. 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:

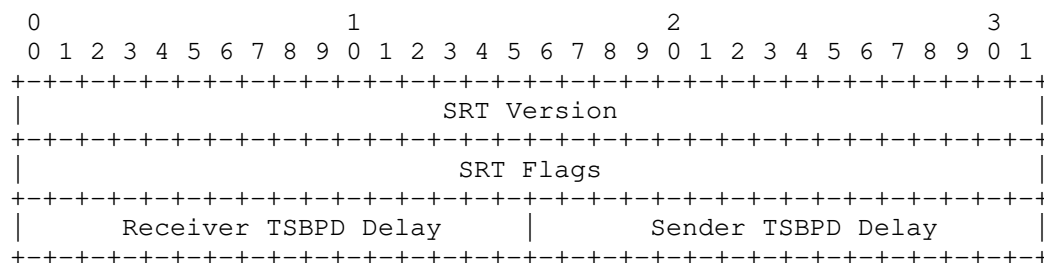


Figure 6: Handshake Extension Message structure

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.

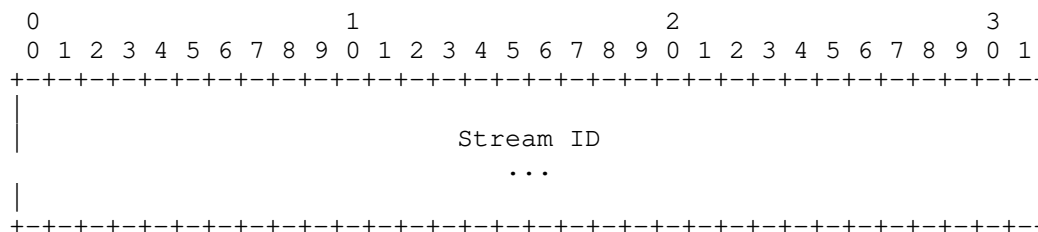


Figure 7: Stream ID Extension Message

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).

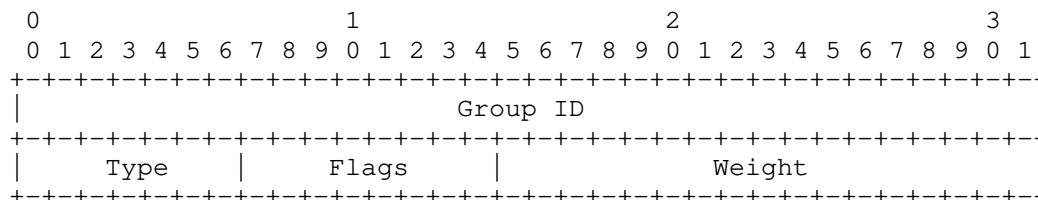


Figure 8: Group Membership Extension Message

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.

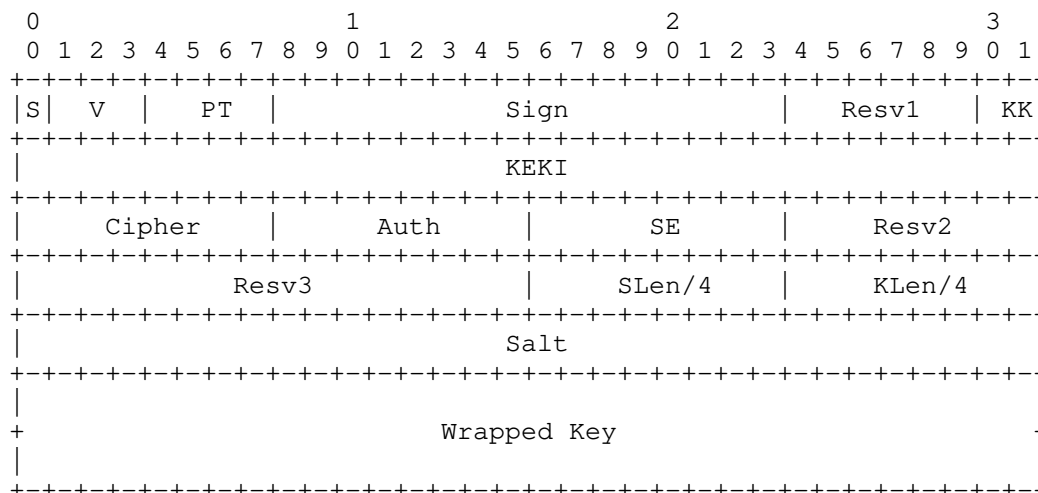


Figure 10: Key Material Message structure

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.

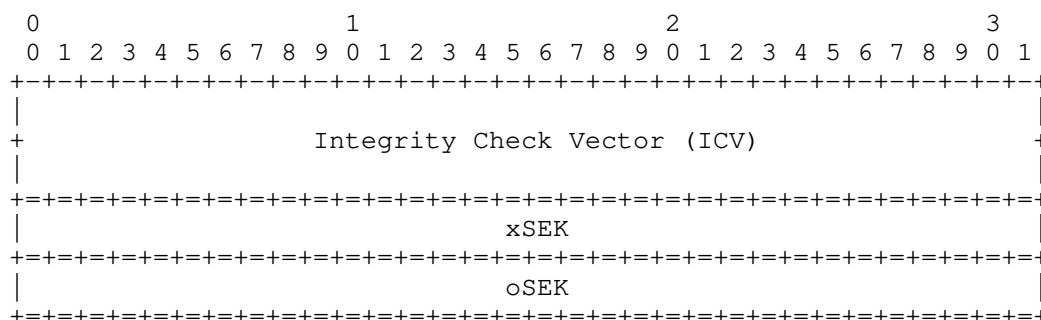


Figure 11: Unwrapped key structure

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 the 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:

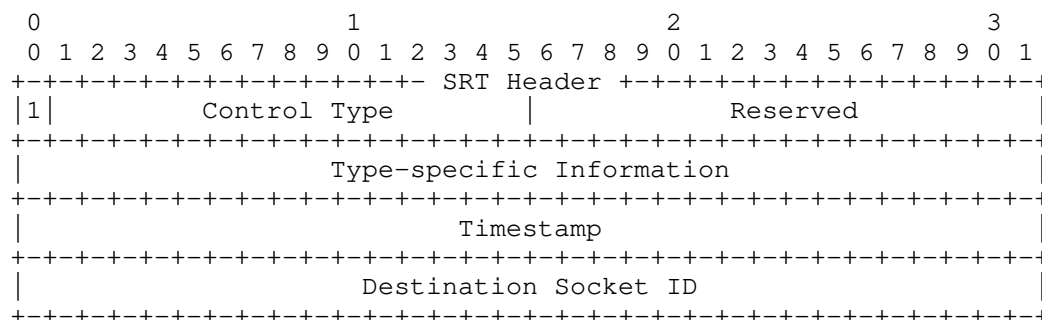


Figure 12: Keep-Alive control packet

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 acknowledged 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), preceeding 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:

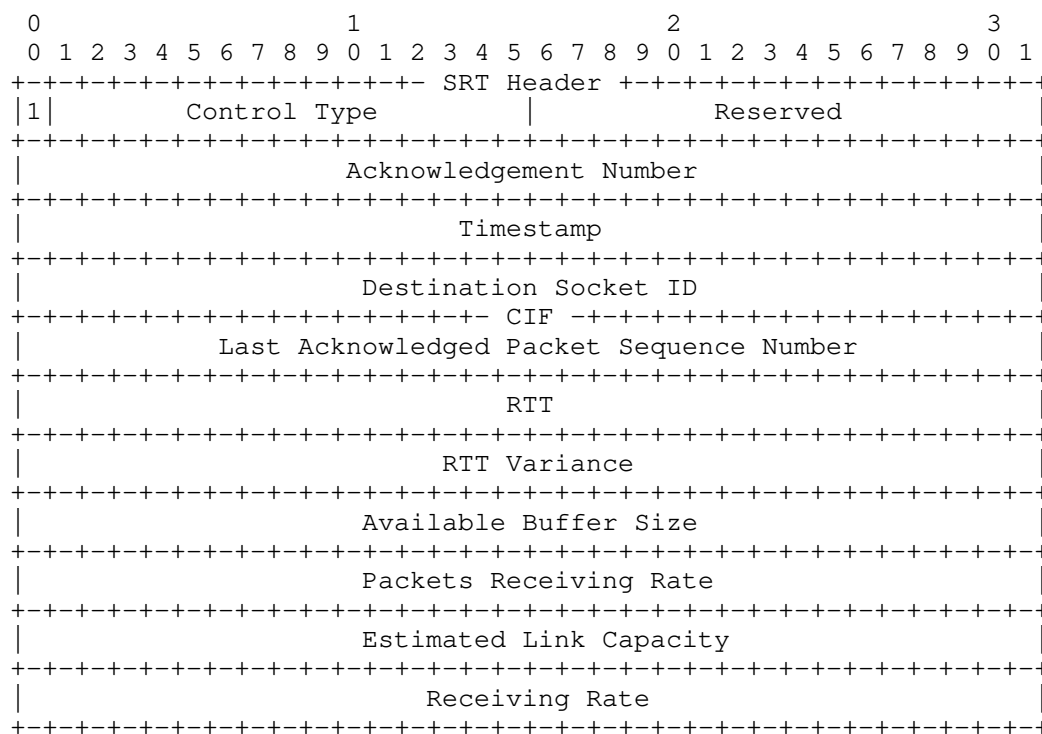


Figure 13: ACK control packet

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:

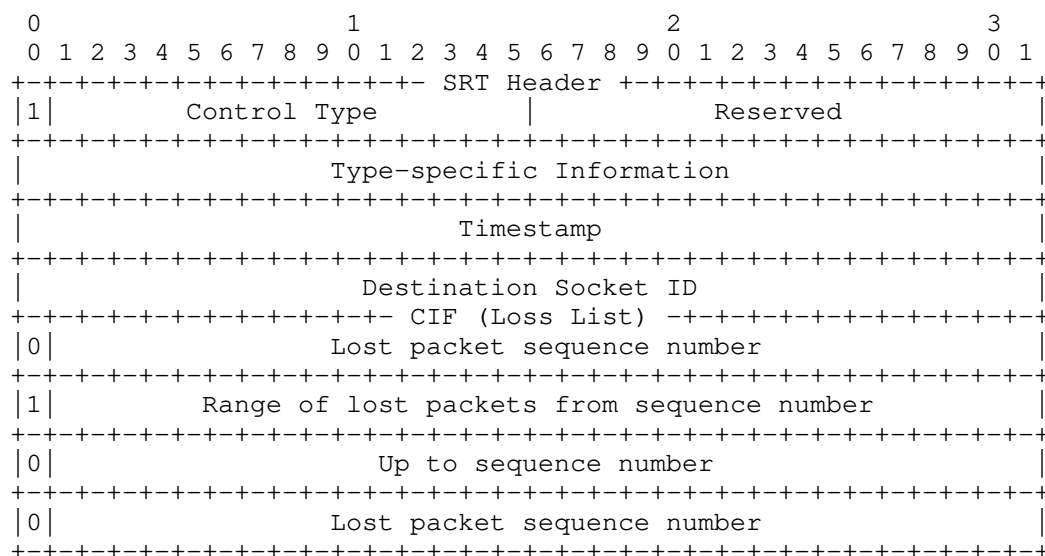


Figure 14: NAK control packet

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:

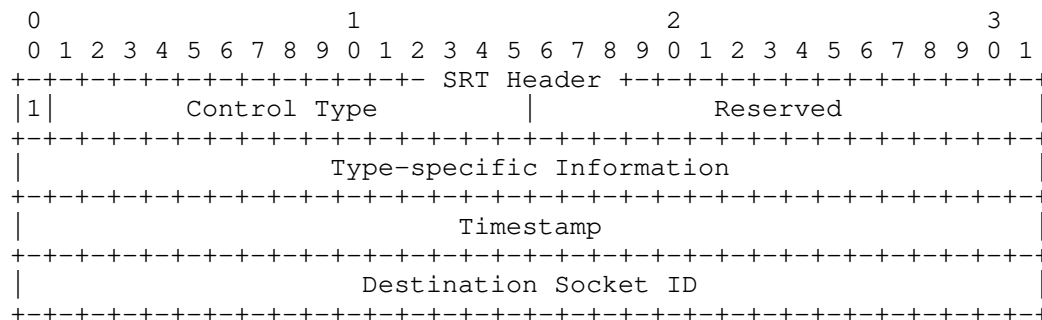


Figure 15: Shutdown control packet

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:

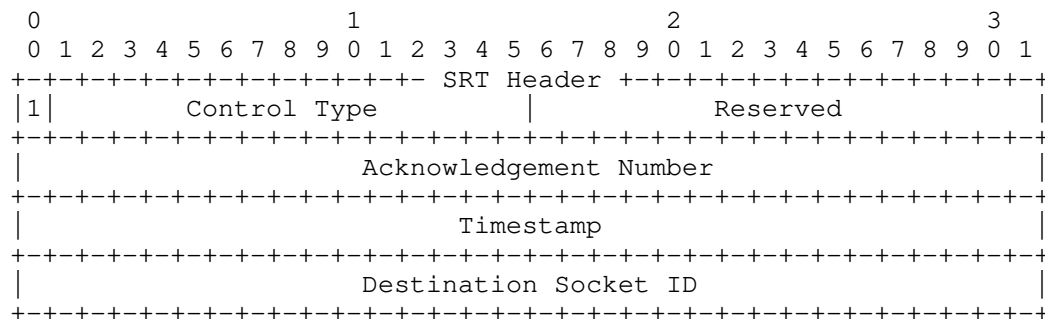


Figure 16: ACKACK control packet

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.

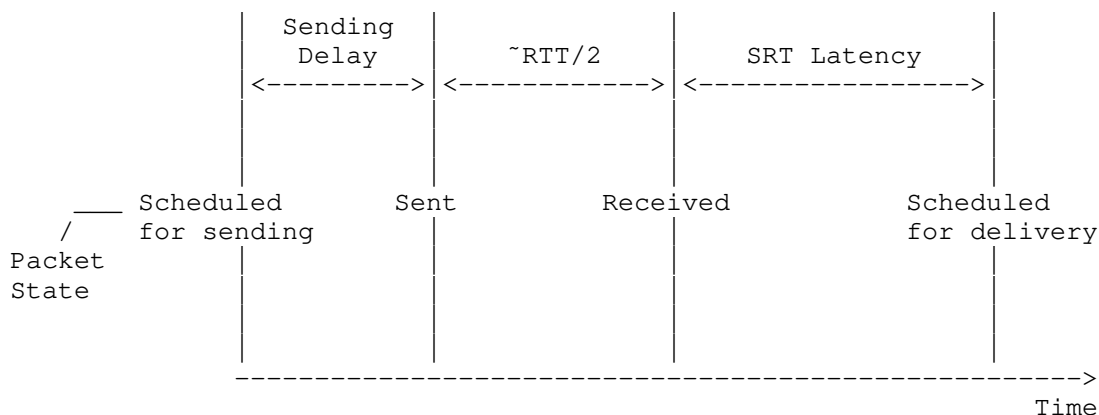


Figure 17: Key Latency Points during the Packet Transmission

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 $(RTT_0/2 + \text{SRT Latency})$ 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:

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

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:

$$RTTVar = RTTVar * 0.75 + \text{abs}(RTT - rtt) * 0.25$$

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^{25} th 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,
- * $\text{LSB}(n, v)$ 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:

$$\text{KEK} = \text{PBKDF2}(\text{passphrase}, \text{LSB}(64, \text{Salt}), \text{Iter}, \text{KLen})$$

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,
- * $\text{LSB}(n, v)$ 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.

$$\text{SEK} = \text{AESkuw}(\text{KEK}, \text{Wrap})$$

where $\text{AESkuw}(\text{KEK}, \text{Wrap})$ 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

$$\text{DecryptedPayload} = \text{AES_CTR_Encrypt}(\text{SEK}, \text{IV}, \text{EncryptedPayload})$$

where the Initialization Vector is derived as

$$\text{IV} = (\text{MSB}(112, \text{Salt}) \ll 2) \text{ XOR } (\text{PktSeqNo})$$

- * 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".

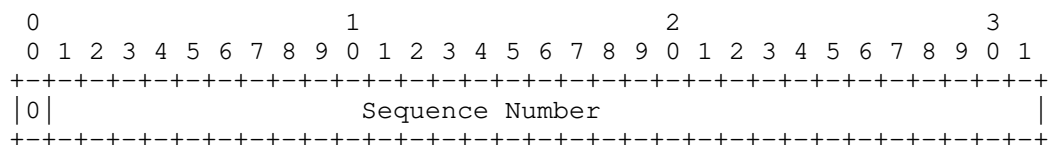


Figure 18: Single sequence numbers coding

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".

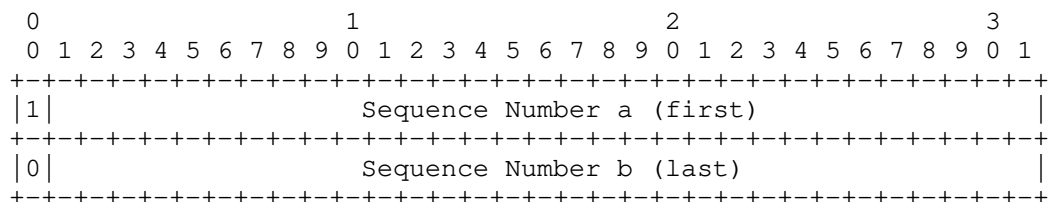


Figure 19: Range of sequence numbers coding

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/query.php?vid=366` the hostname field should have `somehost.com`, and the resource name can have `videos/query.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

INTERNET-DRAFT
<draft-storey-smtp-client-id-12.txt>
Intended Status: Standards Track
Expires May 24, 2022

W. Storey
LinuxMagic

November 24, 2021

SMTP Service Extension for Client Identity
<draft-storey-smtp-client-id-12.txt>

Abstract

This document defines an extension for the Simple Mail Transfer Protocol (SMTP) called "CLIENTID" to provide a method for clients to indicate an identity to the server.

This identity is an additional token that may be used for security and/or informational purposes, and with it a server may optionally apply heuristics using this token.

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), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/lid-abstracts.html>

The list of Internet-Draft Shadow Directories can be accessed at
<http://www.ietf.org/shadow.html>

Copyright Notice

Copyright (c) 2016 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 (<http://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
2. The CLIENTID Service Extension	4
3. The CLIENTID Keyword of the EHLO Command	4
4. The CLIENTID Command	4
5. Formal Syntax	5
6. Discussion	6
6.1. Applying heuristics to CLIENTID	6
6.2. Utility of CLIENTID	6
6.3. Use Cases of CLIENTID	8
6.4. Other SMTP Client Identifiers	9
6.5. Future Considerations	9
7. Client Identity Types	9
8. Examples	11
8.1 UUID Address as Client Identity	11
8.2 Client Identity Without a TLS/SSL Session	12
8.3 Client Identity Leading to Rejection	12
8.4 Malformed CLIENTID Command	13
9. Security Considerations	13
10. IANA Considerations	13
10.1 SMTP Extension Registration	14
11. References	14
11.1 Normative References	14
Appendix A. CLIENTID Product Support	14
Contributors	15
Authors' Addresses	15

1. Introduction

The [SMTP] protocol and its extensions describe methods whereby an SMTP client may provide identity and/or authentication information to an SMTP server. However, these existing methods are subject to limitations and none offer a way to identify the SMTP client with absolute confidence. This document defines an SMTP service extension to provide an additional identity token which can represent the SMTP client with a higher degree of certainty when accessing the SMTP server.

Typically SMTP clients are identified by establishing an authorized connection using the [AUTH] SMTP extension. SMTP servers are often subject to malicious clients attempting to use authorized identities not intended for their use (often referred to as a brute-force attack). When such an attack is attempted, the SMTP server may be unable to identify the impersonation and restrict such an unintended use by someone other than the authorized user of said credentials. While there are ways to identify the source of the SMTP client such as its IP address or EHLO identity, it would be useful if there was an additional way to uniquely identify the client in a method solely available across an encrypted channel.

Using the CLIENTID extension, an SMTP client can provide an additional identity token to the server called its "client identity".

The client identity can provide unique characteristics about the client accessing the SMTP service and may be combined with existing identification mechanisms in order to identify the client. An SMTP server may then apply additional security policies using this identity such as restricting use of the service to clients presenting recognized client identities, or only allowing use of authorized identities that match previously established client identities.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [KEYWORDS].

2. The CLIENTID Service Extension

The following SMTP service extension is hereby defined:

1. The name of this [SMTP] service extension is "Client Identity".
2. The EHLO keyword value associated with this extension is "CLIENTID".
3. The CLIENTID keyword has no parameters.
4. A new [SMTP] verb "CLIENTID" is defined.
5. No parameter is added to any SMTP command.
6. This extension is appropriate for the submission protocol [SUBMIT].

3. The CLIENTID Keyword of the EHLO Command

The CLIENTID keyword is used to tell the SMTP client that the SMTP server supports the CLIENTID service extension. Though certain conditions must be met before the CLIENTID keyword can be advertised.

1. An SMTP server MUST NOT advertise the CLIENTID keyword in any EHLO responses if the CLIENTID extension support is not enabled.
2. An SMTP server MUST NOT advertise the CLIENTID keyword in any EHLO response if the connection is not encrypted.
3. An SMTP server MUST advertise the CLIENTID keyword in all EHLO responses after the connection is successfully encrypted (if CLIENTID is supported).

4. The CLIENTID Command

The format for the CLIENTID command is:

CLIENTID client-id-type client-id-token

Arguments:

client-id-type: A string identifying the identity type the client is providing. It MUST be between 1 and 16 characters and comprised of only alphanumeric and dash characters.

client-id-token: A string identifying the client. It MUST be between 1 and 128 printable characters.

Restrictions:

An SMTP client MUST NOT issue a CLIENTID command unless a TLS/SSL session has been negotiated as described in [STARTTLS] or through other means such as over a historical SMTP-SSL connection. An SMTP server MUST reject any CLIENTID command sent before establishing an encrypted connection with a 500 reply.

An SMTP client MUST only issue the CLIENTID command after the SMTP server advertises the CLIENTID keyword via an EHLO command. An SMTP server MUST reject a CLIENTID command prior to advertising the CLIENTID keyword via an EHLO command.

An SMTP server MUST reject any CLIENTID command that is not well formatted with a 501 reply.

An SMTP client MUST NOT issue any subsequent CLIENTID commands after a successful CLIENTID command in the same session. An SMTP server MUST reject any subsequent CLIENTID commands after a successful CLIENTID command in the same session with a 503 reply.

An SMTP client MUST issue any CLIENTID commands prior to issuing an [AUTH] command. An SMTP server MUST reject any CLIENTID command after receiving an [AUTH] command with a 503 reply.

Several SMTP service extensions such as [AUTH] require that an SMTP session be reset to an initial state under conditions such as after applying a security layer. An SMTP server MUST discard any CLIENTID information after such a reset.

5. Formal Syntax

The following syntax specification uses the Augmented Backus-Naur Form notation as specified in [ABNF]. Non-terminals referenced but not defined below are as defined by [ABNF].

Except as noted otherwise, all alphabetic characters are case-insensitive.

client-id-type-char = ALPHA / DIGIT / "-"
 ;; alphanumeric and dash character

client-id-type = 1*16 client-id-type-char


```
client-id-token      = 1*128 VCHAR  
                      ;; any printable US-ASCII character
```

6. Discussion

6.1 Applying heuristics to CLIENTID

This section discusses the possible heuristics that can be applied to the information that is presented via the CLIENTID command. This information includes whether a valid CLIENTID command was issued, the client identity type and the client identity token.

1. An SMTP server MAY choose to require that a successful CLIENTID command be issued, or that a particular client type be presented before processing or accepting an authentication request.
2. An SMTP server MAY reject any authentication request not preceded with a client identity type that matches ACL's or rules as defined in the SMTP server.
3. An SMTP server MAY reject any authentication request preceded by a CLIENTID command that contains a client identity type or client identity token that the server chooses not to accept for any reason such as by policy.
4. An SMTP server MAY reject any authentication request preceded by a CLIENTID command that contains a client identity type or client identity token that the server has chosen to disable or revoke use of either temporarily or permanently.
5. An SMTP server MAY reject any authentication request where the provided client identity is not on the list of permitted clients for the account holder.

The SMTP server SHOULD only ever reject an SMTP client based on CLIENTID information during or after the authentication process/handler. In the interest of limiting the amount of information being revealed, the rejection message SHOULD be as generic as possible and SHOULD NOT reveal any information on the heuristics or rules on which it bases its decisions.

Even if the client identity type and/or client identity token are not recognized, supported or permitted by the server and/or the owner of the authentication credentials, the presented information may still be useful for analysis.

6.2 Utility of CLIENTID

Regardless of how frowned upon, users commonly reuse authorization information (like the username and password pair) across multiple services. When one service is compromised, malicious actors can also gain access to other services where the user also used the same credentials. Based on this representative problem alone, the utility of CLIENTID as an additional layer of determining the rights to

present such authorization information becomes quickly apparent.

The utility of CLIENTID may be seen by considering the following:

1. An SMTP client may be present on a device that does not have a useful domain name or network address, such as a mobile device, so its EHLO identity may be ambiguous.
2. An SMTP client may utilize the same SMTP server with multiple different authorized identities, so an identity that persists across authorized identities is lacking.
3. An authorized identity may make use of multiple discrete devices over different SMTP sessions, so an identity persisting on one device is lacking.
4. The SMTP DATA payload does not need to be inspected for this identity.
5. Connection information, a type of identity, such as network address frequently changes.

However, this extends beyond just the restriction of authentication. While it might be argued that this can be served as a special form of SASL, by implementing this in the SMTP service itself, the SMTP service can choose before allowing a connection to be passed to a SASL implementation, allowing it to perform other heuristics, such as identifying brute force attacks more effeciently.

The recent evolution of the internet as a whole has brought about large scale data breaches, compromised botnets comprising of millions of nodes, the transition to Carrier Grade NAT and the flourishing of IoT devices means traditional methods of protecting against brute force attacks have become much more difficult. Traditional methods such as rate limiting and/or blocking access by IP are no longer viable without introducing collateral effects, such as either blocking legitimate user, or creating the conditions that allow DOS (Denial of Service) to legitimate users.

Historically, SMTP and other services used what is technically a two factor, the email/user and password, albeit not effective in that the email is a KNOWN value. And with the propensity for users to use a simple password, and the hundreds of millions of email addresses exposed in data breaches, or available by other means the ability to brute force is quite simple. While of course it is recommended that users use longer and more secure passwords, this is not the de facto situation, and the threats when credentials get compromised are significant. And with 'botnet' operators able to engage millions of IoT in a distributed brute force, the status quo is in danger. Adding another non-public factor to be used as part of access control adds a strength against brute force by many factors and accomplishes it in a backwards compatible fashion, encouraging adoption.

Under brute force attacks, rate limiting or blocking by IP Address

was possible with little damage. But with the proliferation of IoT devices, smart phones, and the run-out of IP space, we have conditions where thousands of devices could be behind an IP Address, or IP(s) that are dynamic with devices changing IP(s) in minutes, blocking or rate limiting an IP bears risks of blocking legitimate users. By implementing a level of uniqueness to a connecting device, it introduces the ability to restrict or block a subset from connecting to the service for either brute force or dictionary attacks, while still allowing other devices to continue to be able to present authentication successfully.

While 'forgery' and/or the use of random client identifier is possible, such behavior is also more readily detectable when a device identifier is presented.

1. The SMTP server, when faced with hundreds of devices behind the same IP address, during an attack can restrict authentication attempts to only connections presenting a valid client identifier token.
2. The SMTP server, during an attack, can restrict authentication to only historically known devices.
3. The SMTP server can differentiate between many different devices behind the same IP, and apply maximum connections per device, rather than maximum connections per IP.
4. While a person may present authentication credentials from many different geographical locations, eg, home, office, and travel, a single device will not in general be able to be in two geographical locations at the same time. The SMTP server will have new information to apply to threat detection heuristics, ie to treat the use of the same client identifier token from two locations, as a possible brute force or forgery situation.

6.3 Use Cases of CLIENTID

The SMTP server may use the additional information from CLIENTID with its interactions with SMTP clients in the following manner:

1. Restrict use of an authorized identity to a set of client identities, thereby offering an added level of security. For example, the use of an authorized identity may only be permitted from a single device using the client identity as a form of whitelisting.
2. Identify that the same client identity is used to access multiple authorized identities and restrict access to the SMTP service. For example, a client that has successfully gained access to many authorized identities may be identified through its use of a shared client identity.
3. Retain knowledge of client identities previously presented with an authorized identity and if an identity not previously seen is used

restrict access to the SMTP service.

4. Require that the SMTP client present a token such as a license key established outside of the SMTP session in order to make use of any authorized identity;
5. Apply different security policies to clients that provide a client identity versus those which do not. For example, provide clients providing such an identity with additional trust.
6. Ability to rate limit or block based on the presented client identifier token when multiple devices use a shared IP address without affecting other devices.
7. Ability to detect distributed and localized dictionary attacks and brute force attacks.
8. Use the client identifier token as a third factor to be passed to authentication methods. [SASL]

6.4. Other SMTP Client Identifiers

The [SMTP] protocol and its extensions describe methods whereby an SMTP client may provide identity information to an SMTP server. Some of these identities are listed for contrast:

1. The client connection source provides an IP address associated with the SMTP session. This may be accompanied by a PTR record and/or GeoIP information.
2. The EHLO command allows a client to identify itself with a domain or address for an SMTP session.
3. The [AUTH] SMTP extension allows the client to establish an authorized identity for an SMTP session.
4. The MAIL command identifies a specific sender for a mail transaction.

6.5. Future Considerations

In the future there may be a demand for being able to provide multiple CLIENTID commands with different client identity types. For instance, it may be desirable for a device to identify itself, both with a hardware device identifier and a software identifier. We believe this to be out of scope, and can be accommodated with a special client identifier token which encapsulates both.

7. Client Identity Types

This document does not specify any CLIENTID identity type that MUST be supported. The client identity type is meant to be defined by the client implementation that is designed to access the SMTP server and protocol. For instance, many SMTP client software implementations

already create a distinct UUID for each account. Some commercial email clients have a license key. Some physical devices that need to of client identity type that conforms to the definition, it is interact with SMTP might have a unique hardware ID or MAC Address.

While there is no pre-defined list of client identity type defined by this RFC, and all SMTP servers should be prepared to accept any form suggested that SMTP client developers carefully consider the name of the client identity type. For example, rather than using a client identity type of UUID, consider the advantages of making it more distinct, eg "<product_short_code>UUID". This way the SMTP server can better record histories, eg the difference between say a Thunderbird generated unique id, and a Mutt generated unique id.

Some examples of identity type might be UUID, LICENSE, DEVICE_ID, MAC and/or COOKIE. It is expected that the most common types might be related to distinct UUID, LICENSEKEY, or HARDWAREID.

An SMTP server SHOULD NOT reject an unidentified CLIENTID type, except for specific policy use cases.

It is envisioned that in the future it will be useful to propose a set of standardized client-identity-type to help with validation, or to allow the SMTP server to apply ACL rules on expected types, this would be an extension to this RFC.

1. UUID

UUID is a common practice to represent either a individual user, hardware device or software installation associated with a specific individual. The support of UUID enables existing UUID implementations to be used to semi-uniquely identify a device associated with an individual. A definition of the format should be considered. Otherwise non-standard UUID might be a separate type specific to the software implementation, for instance TBIRD-UUID.

2. LICENSE

An SMTP client may find it useful to identify the license key of software it is using. Such licenses are typically crafted such that they are unique and useful to identify a software installation. This is more normally suited for a software designed for a single-user. While LICENSE could be standard type again, it might more more helpful to specify a vendor specific type such as BBLICENSEKEY.

3. DEVICE_ID

Many hardware devices are designed to be used by a single individual and already have an associated hardware device id. While a standard type might be defined, it also might be more helpful to use a vendor specific type, such as ATOM-DEVICEID.

4. MAC

The MAC address traditionally was used as a worldwide identifier both of the unique device, as well as it's vendor and product category, however this is not always the case anymore, in the case of it's usage in 'virtual' devices. But for many hardware devices which are required to access a defined SMTP resource, the MAC address may still be a simple unique identifier. MAC should NOT be used, unless this is a MAC address that can be associated to a vendor using standard MAC registration information as defined or set by the IEEE Standards Association and is meant to represent a unique device.

5. COOKIE

While not guranteed to be consistent many web applications are designed to access SMTP directly and may need to have a semi-unique identifier available as part of the web based transaction. It is assumed that COOKIE encompasses the group of web based tokens known to persist from session to session. A specific web based application can provide sufficient information in the actual client-identifier-token to differentiate between applications and or websites, and are convenient as they can be related to very specific domains, and are universally available to web application designers.

As a reminder, an SMTP server SHOULD NOT retain and/or store the CLIENTID information WITH authentication credentials or authentication systems directly, but the SMTP service MAY associate the CLIENTID with a specific account holder, eg to create a history file of known CLIENTID tokens associated or permitted to access or present authentication credentials for that account holder.

This document recommends that a server associates a set of flags that describes how the CLIENTID command should be handled for any given client identity type.

1. Handled but treat as not presented (ignored, no persistance)
2. Store in SMTP session but treat as not presented (for debug)
3. Store in the SMTP session, so it is available to System log
4. Store in the SMTP session, so it is available to User log
5. Use for authentication
6. Use for alert when authentication fails
7. Use for alert when authentication succeeds
8. Unused

8. Examples

8.1 UUID Address as Client Identity

```
C: [connection established]
S: 220 server.example.com ESMTP ready
C: EHLO client.example.net
S: 250-server.example.com
```

```
S: 250-STARTTLS
S: 250 AUTH LOGIN
C: STARTTLS
S: 220 Go ahead
C: <starts TLS negotiation>
C & S: <negotiate a TLS session>
C & S: <check result of negotiation>
C: EHLO client.example.net
S: 250-server.example.com
S: 250-AUTH LOGIN
S: 250 CLIENTID
C: CLIENTID UUID 23bf83be-aad7-46aa-9e0f-39191ccf402f
S: 250 OK
C: AUTH LOGIN dGVzdAB0ZXN0ADEyMzQ=
S: 235 Authentication successful
C: MAIL FROM:<sender@example.net>
S: 250 OK
C: RCPT TO:<receiver@example.com>
S: 250 OK
C: DATA
S: 354 Ready for message content
C: <body>
C: .
S: 250 OK
C: QUIT
S: 221 server.example.com Service closing transmission channel
```

8.2 Client Identity Without a TLS/SSL Session

```
C: [connection established over a plaintext connection]
S: 220 server.example.com ESMTP ready
C: EHLO client.example.net
S: 250-server.example.com
S: 250 STARTTLS
C: CLIENTID MAC 08:9e:01:70:f6:46
S: 500 Syntax error, command unrecognised
C: MAIL FROM:<sender@example.net>
S: 250 OK
C: QUIT
S: 221 server.example.com Service closing transmission channel
```

The server rejects use of the CLIENTID command as no TLS/SSL session was yet established.

8.3 Client Identity Leading to Rejection

```
C: [connection established over a plaintext connection]
S: 220 server.example.com ESMTP ready
C: EHLO client.example.net
S: 250-server.example.com
S: 250 STARTTLS
C: STARTTLS
S: 220 Go ahead
C: <starts TLS negotiation>
```

```
C & S: <negotiate a TLS session>
C & S: <check result of negotiation>
C: EHLO client.example.net
S: 250-server.example.com
S: 250 CLIENTID
C: CLIENTID MAC 08:9e:01:70:f6:46
S: 250 OK
C: AUTH LOGIN dGVzdAB0ZXN0ADEyMzQ=
S: 235 Authentication successful
S: 550 Server policy does not permit your use of this mail system
C: QUIT
S: 221 server.example.com Service closing transmission channel
```

The server rejects use of the mail system after deciding that the provided client identity does not establish sufficient privileges.

8.4 Malformed CLIENTID Command

```
C: [connection established over a plaintext connection]
S: 220 server.example.com ESMTP ready
C: EHLO client.example.net
S: 250-server.example.com
S: 250 STARTTLS
C: STARTTLS
S: 220 Go ahead
C: <starts TLS negotiation>
C & S: <negotiate a TLS session>
C & S: <check result of negotiation>
C: EHLO client.example.net
S: 250-server.example.com
S: 250 CLIENTID
C: CLIENTID MAC
S: 501 Syntax error in parameters or arguments
C: QUIT
S: 221 server.example.com Service closing transmission channel
```

The server rejects the CLIENTID command as it is not well formed due to there being only a single parameter provided.

9. Security Considerations

As this extension provides an additional means of communicating information from a client to a server it is clear there is additional information divulged to the server. This may have privacy considerations depending on the client identity type or its contents. For example, it may reveal a MAC address of the device used to communicate with a server that would not previously have been revealed. While it has been useful to use identifier such as email address for authentication it is easy for these authentication tokens to be shared and/or reused and/or be publically available for other purposes. An SMTP server and or its operators SHOULD not share any CLIENTID information presented with a third party as it may represent or be linked to an individual and SHOULD never be shared in association with authentication tokens.

As well, while this service extension requires that the identity information only be transmitted over an encrypted channel to reduce the risk of eavesdropping, it does not specify any policies or practices required in the establishment of such a channel, and so it is the responsibility of the client and the server to determine that the communication medium meets their requirements.

10. IANA Considerations

10.1 SMTP Extension Registration

Section 2.2.2 of [SMTP] sets out the procedure for registering a new SMTP extension.

This extension will need to be registered.

11. References

11.1. Normative References

- [ABNF] Crocker, D., Ed., and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<http://www.rfc-editor.org/info/rfc5234>>.
- [AUTH] Siemborski, R., Ed., and A. Melnikov, Ed., "SMTP Service Extension for Authentication", RFC 4954, DOI 10.17487/RFC4954, July 2007, <<http://www.rfc-editor.org/info/rfc4954>>.
- [KEYWORDS] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [SMTP] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <<http://www.rfc-editor.org/info/rfc5321>>.
- [STARTTLS] Hoffman, P., "SMTP Service Extension for Secure SMTP over Transport Layer Security", RFC 3207, DOI 10.17487/RFC3207, February 2002, <<http://www.rfc-editor.org/info/rfc3207>>.
- [SUBMIT] Gellens, R. and J. Klensin, "Message Submission for Mail", STD 72, RFC 6409, DOI 10.17487/RFC6409, November 2011, <<http://www.rfc-editor.org/info/rfc6409>>.

Appendix A. CLIENTID Product Support

Since publishing the SMTP Client Identity RFC draft, multiple email server and client vendors have implemented CLIENTID support into their products, e.g. MailEnable, MagicMail, SaneBox, BlueMail, emClient, and Thunderbird.

Contributors

Michael Peddemors
LinuxMagic

Authors' Addresses

William Storey
LinuxMagic
#405 - 860 Homer St.
Vancouver, British Columbia
CA V6B 2W5

EMail: william@linuxmagic.com

Deion Yu
LinuxMagic
#405 - 860 Homer St.
Vancouver, British Columbia
CA V6B 2W5

EMail: deiony@linuxmagic.com

INTERNET-DRAFT
<draft-yu-imap-client-id-03.txt>
Intended Status: Standards Track
Expires May 19, 2020

Y. Deion
LinuxMagic

November 19, 2019

IMAP Service Extension for Client Identity
<draft-yu-imap-client-id-03.txt>

Abstract

This document defines an Internet Message Access Protocol (IMAP) service extension called "CLIENTID" which provides a method for clients to indicate an identity to the server.

This identity is an additional token that may be used for security and/or informational purposes, and with it a server may optionally apply heuristics using this token.

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), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/lid-abstracts.html>

The list of Internet-Draft Shadow Directories can be accessed at
<http://www.ietf.org/shadow.html>

Copyright Notice

Copyright (c) 2018 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 (<http://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	2
2. Conventions Used in This Document	3
3. CLIENTID	3
3.1. CLIENTID Command	3
3.2. CLIENTID Arguments	3
3.3. Advertising the CLIENTID capability	4
3.4. Restrictions on the CLIENTID command	4
4. Formal Syntax	4
5. Discussion	5
5.1. Applying heuristics to CLIENTID	5
5.2. Utility of CLIENTID	5
5.3. Use Cases of CLIENTID	6
5.4. Other IMAP Client Identifiers	6
5.5. Future Considerations	7
6. Client Identity Types	7
7. Examples	8
7.1. UUID as Client Identity	8
7.2. Malformed CLIENTID Command	8
7.3. Client Identity Without a TLS/SSL Session	9
7.4. Client Identity Leading to Rejection	9
8. Security Considerations	9
9. IANA Considerations	10
10. References	10
10.1. Normative References	10
Contributors	10
Authors' Addresses	10

1. Introduction

The [IMAP] protocol and its extensions describe methods whereby an client may provide identity and/or authentication information to an IMAP server. However, these existing methods are subject to limitations and none offer a way to identify the IMAP client with absolute confidence. This document defines an IMAP service extension to provide an additional identity token which can represent the IMAP client with a higher degree of certainty when accessing the IMAP server.

Typically IMAP clients enter the authenticated state by using either the AUTHENTICATE or LOGIN command. IMAP servers are often subject to malicious clients attempting to use authorization credentials and/or identities not intended for their use (e.g. stolen credentials or brute force attacks). When such an attack is attempted, the IMAP server may be unable to identify the impersonation and restrict such an unintended use by someone other than the authorized user or said credentials. While there are ways to identify the source of the IMAP client such as its IP address, it would be useful if there was an additional way to uniquely identify the client in a method solely available across an encrypted channel.

Using the CLIENTID extension, an IMAP client can provide an additional identity token to the server called its "client identity".

The client identity can provide unique characteristics about the client accessing the IMAP service and may be combined with existing identification mechanisms in order to identify the client. An IMAP server may then apply additional security policies using this identity such as restricting use of the service to clients presenting recognized client identities or only allowing use of authorized identities that match previously established client identities.

The CLIENTID extension is present in any IMAP implementation that returns "CLIENTID" as one of the supported capabilities to the CAPABILITY command.

2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [KEYWORDS].

Formal syntax is specified using [ABNF].

Example lines prefaced by "C:" are sent by the client and ones prefaced by "S:" by the server.

"Connection" refers to the entire sequence of client/server interaction from the initial establishment of the network connection until its termination.

3. CLIENTID

3.1. CLIENTID Command

Arguments: client identity type
client identity token

Responses: no specific responses for this command

Result: OK - clientid completed, client identity stored
BAD - command unknown or arguments invalid

Note that a valid CLIENTID command will never return the NO result because heuristics MUST NOT be applied to the CLIENTID arguments at this stage. Instead the client identity information SHOULD be stored and passed along to any and all [SASL] authentication mechanisms.

3.2. CLIENTID Arguments

The CLIENTID command takes the following two arguments:

1. client identity type: A string identifying the identity type the client is providing. It MUST be between 1 and 16 characters and comprised of only alphanumeric and dash characters.
2. client identity token: A string identifying the client. It MUST be between 1 and 128 printable characters.

The IMAP server MUST reject any CLIENTID command with badly formatted arguments. The IMAP server MUST accept the arguments from a valid CLIENTID command and SHOULD store it at the minimum for the remaining duration of the IMAP connection.

3.3. Advertising the CLIENTID capability

The CLIENTID capability is used to tell the IMAP client that the IMAP server supports the CLIENTID extension. However, certain conditions MUST be met before the IMAP server advertises the CLIENTID capability.

1. The IMAP server and IMAP client MUST negotiate encryption via STARTTLS/SSL or some other secure mechanism.
2. The IMAP server MUST be in the non-authenticated state.
3. The IMAP server MUST have the CLIENTID extension support enabled.

While all the conditions are met, the IMAP server MUST advertise the CLIENTID capability in all proceeding CAPABILITY commands.

3.4. Restrictions on the CLIENTID command

Under certain circumstances, the use of the CLIENTID command will be restricted:

1. Before the CLIENTID capability has been advertised, the IMAP server MUST reject any issued CLIENTID command and the IMAP client MUST NOT issue the CLIENTID command.
2. Outside of the non-authenticated state, the IMAP server MUST reject any CLIENTID command issued by the IMAP client and the IMAP client MUST NOT issue the CLIENTID command.
3. Once a valid CLIENTID command has been issued, the IMAP server MUST reject any further CLIENTID command issued by the IMAP client and the IMAP client MUST NOT issue any subsequent CLIENTID commands.

4. Formal Syntax

The following syntax specification uses the Augmented Backus-Naur Form notation as specified in [ABNF]. [IMAP] defines the non-terminals "capability" and "command-nonauth".

Except as noted otherwise, all alphabetic characters are case-insensitive. The use of upper or lower case characters to define token strings is for editorial clarity only. Implementations MUST accept these strings in a case-insensitive fashion.

capability =/ "CLIENTID"

command-nonauth =/ client-id

```
client-id      = "CLIENTID" SP client-id-type SP client-id-token

client-id-type = 1*16 ALPHA / DIGIT / "-"
                ;; alphanumeric with dash character

client-id-token = 1*128 VCHAR
                ;; any printable US-ASCII character
```

5. Discussion

5.1. Applying heuristics to CLIENTID

This section discusses the possible heuristics that can be applied to the information that is presented via the CLIENTID command. This information includes whether a valid CLIENTID command was issued, the client identity type and the client identity token.

1. The IMAP server MAY choose to require that a successful CLIENTID command be issued or that a particular client identity type be presented before processing or accepting an authentication request.
2. The IMAP server MAY reject any authentication request not preceded with a client identity type that matches ACL's or rules as defined in the IMAP server.
3. An IMAP server MAY reject any authentication request preceded by a CLIENTID command that contains a client identity type or client identity token that the server chooses not to accept for any reason such as by policy.
4. An IMAP server MAY reject any authentication request preceded by a CLIENTID command that contains a client identity type or client identity token that the server has chosen to disable or revoke use of either temporarily or permanently.

The IMAP server SHOULD only ever reject an IMAP client based on CLIENTID information during or after the authentication process/handler. In the interest of limiting the amount of information being revealed, the rejection message SHOULD be as generic as possible and SHOULD NOT reveal any information on the heuristics.

Even if the client identity type and/or client identity token are not recognized, supported or permitted by the server and/or the owner of the authentication credentials, the presented information may still be useful for analysis.

5.2. Utility of CLIENTID

Regardless of how frowned upon, users commonly reuse authorization information (like the username and password pair) across multiple services. When one service is compromised, malicious actors can also gain access to other services where the user also used the same credentials. Based on this representative problem alone, the utility

of CLIENTID as an additional layer of determining the rights to present such authorization information becomes quickly apparent.

The utility of CLIENTID may be seen by considering the following:

1. An IMAP server could recognize a device not historically known to have presented the authentication credentials before.
2. An IMAP server could restrict authentication from actors not presenting a valid CLIENTID, or an account holder that the IMAP server provides service for could restrict authentication to only those devices that present valid CLIENTID.
3. An IMAP server could restrict authentication to only devices which present a CLIENTID containing a client type identifier which the account holder or operator of the server deems to be permitted. (Eg. Only allow vendor A's devices)
4. An IMAP server could alert an account holder that an attempt to present their authorization credentials came from an unknown, unrecognized, or different device.

However, this extends beyond just the restriction of authentication. While it might be argued that this can be served as a special form of SASL, by implementing this in the IMAP service itself, the IMAP service can choose before allowing a connection to be passed to a SASL implementation, allowing it to perform other heuristics, such as brute force attacks, more effeciently.

Recent evolution of the internet as a whole, has brought about large scale data breaches, compromised botnets comprising of millions of nodes, and transitions to Carrier Grade NAT, and the flourishing of IoT devices, means traditional methods of protecting against brute force attacks have become much more difficult. Traditional methods such as rate limiting and/or blocking access by IP are no longer viable without introducing collateral effects, such as either blocking legitimate users, or creating the conditions that allow DOS (Denial of Service) to legitimate users.

Historically, IMAP and other services used what is technically a two factor, the email/user and password, albeit not effective in that the email is a KNOWN value. And with the propensity for users to use a simple password, and the hundreds of millions of email addresses exposed in data breaches, or available by other means the ability to brute force is quite simple. While of course it is recommended that users use longer and more secure passwords, this is not the de facto situation, and the threats when credentials get compromised are significant. And with 'botnet' operators able to engage millions of IoT in a distributed brute force, the status quo is dangerous. Adding another non-public factor to be used as part of access control adds a strength against brute force by many factors. This accomplishes that in a backwards compatible fashion, encouraging adoption.

But for the IMAP server, it also offers additional abilities.

Historically, under brute force attacks, rate limiting or blocking by IP Address was possible with little damage. But with the proliferation of IoT devices, smart phones, and the run-out of IP space, we have conditions where thousands of devices could be behind an IP Address, or IP(s) that are dynamic with devices changing IP(s) in minutes, blocking or rate limiting an IP bears risks of blocking legitimate users. By implementing a level of uniqueness to a connecting device, introduces the ability to restrict or block a subset from connecting to the service for either brute force or dictionary attacks, while still allowing other devices to continue to be able to present authentication successfully.

While 'forgery' and/or the use of random client identifier is possible, such behavior is also more readily detectable when a device identifier is presented.

1. The IMAP server, when faced with hundreds of devices behind the same IP address, during an attack can restrict authentication attempts to only connections presenting a valid client identifier token.
2. The IMAP server, during an attack, can restrict authentication to only historically known devices.
3. The IMAP server can differentiate between many different devices behind the same IP, and apply maximum connections per device, rather than maximum connections per IP.
4. While a person may present authentication credentials from many different geographical locations, eg, home, office, and travel, a single device will not in general be able to be in two geographical locations at the same time. The IMAP server will have new information to apply to threat detection heuristics, ie to treat the use of the same client identifier token from two locations, as a possible brute force or forgery situation.

5.3. Use Cases of CLIENTID

With CLIENTID the IMAP server has additional information it may use in its interactions with the client. It may:

1. Restrict use of an authorization tokens to a set of client identity token identities, thereby offering an added level of security. For example the use of authorization credentials may only be accompanied by a specified set of CLIENTID tokens and/or types for a specific account holder, or set of account holders
2. Identify that the same CLIENTID token is used to access multiple authorized identities, and restrict access to the IMAP service. For example a malicious client that has attempted to gain access using multiple authorization tokens may be identified through its unusual behavior.
3. Retain knowledge of CLIENTID tokens previously presented with

specific authorization credentials, and if the token has not been previously seen, restrict access to the IMAP service.

4. Require that the IMAP client present a token such as a license key established outside of the IMAP session in order to make use of any authorized identity.
5. Apply different security policies to clients that provide a CLIENTID token versus those which do not. For example, provide clients providing such an identity with additional trust.
6. Ability to rate limit or block based on the presented client-identifier-token, when multiple devices use a shared IP address, without affecting other devices.
7. Ability to detect distributed and localized dictionary attacks and brute force attacks.
8. Use the client-identifier-token as a third factor to be passed to authentication methods. [SASL]

5.4. Other IMAP Client Identifiers

The [IMAP] protocol and its extensions describe methods whereby an

IMAP client may provide identity information to an IMAP server. Some of these identifiers are listed for contrast:

1. The client connection provides a source IP address associated with the IMAP session. This may be accompanied by a PTR record and/or GeoIP information.
2. The AUTHENTICATE and LOGIN command allows the client to present a user and/or password/authentication mechanism for an IMAP session.

5.5. Future Considerations

In the future there may be a demand for being able to provide multiple CLIENTID commands with different client identity types. For instance, it may be desirable for a device to identify itself, both with a hardware device identifier, and a software identifier. We believe this to be out of scope, and can be accommodated with a special client-identifier-token which encapsulates both.

6. Client Identity Types

This document does not specify any CLIENTID identity type that MUST be supported. The client identity type is meant to be defined by the client implementation that is designed to access the IMAP server and protocol. For instance, many IMAP client software implementations already create a distinct UUID for each account. Some commercial email clients have a license key. Some physical devices that need to of client identity type that conforms to the definition, it is

interact with IMAP might have a unique hardware ID or MAC Address. While there is no pre-defined list of client identity type defined by this RFC, and all IMAP servers should be prepared to accept any form suggested that IMAP client developers carefully consider the name of the client identity type. For example, rather than using a client identity type of UUID, consider the advantages of making it more distinct, eg "<product_short_code>UUID". This way the IMAP server can better record histories, eg the difference between say a Thunderbird generated unique id, and a Mutt generated unique id.

Some examples of identity type might be UUID, LICENSE, DEVICE_ID, MAC and/or COOKIE. It is expected that the most common types might be related to distinct UUID, LICENSEKEY, or HARDWAREID.

An IMAP server SHOULD NOT reject an unidentified CLIENTID type, except for specific policy use cases.

It is envisioned that in the future it will be useful to propose a set of standardized client-identity-type to help with validation, or to allow the IMAP server to apply ACL rules on expected types, this would be an extension to this RFC.

1. UUID

UUID is a common practice to represent either a individual user, hardware device or software installation associated with a specific individual. The support of UUID enables existing UUID implementations to be used to semi-uniquely identify a device associated with an individual. A definition of the format should be considered. Otherwise non-standard UUID might be a separate type specific to the software implementation, for instance TBIRD-UUID.

2. LICENSE

An IMAP client may find it useful to identify the license key of software it is using. Such licenses are typically crafted such that they are unique and useful to identify a software installation. This is more normally suited for a software designed for a single-user. While LICENSE could be standard type again, it might be more helpful to specify a vendor specific type such as BBLICENSEKEY.

3. DEVICE_ID

Many hardware devices are designed to be used by a single individual and already have an associated hardware device id. While a standard type might be defined, it also might be more helpful to use a vendor specific type, such as ATOM-DEVICEID.

4. MAC

The MAC address traditionally was used as a worldwide identifier both of the unique device, as well as it's vendor and product

category, however this is not always the case anymore, in the case of it's usage in 'virtual' devices. But for many hardware devices which are required to access a defined IMAP resource, the MAC address may still be a simple unique identifier. MAC should NOT be used, unless this is a MAC address that can be associated to a vendor using standard MAC registration information as defined or set by the IEEE Standards Association and is meant to represent a unique device.

5. COOKIE

While not guranteed to be consistent many web applications are designed to access IMAP directly and may need to have a semi-unique identifier available as part of the web based transaction. It is assumed that COOKIE encompasses the group of web based tokens known to persist from session to session. A specific web based application can provide sufficient information in the actual client-identifier-token to differentiate between applications and or websites, and are convenient as they can be related to very specific domains, and are universally available to web application designers.

As a reminder, an IMAP server SHOULD NOT retain and/or store the CLIENTID information WITH authentication credentials or authentication systems directly, but the IMAP service MAY associate the CLIENTID with a specific account holder, eg to create a history file of known CLIENTID tokens associated or permitted to access or present authentication credentials for that account holder.

This document recommends that an IMAP server handle any given client identity type from a CLIENTID command in one or more of the following manners.

1. Handled but treat as not presented (ignored, no persistance)
2. Store in IMAP session but treat as not presented (debugging)
3. Store in the IMAP session, so it is available to System log
4. Store in the IMAP session, so it is available to User log
5. Use for authentication
6. Use for alert when authentication fails
7. Use for alert when authentication succeeds
8. Unused

7. Examples

7.1. UUID as Client Identity

```
C: [connection established over a plaintext connection]
C: a001 CAPABILITY
S: * CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI LOGINDISABLED
S: a001 OK CAPABILITY completed
C: a002 STARTTLS
S: a002 OK STARTTLS completed
<TLS negotiation, further commands are under [TLS] layer>
C: a003 CAPABILITY
```

```
S: * CAPABILITY IMAP4rev1 AUTH=GSSAPI AUTH=PLAIN CLIENTID
S: a003 OK CAPABILITY completed
C: a004 CLIENTID UUID 23bf83be-aad7-46aa-9e0f-39191ccf402f
S: a004 OK CLIENTID completed
C: a005 LOGIN joe password
S: a005 OK LOGIN completed
```

7.2. Malformed CLIENTID Command

```
C: [connection established over a plaintext connection]
C: a001 CAPABILITY
S: * CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI LOGINDISABLED
S: a001 OK CAPABILITY completed
C: a002 STARTTLS
S: a002 OK STARTTLS completed
<TLS negotiation, further commands are under [TLS] layer>
C: a003 CAPABILITY
S: * CAPABILITY IMAP4rev1 AUTH=GSSAPI AUTH=PLAIN CLIENTID
S: a003 OK CAPABILITY completed
C: a004 CLIENTID UUID
S: a004 BAD Error in IMAP command received by server
```

The IMAP server rejects the CLIENTID command as it is not well formed due to there being only a single parameter provided.

7.3. Client Identity without TLS/SSL Session

```
C: [connection established over a plaintext connection]
C: a001 CAPABILITY
S: * CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI LOGINDISABLED
S: a001 OK CAPABILITY completed
C: a002 CLIENTID UUID 23bf83be-aad7-46aa-9e0f-39191ccf402f
S: a002 BAD Unknown IMAP command received by server
```

The IMAP server rejects use of the CLIENTID command as the CLIENTID capability had not been advertised because no encryption was negotiated between the IMAP server and IMAP client.

7.4. Client Identity Leading to Rejection

```
C: [connection established over a plaintext connection]
C: a001 CAPABILITY
S: * CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI LOGINDISABLED
S: a001 OK CAPABILITY completed
C: a002 STARTTLS
S: a002 OK STARTTLS completed
<TLS negotiation, further commands are under [TLS] layer>
C: a003 CAPABILITY
S: * CAPABILITY IMAP4rev1 AUTH=GSSAPI AUTH=PLAIN CLIENTID
S: a003 OK CAPABILITY completed
C: a004 CLIENTID UUID 23bf83be-aad7-46aa-9e0f-39191ccf402f
S: a004 OK CLIENTID completed
C: a005 LOGIN joe password
S: a005 BAD Failed to authenticate
```

The IMAP server rejects use of the system during the LOGIN command after deciding that the provided client identity does not establish sufficient privileges. Note that the error message that's returned to the client is very generic and does not reveal any information about CLIENTID and/or the existence of 'joe' and/or the validity of the password.

8. Security Considerations

As this extension provides an additional means of communicating information from a client to a server it is clear there is additional information divulged to the server. This may have privacy considerations depending on the client identity type or its contents. For example, it may reveal a MAC address of the device used to communicate with a server that would not previously have been revealed. While it has been useful to use identifier such as email address for authentication it is easy for these authentication tokens to be shared and/or reused and/or be publically available for other purposes. An IMAP server and or its operators SHOULD not share any CLIENTID information presented with a third party as it may represent or be linked to an individual and SHOULD never be shared in association with authentication tokens.

As well, while this service extension requires that the identity information only be transmitted over an encrypted channel to reduce the risk of eavesdropping, it does not specify any policies or practices required in the establishment of such a channel, and so it is the responsibility of the client and the server to determine that the communication medium meets their requirements.

9. IANA Considerations

The IANA is requested to add CLIENTID to the "IMAP 4 Capabilities" registry, <http://www.iana.org/assignments/imap4-capabilities>.

10. References

10.1. Normative References

- [ABNF] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.
- [IMAP] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, March 2003.
- [KEYWORDS] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [SASL] Myers, J., "Simple Authentication and Security Layer (SASL)", RFC 2222, October 1997.

[TLS] Dierks, T. and C. Allen, "The TLS Protocol
Version 1.0", RFC 2246, January 1999.

Contributors

Michael Peddemors
LinuxMagic

Authors' Addresses

Deion Yu
LinuxMagic
#405 - 860 Homer St.
Vancouver, British Columbia
CA V6B 2W5

EMail: deiony@linuxmagic.com

INTERNET-DRAFT
<draft-yu-imap-client-id-07.txt>
Intended Status: Standards Track
Expires May 18, 2022

Y. Deion
LinuxMagic

November 18, 2021

IMAP Service Extension for Client Identity
<draft-yu-imap-client-id-07.txt>

Abstract

This document defines an Internet Message Access Protocol (IMAP) service extension called "CLIENTID" which provides a method for clients to indicate an identity to the server.

This identity is an additional token that may be used for security and/or informational purposes, and with it a server may optionally apply heuristics using this token.

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), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/lid-abstracts.html>

The list of Internet-Draft Shadow Directories can be accessed at
<http://www.ietf.org/shadow.html>

Copyright Notice

Copyright (c) 2018 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 (<http://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	2
2. Conventions Used in This Document	3
3. CLIENTID	3
3.1. CLIENTID Command	3
3.2. CLIENTID Arguments	3
3.3. Advertising the CLIENTID capability	4
3.4. Restrictions on the CLIENTID command	4
4. Formal Syntax	4
5. Discussion	5
5.1. Background	5
5.2. Applying heuristics to CLIENTID	6
5.3. Utility of CLIENTID	6
5.4. Use Cases of CLIENTID	7
5.5. Other IMAP Client Identifiers	8
5.6. Future Considerations	8
6. Client Identity Types	8
7. Examples	10
7.1. UUID as Client Identity	10
7.2. Malformed CLIENTID Command	11
7.3. Client Identity Without a TLS/SSL Session	11
7.4. Client Identity Leading to Rejection	11
8. Security Considerations	12
9. IANA Considerations	12
10. References	12
10.1. Normative References	12
Appendix A. CLIENTID Product Support	13
Contributors	13
Authors' Addresses	13

1. Introduction

The [IMAP] protocol and its extensions describe methods whereby an client may provide identity and/or authentication information to an IMAP server. However, these existing methods are subject to limitations and none offer a way to identify the IMAP client with absolute confidence. This document defines an IMAP service extension to provide an additional identity token which can represent the IMAP client with a higher degree of certainty when accessing the IMAP server.

Typically IMAP clients enter the authenticated state by using either the AUTHENTICATE or LOGIN command. IMAP servers are often subject to malicious clients attempting to use authorization credentials and/or identities not intended for their use (e.g. stolen credentials or brute force attacks). When such an attack is attempted, the IMAP server may be unable to identify the impersonation and restrict such an unintended use by someone other than the authorized user or said credentials. While there are ways to identify the source of the IMAP client such as its IP address, it would be useful if there was an additional way to uniquely identify the client in a method solely available across an encrypted channel.

Using the CLIENTID extension, an IMAP client can provide an additional identity token to the server called its "client identity". The client identity can provide unique characteristics about the client accessing the IMAP service and may be combined with existing identification mechanisms in order to identify the client. An IMAP server may then apply additional security policies using this identity such as restricting use of the service to clients presenting recognized client identities or only allowing use of authorized identities that match previously established client identities.

The CLIENTID extension is present in any IMAP implementation that returns "CLIENTID" as one of the supported capabilities to the CAPABILITY command.

2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [KEYWORDS].

Formal syntax is specified using [ABNF].

Example lines prefaced by "C:" are sent by the client and ones prefaced by "S:" by the server.

"Connection" refers to the entire sequence of client/server interaction from the initial establishment of the network connection until its termination.

3. CLIENTID

3.1. CLIENTID Command

Arguments: client identity type
 client identity token

Responses: no specific responses for this command

Result: OK - clientid completed, client identity stored
 BAD - command unknown or arguments invalid

Note that a valid CLIENTID command will never return the NO result because heuristics MUST NOT be applied to the CLIENTID arguments at this stage. Instead the client identity information SHOULD be stored and passed along to any and all [SASL] authentication mechanisms.

3.2. CLIENTID Arguments

The CLIENTID command takes the following two arguments:

1. client identity type: A string identifying the identity type the client is providing. It MUST be between 1 and 16 alphanumeric and dash characters.

2. client identity token: A string identifying the client. It MUST be between 1 and 128 printable characters.

The IMAP server MUST reject any CLIENTID command with badly formatted arguments. The IMAP server MUST accept the arguments from a valid CLIENTID command and SHOULD store it at the minimum for the remaining duration of the IMAP connection.

3.3. Advertising the CLIENTID capability

The CLIENTID capability is used to tell the IMAP client that the IMAP server supports the CLIENTID extension. However, certain conditions MUST be met before the IMAP server advertises the CLIENTID capability.

1. The IMAP server and IMAP client MUST negotiate encryption via STARTTLS/SSL or some other secure mechanism.
2. The IMAP server MUST be in the non-authenticated state.
3. The IMAP server MUST have the CLIENTID extension support enabled.

While all the conditions are met, the IMAP server MUST advertise the CLIENTID capability in all proceeding CAPABILITY commands.

3.4. Restrictions on the CLIENTID command

Under certain circumstances, the use of the CLIENTID command will be restricted:

1. Before the CLIENTID capability has been advertised, the IMAP server MUST reject any issued CLIENTID command and the IMAP client MUST NOT issue the CLIENTID command.
2. Outside of the non-authenticated state, the IMAP server MUST reject any CLIENTID command issued by the IMAP client and the IMAP client MUST NOT issue the CLIENTID command.
3. Once a valid CLIENTID command has been issued, the IMAP server MUST reject any further CLIENTID command issued by the IMAP client and the IMAP client MUST NOT issue any subsequent CLIENTID commands.

4. Formal Syntax

The following syntax specification uses the Augmented Backus-Naur Form notation as specified in [ABNF]. [IMAP] defines the non-terminals "capability" and "command-nonauth".

Except as noted otherwise, all alphabetic characters are case-insensitive. The use of upper or lower case characters to define token strings is for editorial clarity only. Implementations MUST accept these strings in a case-insensitive fashion.

```
capability      =/ "CLIENTID"

command-nonauth =/ client-id

client-id       = "CLIENTID" SP client-id-type SP client-id-token

client-id-type  = 1*16 ALPHA / DIGIT / "-"
                  ;; alphanumeric with dash character

client-id-token = 1*128 VCHAR
                  ;; any printable US-ASCII character
```

5. Discussion

5.1 Background

The historical standard of using the user and password combination as a means of authentication is no longer effective in this day and age with recent developments in the world.

1. ISPs transistioning to Carrier-grade NAT due to IPv4 address exhaustion placing multiple devices behind the same IP address.
2. Numerous large scale data breaches exposing millions of user and password combinations.
3. Continued propensity for user to use same simple passwords across multiple accounts.
4. Botnets growing larger and more sophisticated due to the profileration of IoT devices.

As a result, brute force attacks against web services have become increasingly effective as malicious actors have easy access to millions of email addresses, commonly used passwords and massive botnets while the safety practices of users have not improved.

The traditional methods of defending against these types of attacks like rate limiting and blocking by IP addresses are no longer viable without collateral damange as thousands of devices could potentially be behind the same IP address as more ISPs adopt the CGN/LSN/NAT444 standard, i.e. blocking an IP address due to the actions of a single malicious actor bears the risk of blocking legitimate users.

By introducing CLIENTID as another non-public factor to be used in tandem with the user and password combination, authentication becomes much more resilient against brute force attacks. The email addresses and passwords exposed from the data breaches will no longer be sufficient to authenticate. Rate limiting and blocking can be performed based on the CLIENTID such that only a subset of devices behind the same IP address gets blocked. CLIENTID would also be backwards compatible with existing authentication protocols encouraging adoption.

5.2. Applying heuristics to CLIENTID

This section discusses the possible heuristics that can be applied to the information that is presented via the CLIENTID command. This information includes whether a valid CLIENTID command was issued, the client identity type and the client identity token.

1. The IMAP server MAY choose to require that a successful CLIENTID command be issued or that a particular client identity type be presented before processing or accepting an authentication request.
2. The IMAP server MAY reject any authentication request not preceded with a client identity type that matches ACL's or rules as defined in the IMAP server.
3. An IMAP server MAY reject any authentication request preceded by a CLIENTID command that contains a client identity type or client identity token that the server chooses not to accept for any reason such as by policy.
4. An IMAP server MAY reject any authentication request preceded by a CLIENTID command that contains a client identity type or client identity token that the server has chosen to disable or revoke use of either temporarily or permanently.

The IMAP server SHOULD only ever reject an IMAP client based on CLIENTID information during or after the authentication process/handler. In the interest of limiting the amount of information being revealed, the rejection message SHOULD be as generic as possible and SHOULD NOT reveal any information on the heuristics.

Even if the client identity type and/or client identity token are not recognized, supported or permitted by the server and/or the owner of the authentication credentials, the presented information may still be useful for analysis.

5.3. Utility of CLIENTID

Regardless of how frowned upon, users commonly reuse authorization information (like the username and password pair) across multiple services. When one service is compromised, malicious actors can also gain access to other services where the user also used the same credentials. Based on this representative problem alone, the utility of CLIENTID as an additional layer of determining the rights to present such authorization information becomes quickly apparent.

The utility of CLIENTID may be seen by considering the following:

1. An IMAP server could recognize a device not historically known to have presented the authentication credentials before.
2. An IMAP server could restrict authentication from actors not presenting a valid CLIENTID, or an account holder that the IMAP

server provides service for could restrict authentication to only those devices that present valid CLIENTID.

3. An IMAP server could restrict authentication to only devices which present a CLIENTID containing a client type identifier which the account holder or operator of the server deems to be permitted. (Eg. Only allow vendor A's devices)
4. An IMAP server could alert an account holder that an attempt to present their authorization credentials came from an unknown, unrecognized, or different device.

However, this extends beyond just the restriction of authentication. While it might be argued that this can be served as a special form of SASL, by implementing this in the IMAP service itself, the IMAP service can choose before allowing a connection to be passed to a SASL implementation, allowing it to perform other heuristics, such as brute force attacks, more effeciently.

While 'forgery' and/or the use of random client identifier is possible, such behavior is also more readily detectable when a device identifier is presented.

1. The IMAP server, when faced with hundreds of devices behind the same IP address, during an attack can restrict authentication attempts to only connections presenting a valid client identifier token.
2. The IMAP server, during an attack, can restrict authentication to only historically known devices.
3. The IMAP server can differentiate between many different devices behind the same IP, and apply maximum connections per device, rather than maximum connections per IP.
4. While a person may present authentication credentials from many different geographical locations, eg, home, office, and travel, a single device will not in general be able to be in two geographical locations at the same time. The IMAP server will have new information to apply to threat detection heuristics, ie to treat the use of the same client indentifier token from two locations, as a possible brute force or forgery situation.

5.4. Use Cases of CLIENTID

With CLIENTID the IMAP server has additional information it may use in its interactions with the client. It may:

1. Restrict use of an authorization tokens to a set of client identity token identities, thereby offering an added level of security. For example the use of authorization credentials may only be accompanied by a specified set of CLIENTID tokens and/or types for a specific account holder, or set of account holders

2. Identify that the same CLIENTID token is used to access multiple authorized identities, and restrict access to the IMAP service. For example a malicious client that has attempted to gain access using multiple authorization tokens may be identified through its unusual behavior.
3. Retain knowledge of CLIENTID tokens previously presented with specific authorization credentials, and if the token has not been previously seen, restrict access to the IMAP service.
4. Require that the IMAP client present a token such as a license key established outside of the IMAP session in order to make use of any authorized identity.
5. Apply different security policies to clients that provide a CLIENTID token versus those which do not. For example, provide clients providing such an identity with additional trust.
6. Ability to rate limit or block based on the presented client-identifier-token, when multiple devices use a shared IP address, without affecting other devices.
7. Ability to detect distributed and localized dictionary attacks and brute force attacks.
8. Use the client-identifier-token as a third factor to be passed to authentication methods. [SASL]

5.5. Other IMAP Client Identifiers

The [IMAP] protocol and its extensions describe methods whereby an IMAP client may provide identity information to an IMAP server. Some of these identifiers are listed for contrast:

1. The client connection provides a source IP address associated with the IMAP session. This may be accompanied by a PTR record and/or GeoIP information.
2. The AUTHENTICATE and LOGIN command allows the client to present a user and/or password/authentication mechanism for an IMAP session.

5.6. Future Considerations

In the future there may be a demand for being able to provide multiple CLIENTID commands with different client identity types. For instance, it may be desirable for a device to identify itself, both with a hardware device identifier, and a software identifier. We believe this to be out of scope, and can be accommodated with a special client-identifier-token which encapsulates both.

6. Client Identity Types

This document does not specify any CLIENTID identity type that MUST

be supported. The client identity type is meant to be defined by the client implementation that is designed to access the IMAP server and protocol. For instance, many IMAP client software implementations already create a distinct UUID for each account. Some commercial email clients have a license key. Some physical devices that need to interact with IMAP might have a unique hardware ID. While there is no pre-defined list of client identity type defined by this RFC, and all IMAP servers should be prepared to accept any form of client identity type that conforms to the definition, it is suggested that IMAP client developers carefully consider the name of the client identity type. For example, rather than using a client identity type of UUID, consider the advantages of making it more distinct, e.g. "<product_short_code>UUID". This way the IMAP server can better record histories, eg the difference between say a Thunderbird generated unique id, and a Mutt generated unique id.

Some examples of identity type might be UUID, LICENSE, DEVICE_ID, and/or COOKIE. It is expected that the most common types might be related to distinct UUID, LICENSEKEY, or HARDWAREID.

An IMAP server SHOULD NOT reject an unidentified CLIENTID type, except for specific policy use cases.

It is envisioned that in the future it will be useful to propose a set of standardized client-identity-type to help with validation, or to allow the IMAP server to apply ACL rules on expected types, this would be an extension to this RFC.

1. UUID

UUID is a common practice to represent either a individual user, hardware device or software installation associated with a specific individual. The support of UUID enables existing UUID implementations to be used to semi-uniquely identify a device associated with an individual. A definition of the format should be considered. Otherwise non-standard UUID might be a separate type specific to the software implementation, for instance TBIRD-UUID.

2. LICENSE

An IMAP client may find it useful to identify the license key of software it is using. Such licenses are typically crafted such that they are unique and useful to identify a software installation. This is more normally suited for a software designed for a single-user. While LICENSE could be standard type again, it might more more helpful to specify a vendor specific type such as BBLICENSEKEY.

3. DEVICE_ID

Many hardware devices are designed to be used by a single individual and already have an associated hardware device id.

While a standard type might be defined, it also might be more helpful to use a vendor specific type, such as ATOM-DEVICEID.

4. COOKIE

While not guaranteed to be consistent many web applications are designed to access IMAP directly and may need to have a semi-unique identifier available as part of the web based transaction. It is assumed that COOKIE encompasses the group of web based tokens known to persist from session to session. A specific web based application can provide sufficient information in the actual client-identifier-token to differentiate between applications and or websites, and are convenient as they can be related to very specific domains, and are universally available to web application designers.

As a reminder, an IMAP server SHOULD NOT retain and/or store the CLIENTID information WITH authentication credentials or authentication systems directly, but the IMAP service MAY associate the CLIENTID with a specific account holder, eg to create a history file of known CLIENTID tokens associated or permitted to access or present authentication credentials for that account holder.

This document recommends that an IMAP server handle any given client identity type from a CLIENTID command in one or more of the following manners.

1. Handled but treat as not presented (ignored, no persistence)
2. Store in IMAP session but treat as not presented (debugging)
3. Store in the IMAP session, so it is available to System log
4. Store in the IMAP session, so it is available to User log
5. Use for authentication
6. Use for alert when authentication fails
7. Use for alert when authentication succeeds
8. Unused

7. Examples

7.1. UUID as Client Identity

```
C: [connection established over a plaintext connection]
C: a001 CAPABILITY
S: * CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI LOGINDISABLED
S: a001 OK CAPABILITY completed
C: a002 STARTTLS
S: a002 OK STARTTLS completed
<TLS negotiation, further commands are under [TLS] layer>
C: a003 CAPABILITY
S: * CAPABILITY IMAP4rev1 AUTH=GSSAPI AUTH=PLAIN CLIENTID
S: a003 OK CAPABILITY completed
C: a004 CLIENTID UUID 23bf83be-aad7-46aa-9e0f-39191ccf402f
S: a004 OK CLIENTID completed
C: a005 LOGIN joe password
S: a005 OK LOGIN completed
```

7.2. Malformed CLIENTID Command

```
C: [connection established over a plaintext connection]
C: a001 CAPABILITY
S: * CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI LOGINDISABLED
S: a001 OK CAPABILITY completed
C: a002 STARTTLS
S: a002 OK STARTTLS completed
<TLS negotiation, further commands are under [TLS] layer>
C: a003 CAPABILITY
S: * CAPABILITY IMAP4rev1 AUTH=GSSAPI AUTH=PLAIN CLIENTID
S: a003 OK CAPABILITY completed
C: a004 CLIENTID UUID
S: a004 BAD Error in IMAP command received by server
```

The IMAP server rejects the CLIENTID command as it is not well formed due to there being only a single parameter provided.

7.3. Client Identity without TLS/SSL Session

```
C: [connection established over a plaintext connection]
C: a001 CAPABILITY
S: * CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI LOGINDISABLED
S: a001 OK CAPABILITY completed
C: a002 CLIENTID UUID 23bf83be-aad7-46aa-9e0f-39191ccf402f
S: a002 BAD Unknown IMAP command received by server
```

The IMAP server rejects use of the CLIENTID command as the CLIENTID capability had not been advertised because no encryption was negotiated between the IMAP server and IMAP client.

7.4. Client Identity Leading to Rejection

```
C: [connection established over a plaintext connection]
C: a001 CAPABILITY
S: * CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI LOGINDISABLED
S: a001 OK CAPABILITY completed
C: a002 STARTTLS
S: a002 OK STARTTLS completed
<TLS negotiation, further commands are under [TLS] layer>
C: a003 CAPABILITY
S: * CAPABILITY IMAP4rev1 AUTH=GSSAPI AUTH=PLAIN CLIENTID
S: a003 OK CAPABILITY completed
C: a004 CLIENTID UUID 23bf83be-aad7-46aa-9e0f-39191ccf402f
S: a004 OK CLIENTID completed
C: a005 LOGIN joe password
S: a005 BAD Failed to authenticate
```

The IMAP server rejects use of the system during the LOGIN command after deciding that the provided client identity does not establish sufficient privileges. Note that the error message that's returned to the client is very generic and does not reveal any information about CLIENTID and/or the existence of 'joe' and/or the validity of the password.

8. Security Considerations

As this extension provides an additional means of communicating information from a client to a server it is clear there is additional information divulged to the server. This may have privacy considerations depending on the client identity type or its contents. For example, it may reveal a MAC address of the device used to communicate with a server that would not previously have been revealed. While it has been useful to use identifier such as email address for authentication it is easy for these authentication tokens to be shared and/or reused and/or be publically available for other purposes. An IMAP server and or its operators SHOULD not share any CLIENTID information presented with a third party as it may represent or be linked to an individual and SHOULD never be shared in association with authentication tokens.

As well, while this service extension requires that the identity information only be transmitted over an encrypted channel to reduce the risk of eavesdropping, it does not specify any policies or practices required in the establishment of such a channel, and so it is the responsibility of the client and the server to determine that the communication medium meets their requirements.

9. IANA Considerations

The IANA is requested to add CLIENTID to the "IMAP 4 Capabilities" registry, <http://www.iana.org/assignments/imap4-capabilities>.

10. References

10.1. Normative References

- [ABNF] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.
- [IMAP] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, March 2003.
- [KEYWORDS] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [SASL] Myers, J., "Simple Authentication and Security Layer (SASL)", RFC 2222, October 1997.
- [TLS] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

Appendix A. CLIENTID Product Support

Since publishing the IMAP Client Identity RFC draft, multiple email server and client vendors have implemented CLIENTID support into their products, e.g. MailEnable, MagicMail, SaneBox, BlueMail, emClient, and Thunderbird.

Contributors

Michael Peddemors
LinuxMagic

Authors' Addresses

Deion Yu
LinuxMagic
#405 - 860 Homer St.
Vancouver, British Columbia
CA V6B 2W5

EMail: deiony@linuxmagic.com

