Real Time Messaging Protocol Chunk Stream
draft-rtmpcs-01.txt

Abstract

This memo describes the Real Time Messaging Protocol Chunk Stream(RTMP Chunk Stream), an application-level protocol designed for multiplexing and packetizing multimedia transport streams (such as audio, video, and interactive content) over a suitable transport protocol (such as TCP).

Table of Contents

1. Introduction

The document specifies the Real Time Messaging Protocol Chunk Stream(RTMP Chunk Stream). It provides multiplexing and packetizing services for a higher-level multimedia stream protocol.

While RTMP Chunk Stream was designed to work with the Real Time Messaging Protocol [RTMP], it can handle any protocol that sends a stream of messages. Each message contains timestamp and payload type identification. RTMP Chunk Stream and RTMP together are suitable for a wide variety of audio-video applications, from one-to-one and one-to-many live broadcasting to video-on-demand services to interactive conferencing applications.

When used with a reliable transport protocol such as [TCP], RTMP Chunk Stream provides guaranteed timestamp-ordered end-to-end delivery of all messages, across multiple streams. RTMP Chunk Stream does not provide any prioritization or similar forms of control, but can be used by higher-level protocols to provide such prioritization. For example, a live video server might choose to drop video messages for a slow client to ensure that audio messages are received in a timely fashion, based on either the time to send or the time to acknowledge each message.

RTMP Chunk Stream includes its own in-band protocol control messages, and also offers a mechanism for the higher-level protocol to embed user control messages.

1.1. Terminology

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 [BCP14], [RFC2119].

2. Definitions

Payload:
The data contained in a packet, for example audio samples or compressed video data. The payload format and interpretation are beyond the scope of this document.

Packet:
A data packet consists of fixed header and payload data. Some underlying protocols may require an encapsulation of the packet to be defined.

Port:
The "abstraction that transport protocols use to distinguish among multiple destinations within a given host computer. TCP/IP protocols identify ports using small positive integers." The transport selectors (TSEL) used by the OSI transport layer are equivalent to ports.

Transport address:
The combination of a network address and port that identifies a transport-level endpoint, for example an IP address and a TCP port. Packets are transmitted from a source transport address to a destination transport address.

Message stream:
A logical channel of communication that allows the flow of messages.

Message stream ID:
Each message has an ID associated with it to identify the message stream in which it is flowing.

Chunk:
A fragment of a message. The messages are broken into smaller parts and interleaved before they are sent over the network. The chunks ensure timestamp-ordered end-to-end delivery of all messages, across multiple streams.

Chunk stream:
A logical channel of communication that allows flow of chunks in a particular direction. The chunk stream can travel from the client to the server and reverse.

Chunk stream ID:
Every chunk has an ID associated with it to identify the chunk stream in which it is flowing.

Multiplexing:
Process of making separate audio/video data into one coherent audio/video stream, making it possible to transmit several video and audio simultaneously.

DeMultiplexing:
Reverse process of multiplexing, in which interleaved audio and video data are assembled to form the original audio and video data.

3. Byte Order, Alignment, and Time Format

All integer fields are carried in network byte order, byte zero is the first byte shown, and bit zero is the most significant bit in a word or field. This byte order is commonly known as big-endian. The transmission order is described in detail in [STD5]. Unless otherwise noted, numeric constants in this document are in decimal (base 10).

Except as otherwise specified, all data in RTMP Chunk Stream is byte-aligned; for example, a 16-bit field may be at an odd byte offset. Where padding is indicated, padding bytes SHOULD have the value zero.

Timestamps in RTMP Chunk Stream are given as an integer number of milliseconds, relative to an unspecified epoch. Typically, each Chunk Stream will start with a timestamp of 0, but this is not required, as long as the two endpoints agree on the epoch. Note that this means that any synchronization across multiple chunk streams (especially from separate hosts) requires some additional mechanism outside of RTMP Chunk Stream.

Timestamps MUST be monotonically increasing, and SHOULD be linear in time, to allow applications to handle synchronization, bandwidth measurement, jitter detection, and flow control.

Because timestamps are generally only 32 bits long, they will roll over after fewer than 50 days. Because streams are allowed to run continuously, potentially for years on end, an RTMP Chunk Stream application MUST use modular arithmetic for subtractions and comparisons, and SHOULD be capable of handling this wraparound heuristically. Any reasonable method is acceptable, as long as both endpoints agree. An application could assume, for example, that all adjacent timestamps are within 2^31 milliseconds of each other, so 10000 comes after 4000000000, while 3000000000 comes before 4000000000.

Timestamp deltas are also specified as an unsigned integer number of milliseconds, relative to the previous timestamp. Timestamp deltas may be either 24 or 32 bits long.

4. Message Format

The format of a message that can be split into chunks to support multiplexing, depends on higher level protocol. The message format SHOULD however contain the following fields which are necessary for creating the chunks.

Timestamp:
Timestamp of the message. This field can transport 4 bytes.

Length:
Length of the message payload. If the message header cannot be elided, it should be included in the length. This field occupies 3 bytes in the chunk header.

Type Id:
A range of type IDs are reserved for protocol control messages. These messages which propagate information are handled by both RTMP Chunk Stream protocol and the higher-level protocol. All other type IDs are available for use by the higher-level protocol, and treated as opaque values by RTMP Chunk Stream. In fact, nothing in RTMP Chunk Stream requires these values to be used as a type; all (non-protocol) messages could be of the same type, or the application could use this field to distinguish simultaneous tracks rather than types. This field occupies 1 byte in the chunk header.

Message Stream ID:
The message stream ID can be any arbitrary value. Different message streams multiplexed onto the same chunk stream are demultiplexed based on their message stream IDs. Beyond that, as far as RTMP Chunk Stream is concerned, this is an opaque value. This field occupies 4 bytes in the chunk header in little endian format.

5. Handshake

An RTMP connection begins with a handshake. The handshake is unlike the rest of the protocol; it consists of three static-sized chunks rather than consisting of variable-sized chunks with headers.

The client (the endpoint that has initiated the connection) and the server each send the same three chunks. For exposition, these chunks will be designated C0, C1, and C2 when sent by the client; S0, S1, and S2 when sent by the server.

5.1. Handshake sequence

The handshake begins with the client sending the C0 and C1 chunks.

The client MUST wait until S1 has been received before sending C2. The client MUST wait until S2 has been received before sending any other data.

The server MUST wait until C0 has been received before sending S0 and S1, and MAY wait until after C1 as well. The server MUST wait until C1 has been received before sending S2. The server MUST wait until C2 has been received before sending any other data.

5.2. C0 and S0 Format

The C0 and S0 packets are a single octet, treated as a single 8-bit integer field:

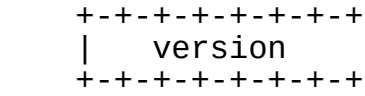


Figure 1 C0 and S0 bits

Following are the fields in the C0/S0 packets:

Version: 8 bits
In C0, this field identifies the RTMP version requested by the client. In S0, this field identifies the RTMP version selected by the server. The version defined by this specification is 3. Values 0-2 are deprecated values used by earlier proprietary products; 4-31 are reserved for future implementations; and 32-255 are not allowed (to allow distinguishing RTMP from text-based protocols, which always start with a printable character). A server that does not recognize the client's requested version SHOULD respond with 3. The client MAY choose to degrade to version 3, or to abandon the handshake.

5.3. C1 and S1 Format

The C1 and S1 packets are 1536 octets long, consisting of the following fields:

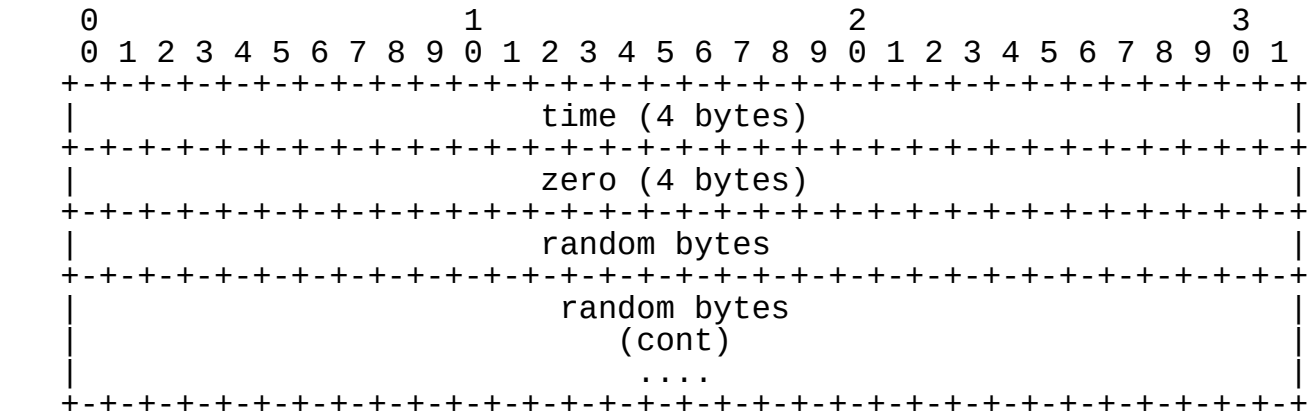


Figure 2 C1 and S1 bits

Time: 4 bytes
This field contains a timestamp, which SHOULD be used as the epoch for all future chunks sent from this endpoint. This may be 0, or some arbitrary value. To synchronize multiple chunkstreams, the endpoint may wish to send the current value of the other chunkstream's timestamp.

Zero: 4 bytes
This field MUST be all 0s.

Random data: 1528 bytes
This field can contain any arbitrary values. Since each endpoint has to distinguish between the response to the handshake it has initiated and the handshake initiated by its peer,this data SHOULD send something sufficiently random. But there is no need for cryptographically-secure randomness, or even dynamic values.

5.4. C2 and S2 Format

The C2 and S2 packets are 1536 octets long, and nearly an echo of S1 and C1 (respectively), consisting of the following fields:

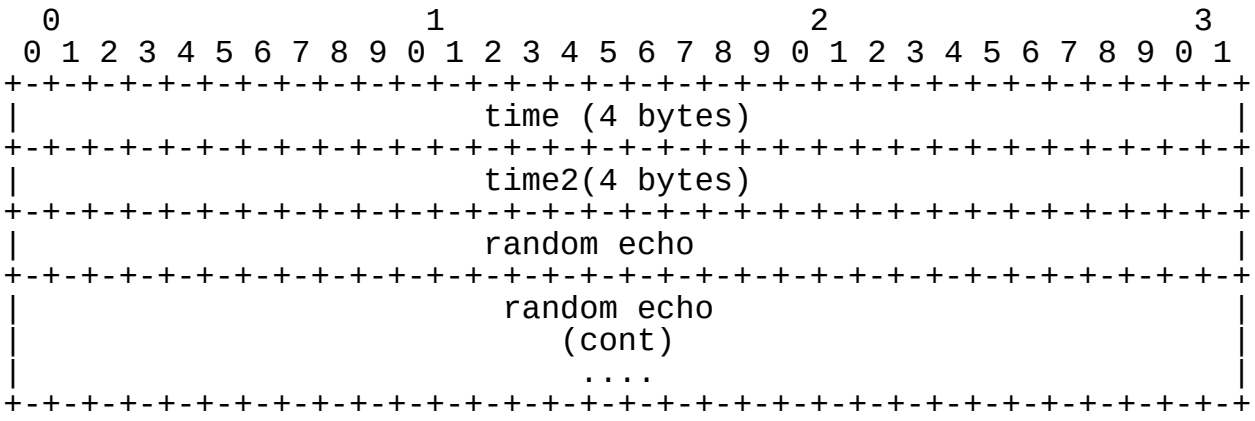


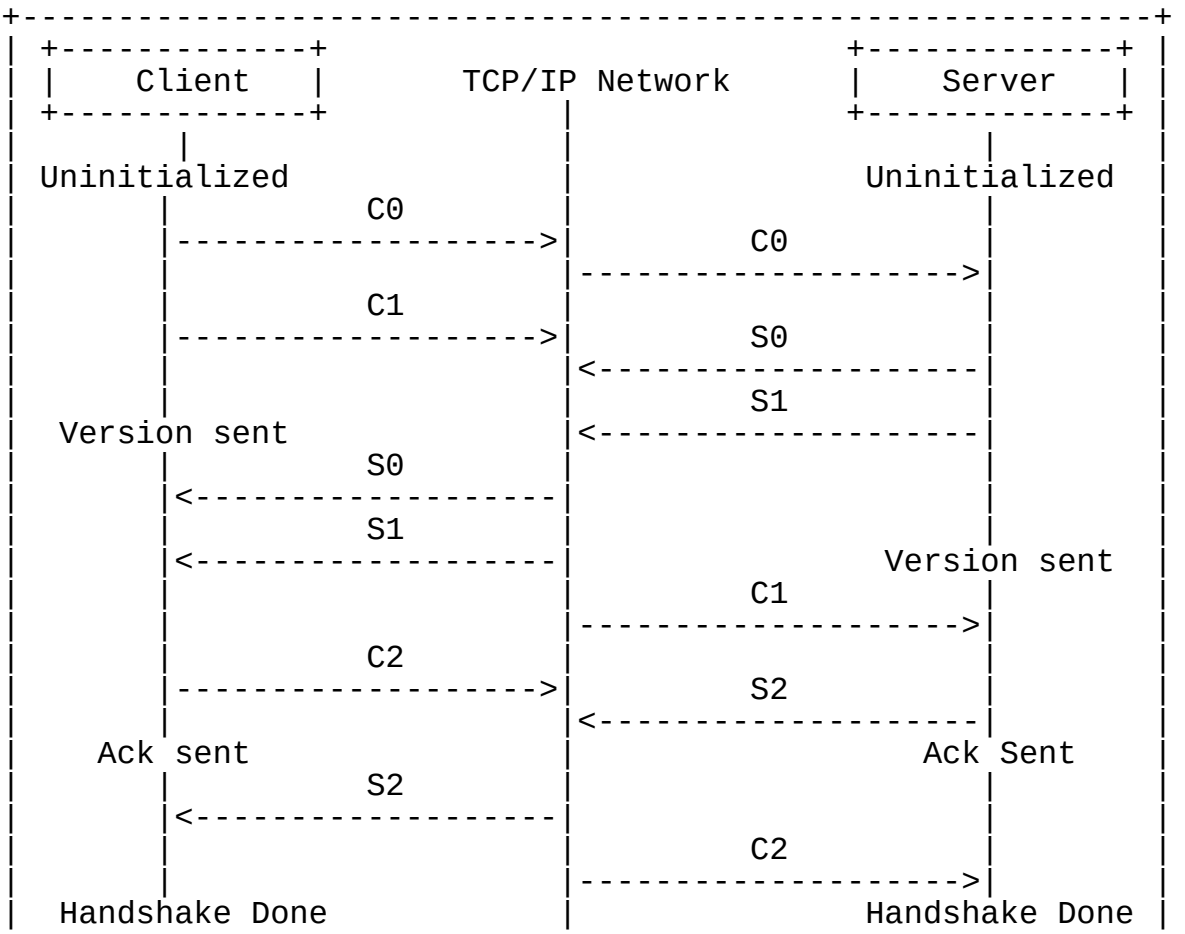
Figure 3 C2 and S2 bits

Time: 4 bytes
This field MUST contain the timestamp sent by the peer in S1 (for C2) or C1 (for S2).

Time2: 4 bytes
This field MUST contain the timestamp at which the previous packet(s1 0r c1) sent by the peer was read.

Random echo: 1528 bytes
This field MUST contain the random data field sent by the peer in S1 (for C2) or S2 (for C1).
Either peer can use the time and time2 fields together with the current timestamp as a quick estimate of the bandwidth and/or latency of the connection, but this is unlikely to be useful.

5.5. Handshake Diagram



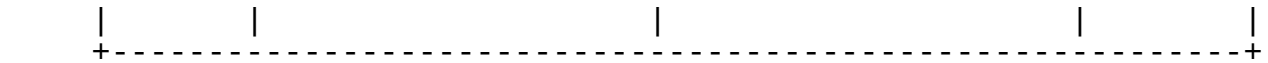


Figure 4 Pictorial Representation of Handshake

The following table describes the states mentioned in the hand shake diagram:

States	Description
Uninitialized	The protocol version is sent during this stage. Both the client and server are uninitialized. The The client sends the protocol version in packet C0. If the server supports the version, it sends S0 and S1 in response. If not, the server responds by taking the appropriate action. In RTMP, this action is terminating the connection.
Version Sent	Both client and server are in the Version Sent state after the Uninitialized state. The client is waiting for the packet S1 and the server is waiting for the packet C1. On receiving the awaited packets, the client sends the packet C2 and the server sends the packet S2. The state then becomes Ack Sent.
Ack Sent	The client and the server wait for S2 and C2, respectively.
HandshakeDone	The client and the server exchange messages.

6. Chunking

After handshaking, the connection multiplexes one or more chunk streams. Each chunk stream carries messages of one type from one message stream. Each chunk that is created has a unique ID associated with it called chunk stream ID. The chunks are transmitted over the network. While transmitting, each chunk must be sent in full before the next chunk. At the receiver end, the chunks are assembled into messages based on the chunk stream ID.

Chunking allows large messages at the higher-level protocol to be broken down into smaller messages,for example, to prevent large low-priority messages from blocking smaller high-priority messages.

Chunking also allows small messages to be sent with less overhead, as the chunk header contains a compressed representation of information that would otherwise have to be included in the message itself.

The chunk size is configurable. It can be set using a control message(Set Chunk Size) as described in section 7.1. The maximum chunk size can be 65536 bytes and minimum 128 bytes. Larger values reduce CPU usage, but also commit to larger writes that can delay other content on lower bandwidth connections. Smaller chunks are not good for high-bit rate streaming. Chunk size is maintained independently for each direction.

6.1. Chunk Format

Each chunk consists of a header and data. The header itself is broken down into three parts:

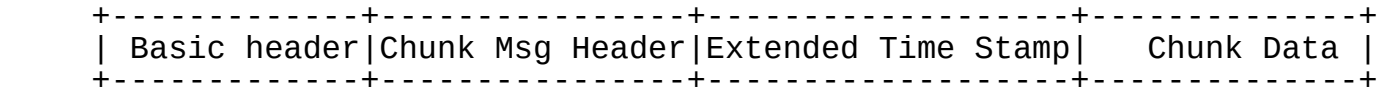


Figure 5 Chunk Format.

Chunk basic header: 1 to 3 bytes

This field encodes the chunk stream ID and the chunk type. Chunk type determines the format of the encoded message header. The length depends entirely on the chunk stream ID, which is a variable-length field.

Chunk message header: 0, 3, 7, or 11 bytes

This field encodes information about the message being sent (whether in whole or in part). The length can be determined using the chunk type specified in the chunk header.

Extended timestamp: 0 or 4 bytes

This field MUST be sent when the normal timsestamp is set to 0xffffff, it MUST NOT be sent if the normal timestamp is set to anything else. So for values less than 0xffffff the normal timestamp field SHOULD be used in which case the extended timestamp

MUST NOT be present. For values greater than or equal to 0xffffff the normal timestamp field MUST NOT be used and MUST be set to 0xffffff and the extended timestamp MUST be sent.

6.1.1. Chunk Basic Header

The Chunk Basic Header encodes the chunk stream ID and the chunk type(represented by fmt field in the figure below). Chunk type determines the format of the encoded message header. Chunk Basic Header field may be 1, 2, or 3 bytes, depending on the chunk stream ID.

An implementation SHOULD use the smallest representation that can hold the ID.

The protocol supports up to 65597 streams with IDs 3–65599. The IDs 0, 1, and 2 are reserved. Value 0 indicates the ID in the range of 64–319 (the second byte + 64). Value 1 indicates the ID in the range of 64–65599 ((the third byte)*256 + the second byte + 64). Value 2 indicates its low-level protocol message. There are no additional bytes for stream IDs. Values in the range of 3–63 represent the complete stream ID. There are no additional bytes used to represent it.

The bits 0–5 (least significant) in the chunk basic header represent the chunk stream ID.

Chunk stream IDs 2-63 can be encoded in the 1-byte version of this field.

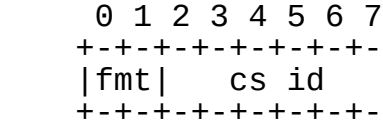


Figure 6 Chunk basic header 1

Chunk stream IDs 64-319 can be encoded in the 2-byte version of this field. ID is computed as (the second byte + 64).

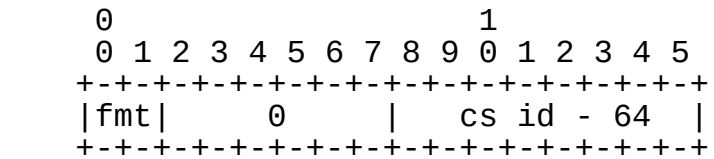


Figure 7 Chunk basic header 2

Chunk stream IDs 64-65599 can be encoded in the 3-byte version of this field. ID is computed as ((the third byte)*256 + the second byte + 64).

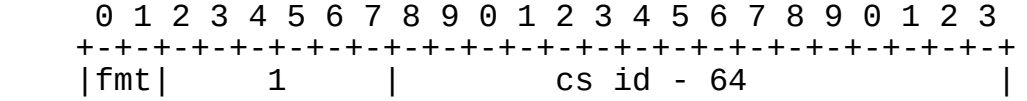


Figure 8 Chunk basic header 3

cs id: 6 bits
This field contains the chunk stream ID, for values from 2-63. Values 0 and 1 are used to indicate the 2- or 3-byte versions of this field.

fmt: 2 bits
This field identifies one of four format used by the ‘chunk message header’.The ‘chunk message header’ for each of the chunk types is explained in the next section.

cs id - 64: 8 or 16 bits
This field contains the chunk stream ID minus 64. For example, ID 365 would be represented by a 1 in cs id, and a 16-bit 301 here.

Chunk stream IDs with values 64-319 could be represented by both 2-byte version and 3-byte version of this field.

6.1.2. Chunk Message Header

There are four different formats for the chunk message header, selected by the "fmt" field in the chunk basic header.

An implementation SHOULD use the most compact representation possible for each chunk message header.

6.1.2.1. Type 0

Chunks of Type 0 are 11 bytes long. This type MUST be used at the start of a chunk stream, and whenever the stream timestamp goes backward (e.g., because of a backward seek).

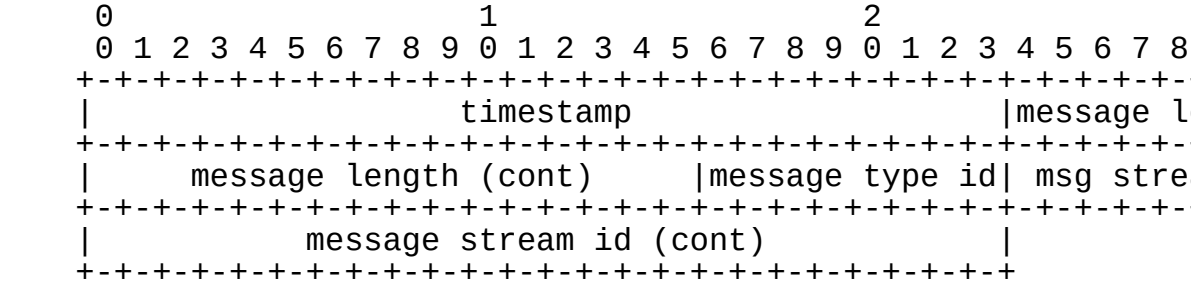


Figure 9 Chunk Message Header – Type 0

timestamp: 3 bytes
For a type-0 chunk, the absolute timestamp of the message is sent here. If the timestamp is greater than or equal to 16777215 (hexadecimal 0x00ffffff), this value MUST be 16777215, and the 'extended timestamp header' MUST be present. Otherwise, this value SHOULD be the entire timestamp.

6.1.2.2. Type 1

Chunks of Type 1 are 7 bytes long. The message stream ID is not included; this chunk takes the same stream ID as the preceding chunk. Streams with variable-sized messages (for example, many video formats) SHOULD use this format for the first chunk of each new message after the first.

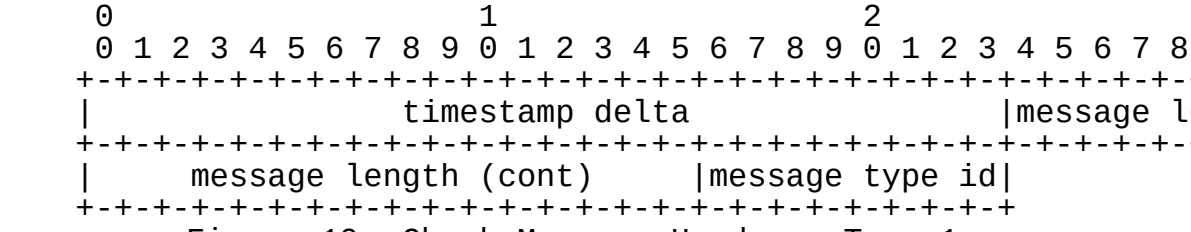


Figure 10 Chunk Message Header – Type 1

6.1.2.3. Type 2

Chunks of Type 2 are 3 bytes long. Neither the stream ID nor the message length is included; this chunk has the same stream ID and message length as the preceding chunk. Streams with constant-sized messages (for example, some audio and data formats) SHOULD use this format for the first chunk of each message after the first.

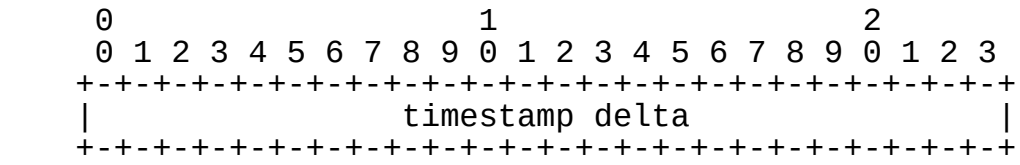


Figure 11 Chunk Message Header – Type 2

6.1.2.4. Type 3

Chunks of Type 3 have no header. Stream ID, message length and timestamp delta are not present; chunks of this type take values from the preceding chunk. When a single message is split into chunks, all chunks of a message except the first one, SHOULD use this type. Refer to example 2 in section 6.2.2. Stream consisting of messages of exactly the same size, stream ID and spacing in time SHOULD use this type for all chunks after chunk of Type 2. Refer to example 1 in section 6.2.1. If the delta between the first message and the second message is same as the time stamp of first message, then chunk of type 3 would immediately follow the chunk of type 0 as there is no need for a chunk of type 2 to register the delta. If Type 3 chunk follows a Type 0 chunk, then timestamp delta for this Type 3 chunk is the same as the timestamp of Type 0 chunk.

Description of each field in the chunk message header.

timestamp delta: 3 bytes
For a type-1 or type-2 chunk, the difference between the previous chunk's timestamp and the current chunk's timestamp is sent here. If the delta is greater than or equal to 16777215 (hexadecimal 0x00ffffff), this value MUST be 16777215, and the 'extended timestamp header' MUST be present. Otherwise, this value SHOULD be the entire delta.

message length: 3 bytes
For a type-0 or type-1 chunk, the length of the message is sent here.

Note that this is generally not the same as the length of the chunk payload. The chunk payload length is the maximum chunk size for all but the last chunk, and the remainder (which may be the entire length, for small messages) for the last chunk.

message type id: 1 byte
For a type-0 or type-1 chunk, type of the message is sent here.

message stream id: 4 bytes
For a type-0 chunk, the message stream ID is stored. Message stream ID is stored in little-endian format. Typically, all messages in the same chunk stream will come from the same message stream. While it is possible to multiplex separate message streams into the same chunk stream, this defeats all of the header compression. However, if one message stream is closed and another one subsequently opened, there is no reason an existing chunk stream cannot be reused by sending a new type-0 chunk.

6.1.3. Extended Timestamp

This field is transmitted only when the normal time stamp in the chunk message header is set to 0x00ffffff. If normal time stamp is set to any value less than 0x00ffffff, this field MUST NOT be present. This field MUST NOT be present if the timestamp field is not present. Type 3 chunks MUST NOT have this field.

This field if transmitted is located immediately after the chunk message header and before the chunk data.

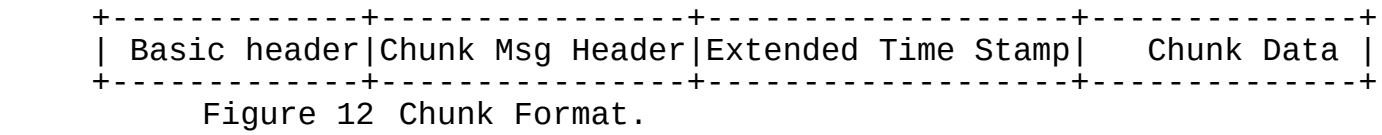


Figure 12 Chunk Format.

6.2. Examples

6.2.1. Example 1

Example 1 shows a simple stream of audio messages. This example demonstrates the redundancy of information.

	Message Stream ID	Message TYpe ID	Time	Length
Msg # 1	12345	8	1000	32
Msg # 2	12345	8	1020	32
Msg # 3	12345	8	1040	32
Msg # 4	12345	8	1060	32

Figure 13 Sample Audio messages to be made into chunks

The next table shows chunks produced in this stream. From message 3 onward, data transmission is optimized. There is only 1 byte of overhead per message beyond this point.

	Chunk Stream ID	Chunk Type	Header Data	No.of Bytes After Header	Total No.of Bytes in the Chunk
Chunk#1	3	0	delta: 1000 length: 32, type: 8, stream ID: 12345 (11 bytes)	32	44
Chunk#2	3	2	20 (3 bytes)	32	36
Chunk#3	3	3	none (0 bytes)	32	33
Chunk#4	3	3	none (0 bytes)	32	33

Figure 14 Format of each of the chunks of audio messages above

6.2.2. Example 2

Example 2 illustrates a message that is too long to fit in a 128-chunk and is broken into several chunks.

	Message Stream ID	Message TYpe ID	Time	Length
Msg # 1	12346	9 (video)	1000	307

Figure 15 Sample Message to be broken to chunks

Here are the chunks that are produced:

	Chunk Stream ID	Chunk Type	Header Data	No. of Bytes after Header	Total No. of bytes in the chunk
Chunk#1	4	0	delta: 1000 length: 307 type: 9, stream ID: 12346 (11 bytes)	128	140
Chunk#2	4	3	none (0 bytes)	128	129
Chunk#3	4	3	none (0 bytes)	51	52

Figure 16 Format of each of the broken chunk.

The header data of chunk 1 specifies that the overall message is 307 bytes.

Notice from the two examples, that chunk type 3 can be used in two different ways. The first is to specify the continuation of a message. The second is to specify the beginning of a new message whose header can be derived from the existing state data.

7. Protocol Control Messages

RTMP Chunk Stream supports some protocol control messages. These messages contain information required by RTMP Chunk Stream protocol and will not be propagated to the higher protocol layers. Currently there are two protocol messages used in RTMP Chunk Stream. One protocol message is used for setting the chunk size and the other is used to abort a message due to non-availability of remaining chunks

Protocol control messages SHOULD have message stream ID 0(called as control stream) and chunk stream ID 2, and are sent with highest priority.

Each protocol control message type has a fixed-size payload, and is always sent in a single chunk.

7.1. Set Chunk Size

Protocol control message 1, Set Chunk Size, is used to notify the peer about the new maximum chunk size.

A default value can be set for the chunk size, but the client or the server can change this value and update it to the peer. For example, suppose a client wants to send 131 bytes of audio data and the chunk size is 128. In this case, the client can send this protocol message to the server to notify that the chunk size is set to 131 bytes. The client can then send the audio data in a single chunk.

The maximum chunk size can be 65536 bytes. The chunk size is maintained independently for each direction.

7.2. Abort Message

This protocol control message is used to notify the peer if it is waiting for chunks to complete a message, then to discard the partially received message over a chunk stream. The peer receives the chunk stream ID as this protocol message’s payload. An application may send this message when closing in order to indicate that further processing of the messages is not required.

8. References

8.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Crocker, D. and Overell, P.(Editors), "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, Internet Mail Consortium and Demon Internet Ltd., November 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2234] Crocker, D. and Overell, P.(Editors), "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, Internet Mail Consortium and Demon Internet Ltd., November 1997.

8.2. Informative References

[3] Faber, T., Touch, J. and W. Yue, "The TIME-WAIT state in TCP and Its Effect on Busy Servers", Proc. Infocom 1999 pp. 1573-1583.

[Fab1999] Faber, T., Touch, J. and W. Yue, "The TIME-WAIT state in TCP and Its Effect on Busy Servers", Proc. Infocom 1999 pp. 1573-1583.

9. Acknowledgments

Address:

Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704