RTMP Specification License

Published April 2009

This is a legal agreement ("Agreement") between the user of the Specification (either an individual or an entity) ("You"), and Adobe Systems Incorporated ("Adobe") (collectively the "Parties"). If You want a license from Adobe to implement the RTMP Specification (as defined below), You must agree to these terms. This is an offer to be accepted only on the terms set forth in this Agreement. You accept these terms by accessing, downloading, using, or reviewing the RTMP Specification. Once accepted, Adobe grants You the license below, provided that You fully comply with these terms. If You do not agree to these terms, then You are not licensed under this Agreement. If You attempt to alter or modify these terms, then the offer is no longer valid and is revoked.

The RTMP Specification provides a protocol for high-performance streaming transmissions of audio, video, and data content between Adobe Flash Platform technologies. We offer this license to encourage streaming rich content via the RTMP protocol.

The RTMP Specification may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without written approval from Adobe. Notwithstanding the foregoing, upon acceptance of the terms of this Agreement, You may print out one copy of this manual for personal use provided that no part of this manual may be printed out, reproduced, distributed, resold, or transmitted for any other purposes, including, without limitation, commercial purposes, such as selling copies of this documentation or providing paid-for support services.

Definitions
"Compliant Implementation" means the portion of an application, product, or service that defines, creates or processes data compliant with the requirements expressly stated in the RTMP Specification.

"Essential Claim" means a claim of a patent that is necessarily infringed in order to achieve a Compliant Implementation. A claim is necessarily infringed only when there is no technically reasonable way to avoid infringement of that claim when following the requirements of the Specification to make a Compliant Implementation.

"RTMP Specification" means the specifications found at http://www.adobe.com/devnet/rtmp/. Only the RTMP Specification published by Adobe will be considered the RTMP Specification for purposes of this Agreement. Any specification not published by Adobe or one that incorporates the RTMP Specification in part, in whole or by reference shall not be considered the RTMP Specification for purposes of this Agreement.

Patent License
Upon your acceptance of these terms, Adobe grants You a non-exclusive, royalty-free, non-transferable, non-sublicensable, personal, worldwide license under Adobe's Essential Claims to make, have made, use, sell, offer to sell, import and distribute Compliant Implementations.

Prohibited Uses
The rights and licenses granted by Adobe in the RTMP Specification, including those granted in the Patent License, are conditioned upon Your agreement to use the RTMP Specification for only streaming video, audio and/or data content and not to make, have made, use, sell, offer to sell, import or distribute: (i) any technology that intercepts streaming video, audio and/or data content for storage in any device or medium; or (ii) any technology that circumvents technological measures for the protection of audio, video and/or data content, including any of Adobe's secure RTMP measures. No right or license to any Adobe intellectual property is granted for such prohibited uses.

Defensive Suspension
If You assert, threaten to assert, or participate in the assertion of a lawsuit, proceeding, claim or similar action directed either (i) against any other person or entity, including Adobe, claiming that a Compliant Implementation infringes a patent, or (ii) against Adobe, claiming that any Adobe product infringes a patent, then Adobe may at its discretion terminate all license grants and any other rights provided under this Agreement to You.

Reservation of Rights
All rights not expressly granted herein are reserved.

Trademarks
Adobe, ActionScript, and Flash are either registered trademarks or trademarks of Adobe and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words, or phrases mentioned within this publication may be trademarks, service marks, or trade names of Adobe or other entities and may be registered in certain jurisdictions including internationally. No right or license is granted to any Adobe trademark.

Third-Party Information
This guide contains links to third-party websites that are not under the control of Adobe, and Adobe is not responsible for the content on any linked site. If You access a third-party website mentioned in this guide, then You do so at Your own risk. Adobe provides these links only as a convenience, and the inclusion of the link does not imply that Adobe endorses or accepts any responsibility for the content on those third-party sites. No right, license or interest is granted in any third party technology referenced in this guide.

No Warranty

THE RTMP SPECIFICATION AND THE LICENSES GRANTED ABOVE ARE PROVIDED "AS IS" SUBJECT TO CHANGE BY ADOBE WITHOUT NOTICE, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE OR NON-INFRINGEMENT. NOTHING IN THIS DOCUMENT SHALL BE CONSTRUED AS A COMMITMENT BY ADOBE INCLUDING A COMMITMENT FOR MAINTENANCE OF ANY ADOBE PATENT, A WARRANTY OR REPRESENTATION AS TO THE VALIDITY OR SCOPE OF ANY ADOBE PATENT, AN AGREEMENT TO PROTECT OR COOPERATE WITH ANY PARTY OR TO BRING OR PROSECUTE ACTIONS AGAINST ANY PARTY, OR A GRANT OF ANY RIGHT UNDER ANY ADOBE PATENT CLAIM OTHER THAN AN ESSENTIAL CLAIM. ADOBE SHALL HAVE NO DUTY TO INDEMNIFY, HOLD HARMLESS OR DEFEND YOU OR ANY THIRD PARTY FROM AND AGAINST ANY LOSS, DAMAGE, LAWSUITS, PROCEEDINGS, CLAIMS OR SIMILAR ACTIONS THAT ARISE OR RESULT FROM THE USE OF THE RTMP SPECIFICATION OR THE DEVELOPMENT, USE, MANUFACTURE, OFFER TO SELL, SALE, IMPORTATION, OR DISTRIBUTION OF ANY COMPLIANT IMPLEMENTATION.

Limitation Of Liability.

ADOBE SHALL NOT BE LIABLE FOR ANY DAMAGES ARISING FROM OR RELATED TO THIS AGREEMENT, INCLUDING INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR SPECIAL DAMAGES EVEN IF ADOBE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES IN ADVANCE.

Governing Law

This Agreement shall be construed in accordance with, and all disputes hereunder shall be governed by, the laws of the State of California. The Parties consent to exclusive jurisdiction and venue for any dispute hereunder in San Jose, California.

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:

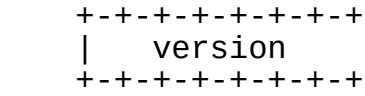


Figure 1 C0 and S0 bits

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:

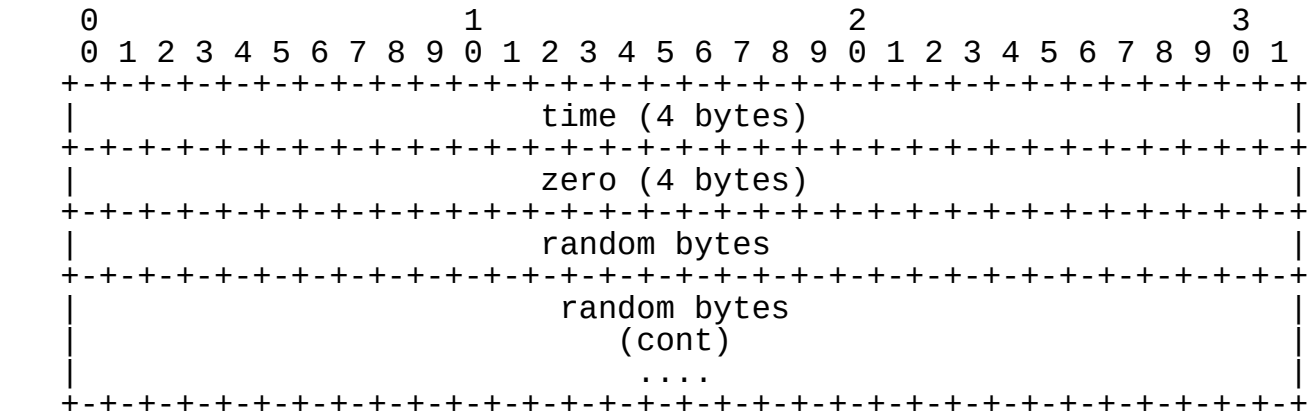


Figure 2 C1 and S1 bits

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:

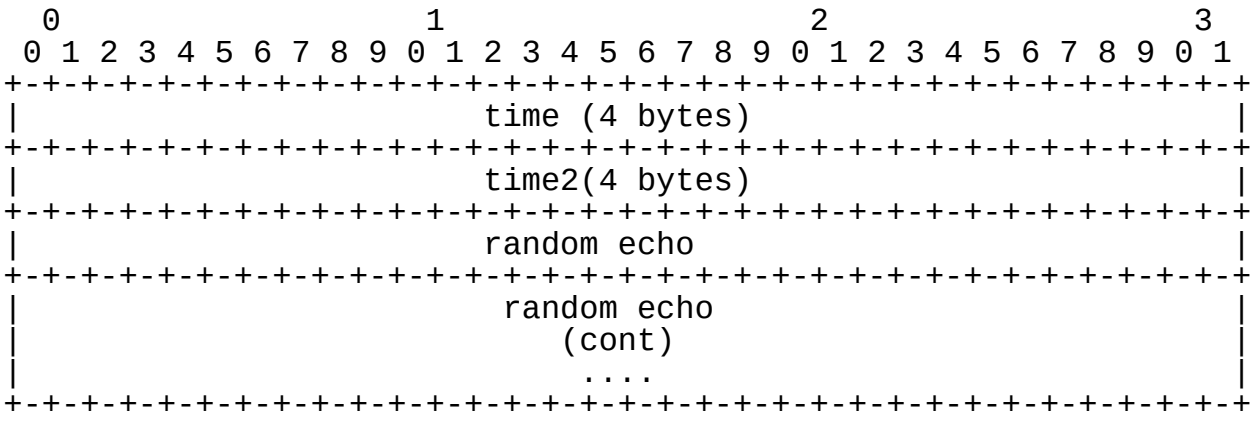


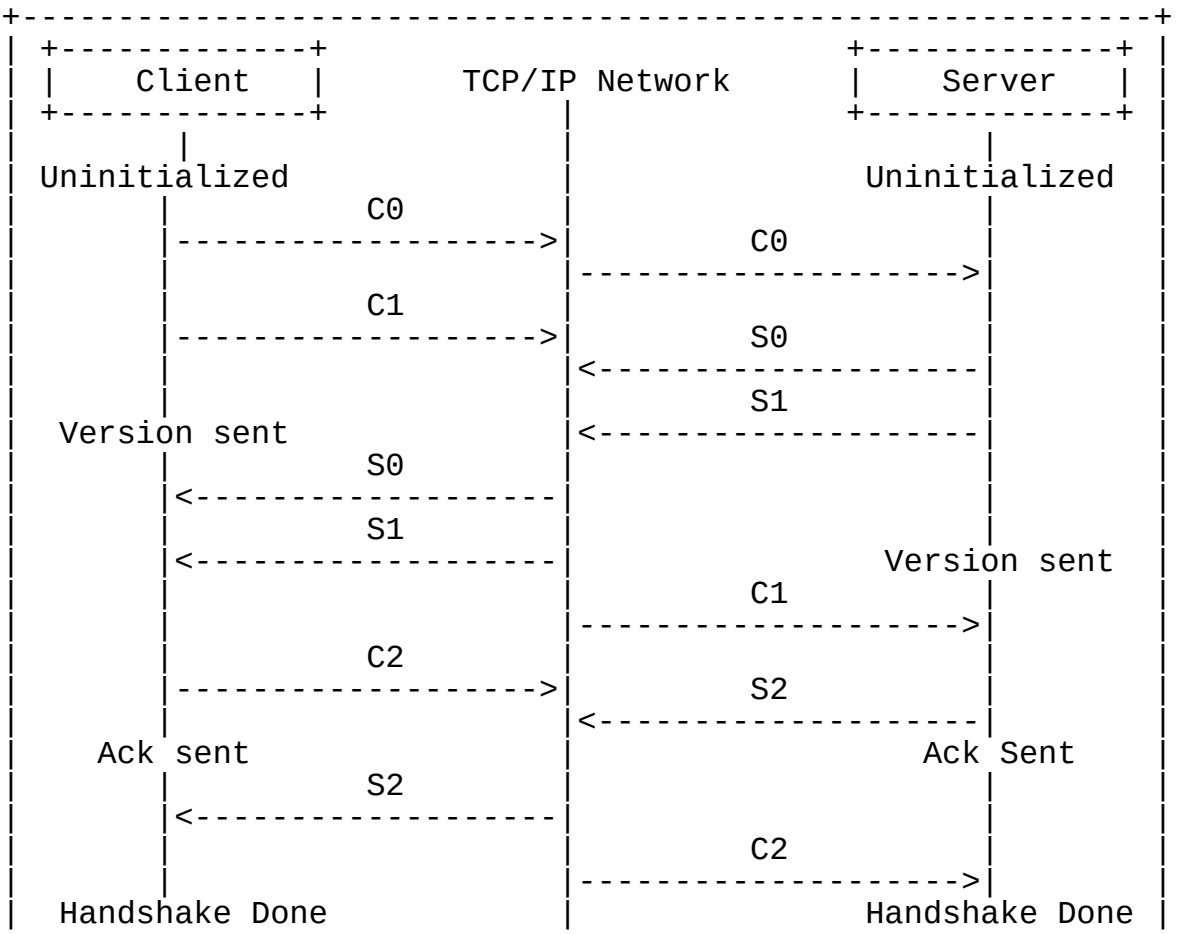
Figure 3 C2 and S2 bits

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



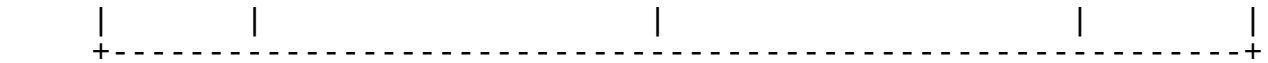


Figure 4 Pictorial Representation of Handshake

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:

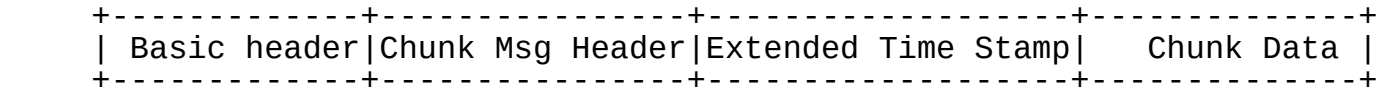


Figure 5 Chunk Format.

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.

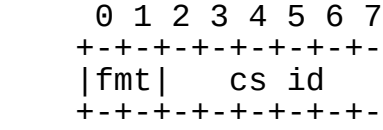


Figure 6 Chunk basic header 1

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

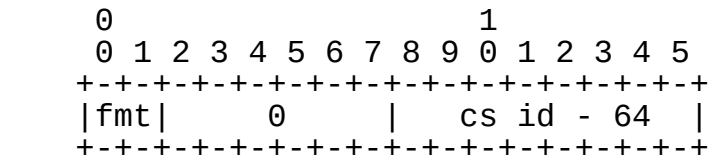


Figure 7 Chunk basic header 2

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).

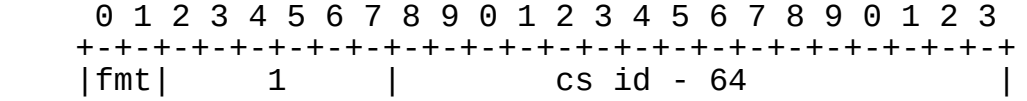


Figure 8 Chunk basic header 3

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).

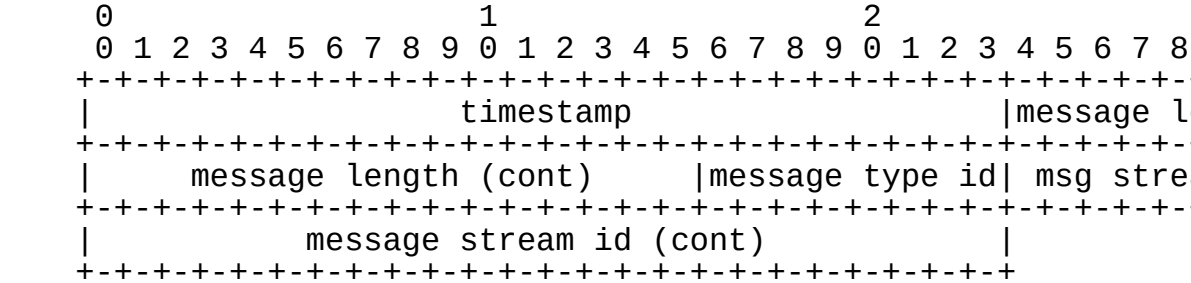


Figure 9 Chunk Message Header – Type 0

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.

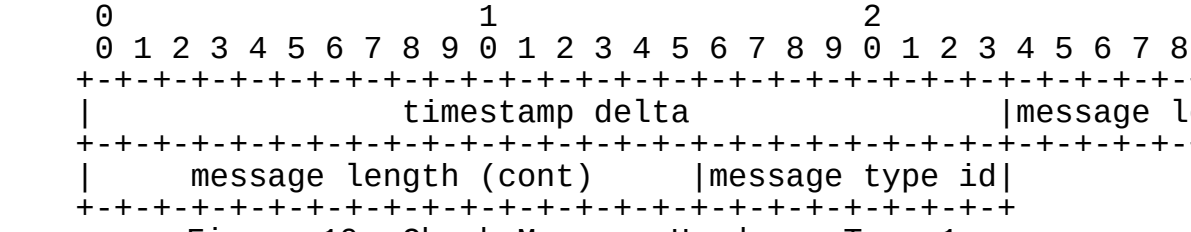


Figure 10 Chunk Message Header – Type 1

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.

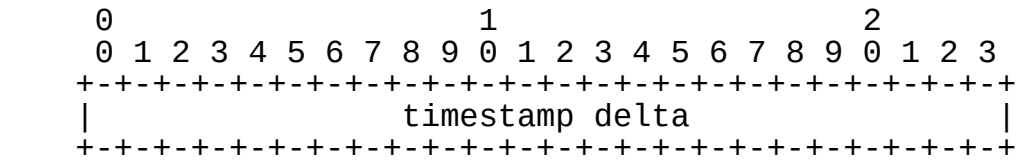


Figure 11 Chunk Message Header – Type 2

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.

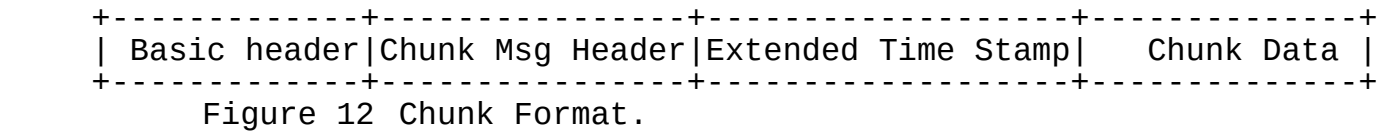


Figure 12 Chunk Format.

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

Real Time Messaging Protocol (RTMP) Message Formats
draft-rtmp-01.txt

Copyright Notice

Abstract

This memo describes the Real Time Messaging Protocol, 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 RTMP chunking stream protocol.

Table of Contents

1. Introduction

The document specifies the Real Time Messaging Protocol, which specifies the format of the messages that are transferred between entities on a network using a lower level transport layer.

While RTMP was designed to work with the Real Time Messaging Chunk Stream Protocol, it can send the messages using any other transport protocol. 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.

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

Message stream:
A logical channel of communication in which messages flow.

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

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

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 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 are given as an integer number of milliseconds, relative to an unspecified epoch. Typically, each message stream will start with a message having timestamp 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 streams (especially from separate hosts) requires some additional mechanism outside of RTMP.

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 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. RTMP Message Format

The server and the client send RTMP messages over the network to communicate with each other. The messages could include audio, video, data, or any other messages.

The RTMP message has two parts, a header and its payload.

4.1. Message Header

The message header contains the following:

Message Type:
One byte field to represent the message type. A range of type IDs (1-7) are reserved for protocol control messages.

Length:
Three-byte field that represents the size of the payload in bytes. It is set in big-endian format.

Timestamp:
Four-byte field that contains a timestamp of the message. The 4 bytes are packed in the big-endian order.

Message Stream Id:
Three-byte field that identifies the stream of the message. These bytes are set in big-endian format.

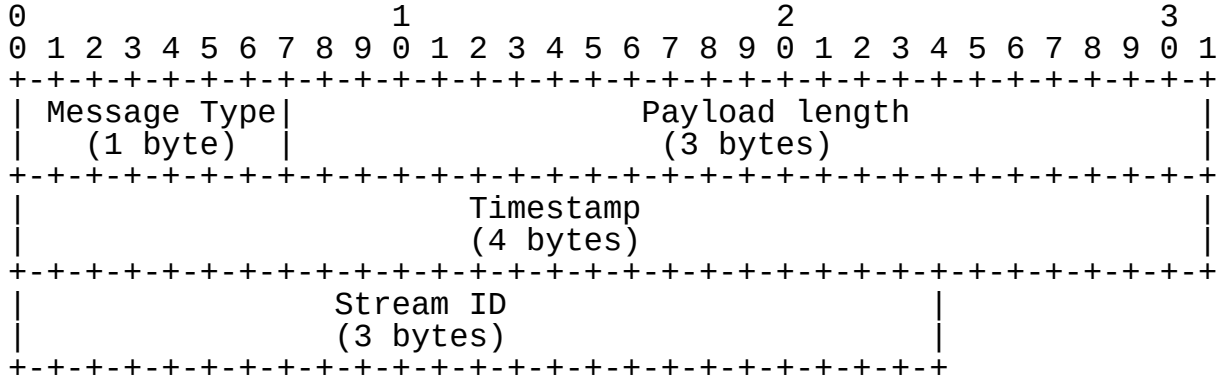


Figure 1 Message header

4.2. Message Payload

The other part which is the payload is the actual data that is contained in the message. For example, it could be some audio samples or compressed video data. The payload format and interpretation are beyond the scope of this document.

5. Protocol Control Messages

RTMP reserves message type IDs 1-7 for protocol control messages. These messages contain information needed by the RTM Chunk Stream protocol or RTMP itself. Protocol messages with IDs 1 & 2 are reserved for usage with RTM Chunk Stream protocol. Protocol messages

with IDs 3-6 are reserved for usage of RTMP. Protocol message with ID 7 is used between edge server and origin server.

Protocol control messages MUST 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.

5.1. Set Chunk Size (1)

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

The value of the chunk size is carried as 4-byte message payload. A default value exists for chunk size, but if the sender wants to change this value it notifies the peer about it through this protocol message. For example, a client wants to send 131 bytes of data and the chunk size is at its default value of 128. So every message from the client gets split into two chunks. The client can choose to change the chunk size to 131 so that every message get split into two chunks. The client MUST send this protocol message to the server to notify that the chunk size is set to 131 bytes.

The maximum chunk size can be 65536 bytes. Chunk size is maintained independently for server to client communication and client to server communication.

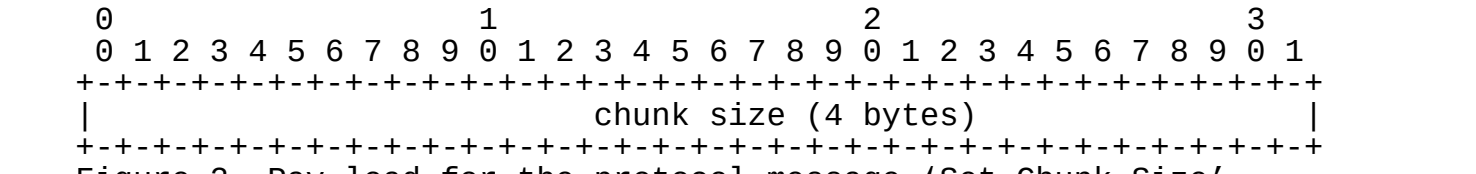


Figure 2 Pay load for the protocol message ‘Set Chunk Size’

chunk size: 32 bits
This field holds the new chunk size, which will be used for all future chunks sent by this chunk stream.

5.2. Abort Message (2)

Protocol control message 2, Abort 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 and abort processing of that message. The peer receives the chunk stream ID of the message to be discarded as payload of this protocol message. This message is sent when the sender has sent part of a message, but wants to tell the receiver that the rest of the message will not be sent.

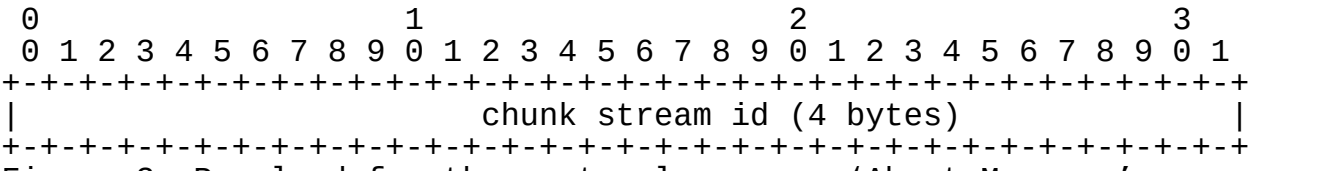


Figure 3 Pay load for the protocol message ‘Abort Message’.

chunk stream ID: 32 bits
This field holds the chunk stream ID, whose message is to be discarded.

5.3. Acknowledgement (3)

The client or the server sends the acknowledgment to the peer after receiving bytes equal to the window size. The window size is the maximum number of bytes that the sender sends without receiving acknowledgment from the receiver. The server sends the window size to the client after application connects. This message specifies the sequence number, which is the number of the bytes received so far.

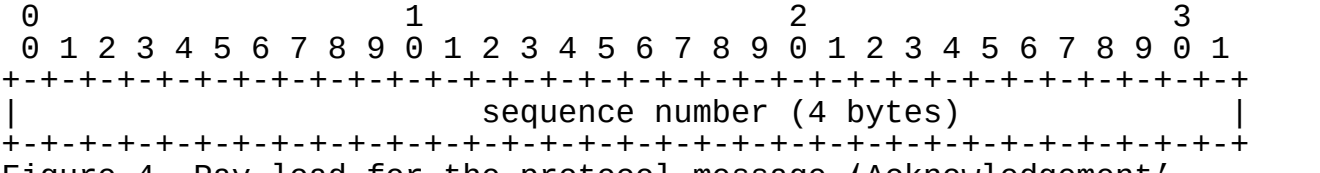


Figure 4 Pay load for the protocol message ‘Acknowledgement’.

sequence number: 32 bits
This field holds the number of bytes received so far.

5.4. User Control Message (4)

The client or the server sends this message to notify the peer about the user control events. This message carries Event type and Event data.

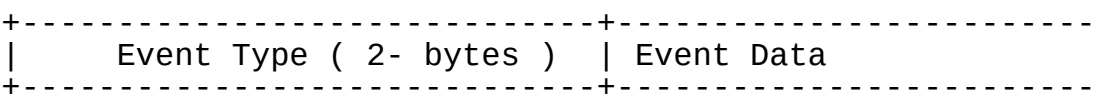


Figure 5 Pay load for the ‘User Control Message’.

The first 2 bytes of the message data are used to identify the Event type. Event type is followed by Event data. Size of Event data field is variable.

5.5. Window Acknowledgement Size (5)

The client or the server sends this message to inform the peer which window size to use when sending acknowledgment. For example, a server expects acknowledgment from the client every time the server sends bytes equivalent to the window size. The server updates the client about its window size after successful processing of a connect request from the client.

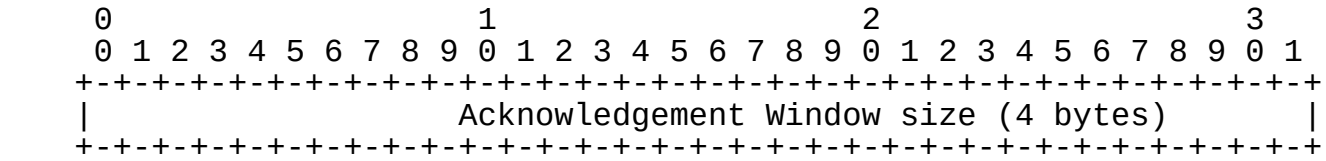


Figure 6 Pay load for ‘Window Acknowledgement Size’.

5.6. Set Peer Bandwidth (6)

The client or the server sends this message to update the output bandwidth of the peer. The output bandwidth value is the same as the window size for the peer. The peer sends ‘Window Acknowledgement Size’ back if its present window size is different from the one received in the message.

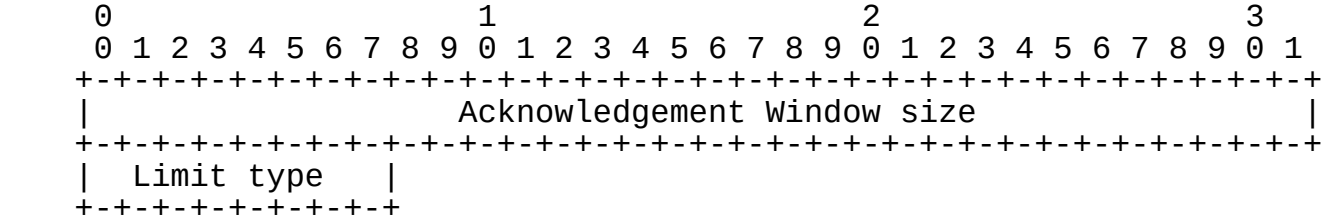


Figure 7 Pay load for ‘Set Peer Bandwidth’

The sender can mark this message hard (0), soft (1), or dynamic (2) using the Limit type field. In a hard (0) request, the peer must send the data in the provided bandwidth. In a soft (1) request, the bandwidth is at the discretion of the peer and the sender can limit the bandwidth. In a dynamic (2) request, the bandwidth can be hard or soft.

6. References

6.1. 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.

6.2. 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.

Address:

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

RTMP Commands Messages
draft-rtmpcommandmessages-01.txt

Abstract

This document describes the different types of messages and commands that are exchanged between the server and the client to communicate with each other.

Table of Contents

1. Introduction

The different types of messages that are exchanged between the server and the client include audio messages for sending the audio data, video messages for sending video data, data messages for sending any user data, shared object messages, and command messages. Shared objects messages provide a general purpose way to manage distributed data among multiple clients and a server.
Command messages carry the AMF encoded commands between the client and the server. A client or a server can request Remote Procedure Calls (RPC) over streams that are communicated using the command messages to the peer.

2. Definitions

Message stream:
A logical channel of communication in which messages flow.

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

Remote Procedure Calls (RPC)
A request that allows a client or a server to call a subroutine or procedure at the peer end.

Metadata
A description about the data. The metadata of a movie includes the movie title, duration, date of creation, and so on.

Application instance
The instance of the application at the server with which the clients connect by sending the connect request.

Action Message Format (AMF)
A compact binary format that is used to serialize ActionScript object graphs. Formats Specifications:
AMF0(http://opensource.adobe.com/wiki/download/attachments/1114283/amf0_spec_121207.pdf?version=1) and
AMF3(http://opensource.adobe.com/wiki/download/attachments/1114283/amf3_spec_05_05_08.pdf?version=1).

3. Types of messages

The server and the client send messages over the network to communicate with each other. The messages can be of any type which includes audio messages, video messages, command messages, shared object messages, data messages, and user control messages.

3.1. Command message

Command messages carry the AMF-encoded commands between the client and the server.These messages have been assigned message type value of 20 for AMF0 encoding and message type value of 17 for AMF3 encoding. These messages are sent to perform some operations like connect, createStream, publish, play, pause on the peer. Command messages like onstatus, result etc. are used to inform the sender about the status of the requested commands. A command message consists of command name, transaction ID, and command object that contains related parameters. A client or a server can request Remote Procedure Calls (RPC) over streams that are communicated using the command messages to the peer.

3.2. Data message

The client or the server sends this message to send Metadata or any user data to the peer. Metadata includes details about the data(audio, video etc.) like creation time, duration, theme and so on. These messages have been assigned message type value of 18 for AMF0 and message type value of 15 for AMF3.

3.3. Shared object message

A shared object is a Flash object (a collection of name value pairs) that are in synchronization across multiple clients, instances, and so on. The message types kMsgContainer=19 for AMF0 and kMsgContainerEx=16 for AMF3 are reserved for shared object events. Each message can contain multiple events.

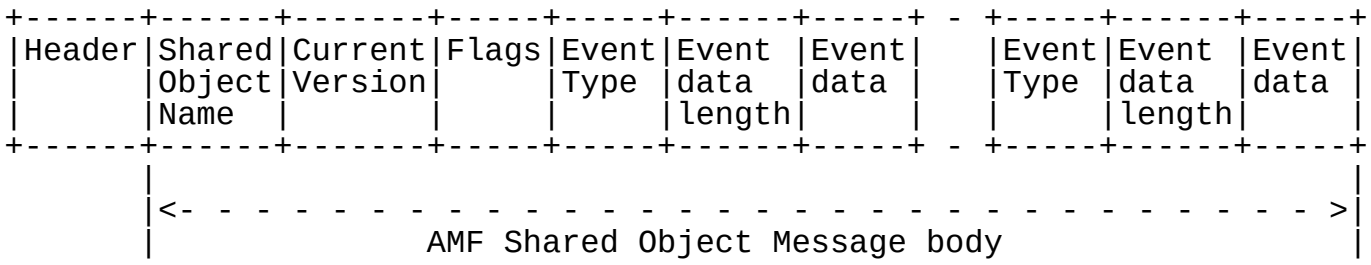


Figure 1 The shared object message format

The following event types are supported:

Event	Description
Use(=1)	The client sends this event to inform the server about the creation of a named shared object.
Release(=2)	The client sends this event to the server when the shared object is deleted on the client side.
Request Change (=3)	The client sends this event to request that the change the value associated with a named parameter of the shared object.
Change (=4)	The server sends this event to notify all clients, except the client originating the request, of a change in the value of a named parameter.
Success (=5)	The server sends this event to the requesting client in response to RequestChange event if the request is accepted.
SendMessage (=6)	The client sends this event to the server to broadcast a message. On receiving this event, the server broadcasts a message to all the clients, including the sender.
Status (=7)	The server sends this event to notify clients about error conditions.
Clear (=8)	The server sends this event to the client to clear a shared object. The server also sends this event in response to Use event that the client sends on connect.
Remove (=9)	The server sends this event to have the client delete a slot.
Request Remove (=10)	The client sends this event to have the client delete a slot.

Use Success (=11)	The server sends this event to the client on a successful connection.

3.4. Audio message

The client or the server sends this message to send audio data to the peer. The message type value of 8 is reserved for audio messages.

3.5. Video message

The client or the server sends this message to send video data to the peer. The message type value of 9 is reserved for video messages. These messages are large and can delay the sending of other type of messages. To avoid such a situation, the video message is assigned the lowest priority.

3.6. Aggregate message

An aggregate message is a single message that contains a list of sub-messages. The message type value of 22 is reserved for aggregate messages.

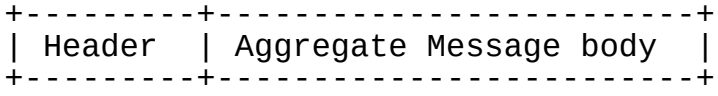


Figure 2 The aggregate message format

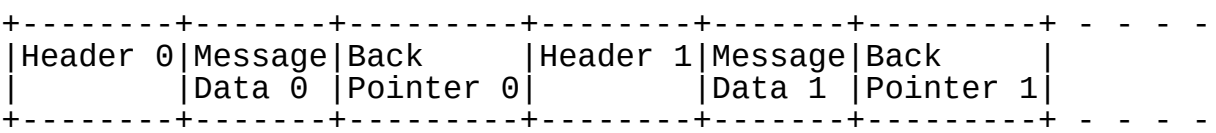


Figure 3 The aggregate message body format

The back pointer contains the size of the previous message including its header. It is included to match the format of FLV file and is used for backward seek.

Using aggregate messages has several performance benefits:

- The chunk stream can send at most a single complete message within a chunk. Therefore, increasing the chunk size and using the aggregate message reduces the number of chunks sent.

- The sub-messages can be stored contiguously in memory. It is more efficient when making system calls to send the data on the network.

3.7. User Control message

The client or the server sends this message to notify the peer about the user control events. For information about the message format, refer to the User Control Messages section in the RTMP Message Foramts draft.

The following user control event types are supported:

Event	Description
Stream Begin (=0)	The server sends this event to notify the client that a stream has become functional and can be used for communication. By default, this event is sent on ID 0 after the application connect command is successfully received from the client. The event data is 4-byte and represents the stream ID of the stream that became functional.
Stream EOF (=1)	The server sends this event to notify the client that the playback of data is over as requested on this stream. No more data is sent without issuing additional commands. The client discards the messages received for the stream. The 4 bytes of event data represent the ID of the stream on which playback has ended.
StreamDry (=2)	The server sends this event to notify the client that there is no more data on the stream. If the server does not detect any message for a time period, it can notify the subscribed clients that the stream is dry. The 4 bytes of event data represent the stream ID of the dry stream.
SetBuffer Length (=3)	The client sends this event to inform the server of the buffer size (in milliseconds) that is used to buffer any data coming over a stream. This event is sent before the server starts processing the stream. The first 4 bytes of the event data represent the stream ID and the next 4 bytes represent the buffer length, in milliseconds.
StreamIs Recorded (=4)	The server sends this event to notify the client that the stream is a recorded stream. The 4 bytes event data represent the stream ID of the recorded stream.
PingRequest (=6)	The server sends this event to test whether the client is reachable. Event data is a 4-byte timestamp, representing the local server time when the server dispatched the command. The client responds with kMsgPingResponse on receiving kMsgPingRequest.

PingResponse (=7)	The client sends this event to the server in response to the ping request. The event data is a 4-byte timestamp, which was received with the kMsgPingRequest request.

4. Types of commands

The client and the server exchange commands which are AMF encoded. The sender sends a command message that consists of command name, transaction ID, and command object that contains related parameters. For example, the connect command contains ‘app’ parameter, which tells the server application name the client is connected to. The receiver processes the command and sends back the response with the same transaction ID. The response string is either _result, _error, or a method name, for example, verifyClient or contactExternalServer.

A command string of _result or _error signals a response. The transaction ID indicates the outstanding command to which the response refers. It is identical to the tag in IMAP and many other protocols. The method name in the command string indicates that the sender is trying to run a method on the receiver end.

The following class objects are used to send various commands:

- NetConnection – An object that is a higher-level representation of connection between the server and the client.
- NetStream – An object that represents the channel over which audio streams, video streams and other related data are sent. We also send commands like play , pause etc. which control the flow of the data.

4.1. NetConnection commands

The NetConnection manages a two-way connection between a client application and the server. In addition, it provides support for asynchronous remote method calls.

The following commands can be sent on the NetConnection :

- connect

- call
- close
- createStream

4.1.1. connect

The client sends the connect command to the server to request connection to a server application instance.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command. Set to “connect”.
Transaction ID	Number	Always set to 1.
Command Object	Object	Command information object which has the name-value pairs.
Optional User Arguements	Object	Any optional information

Following is the description of the name-value pairs used in Command Object of the connect command.

Property	Type	Description	Example Value
app	String	The Server application name the client is connected to.	testapp
flashver	String	Flash Player version. It is the same string as returned by the ApplicationScript getversion () function.	FMSc/1.0
swfUrl	String	URL of the source SWF file making the connection.	file://C:/FlvPlayer.swf
tcUrl	String	URL of the Server. It has the following format. protocol://servername:port/appName/appInstance	rtmp://localhost:1935/testapp/instance1
fpad	Boolean	True if proxy is being used.	true or false
audioCodecs	Number	Indicates what audio codecs the client supports.	SUPPORT_SND_MP3
videoCodecs	Number	Indicates what video codecs are supported.	SUPPORT_VID_SORENSON
pageUrl	String	URL of the web page from where the SWF file was loaded.	http://somehost/sample.html
object Encoding	Number	AMF encoding method.	kAMF3

Values for the audio codecs property:

Source Code Constant	Usage	Value
SUPPORT_SND_NONE	Raw sound, no compression	0x0001
SUPPORT_SND_ADPCM	ADPCM compression	0x0002
SUPPORT_SND_MP3	mp3 compression	0x0004
SUPPORT_SND_INTEL	Not used	0x0008
SUPPORT_SND_UNUSED	Not used	0x0010
SUPPORT_SND_NELLY8	NellyMoser at 8-kHz compression	0x0020
SUPPORT_SND_NELLY	NellyMoser compression (5, 11, 22, and 44 kHz)	0x0040
SUPPORT_SND_G711A	G711A sound compression (Flash Media Server only)	0x0080
SUPPORT_SND_G711U	G711U sound compression (Flash Media Server only)	0x0100
SUPPORT_SND_NELLY16	NellyMouser at 16-kHz compression	0x0200
SUPPORT_SND_AAC	Advanced audio coding (AAC) codec	0x0400
SUPPORT_SND_SPEEX	Speex Audio	0x0800
SUPPORT_SND_ALL	All RTMP-supported audio codecs	0x0FFF

Values for the videoCodecs Property:

Source Code Constant	Usage	Value
SUPPORT_VID_UNUSED	Obsolete value	0x0001
SUPPORT_VID_JPEG	Obsolete value	0x0002
SUPPORT_VID_SORENSON	Sorenson Flash video	0x0004
SUPPORT_VID_HOMEBREW	V1 screen sharing	0x0008
SUPPORT_VID_VP6 (On2)	On2 video (Flash 8+)	0x0010
SUPPORT_VID_VP6ALPHA (On2 with alpha channel)	On2 video with alpha channel	0x0020
SUPPORT_VID_HOMEBREWV (screensharing v2)	Screen sharing version 2 (Flash 8+)	0x0040
SUPPORT_VID_H264	H264 video	0x0080
SUPPORT_VID_ALL	All RTMP-supported video codecs	0x00FF

Values for the video function property:

Source Code Constant	Usage	Value
SUPPORT_VID_CLIENT_SEEK	Indicates that the client can seek frame-accurate on the client	1

Values for the object encoding property:

Source Code Constant	Usage	Value
kAMF0	AMF0 object encoding supported by Flash 6 and later	0
kAMF3	AMF3 encoding from Flash 9 (AS3)	3

The command structure from server to client is as follows:

Field Name	Type	Description
Command Name	String	_result or _error; indicates whether the response is result or error.
Transaction ID	Number	Transaction ID is 1 for call connect responses
Properties	Object	Name-value pairs that describe the properties(fmsver etc.) of the connection.
Information	Object	Name-value pairs that describe the response from|the server. ‘code’, ‘level’, ‘description’ are names of few among such information.

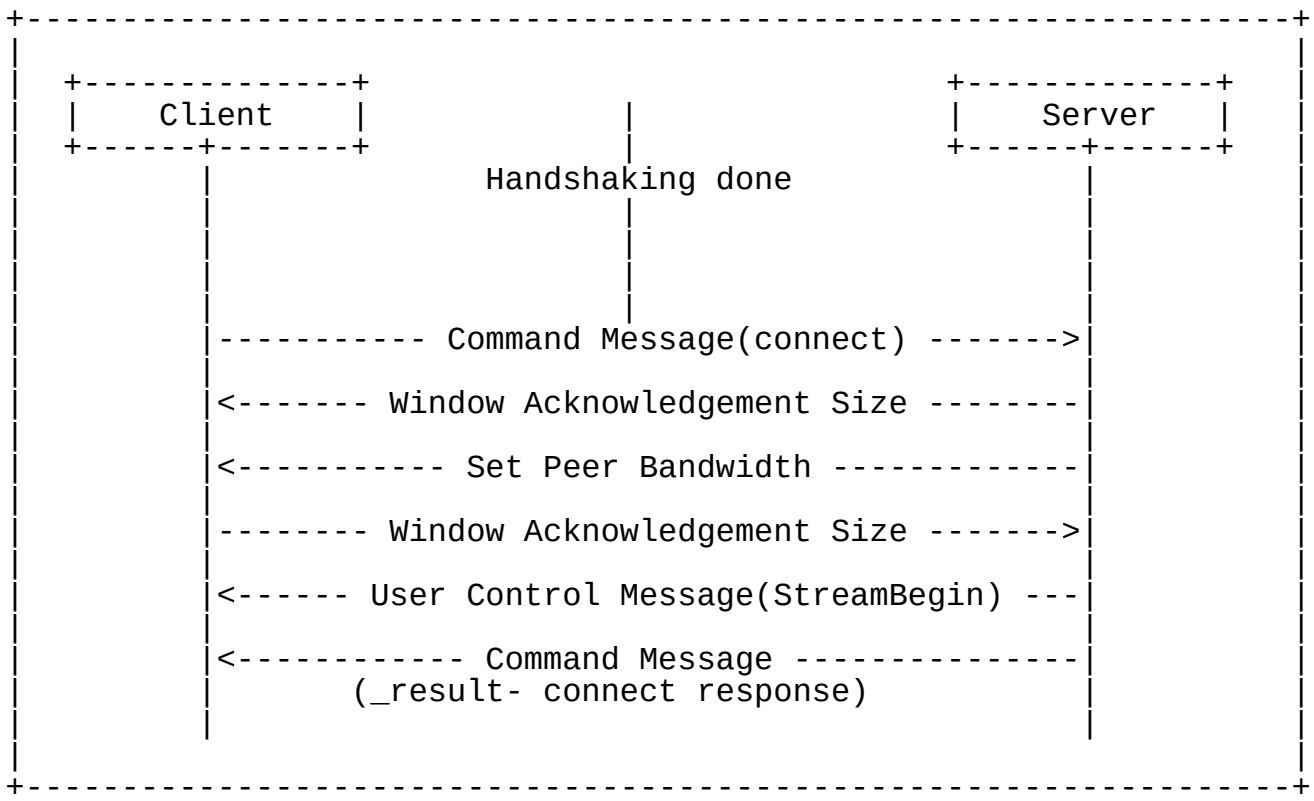


Figure 4 Message flow in the connect command

The message flow during the execution of the command is:

- Client sends the connect command to the server to request to connect with the server application instance.
- After receiving the connect command, the server sends the protocol message 'Window Acknowledgement Size' to the client. The server also connects to the application mentioned in the connect command.
- The server sends the protocol message 'Set Peer Bandwidth' to the client.
- The client sends the protocol message 'Window Acknowledgement Size' to the server after processing the protocol message 'Set Peer Bandwidth'.
- The server sends an another protocol message of type User Control Message(StreamBegin) to the client.

- The server sends the result command message informing the client of the connection status (success/fail). The command specifies the transaction ID (always equal to 1 for the connect command). The message also specifies the properties, such as Flash Media Server version (string), capabilities (number) In addition it specificies other connection response related information like level(string), code(string), description (string), objectencoding (number)etc.

4.1.2. Call

The call method of the NetConnection object runs remote procedure calls (RPC) at the receiving end. The called RPC name is passed as a parameter to the call command.

The command structure from the sender to the receiver is as follows:

Field Name	Type	Description
Procedure Name	String	Name of the remote procedure that is called.
Transaction ID	Number	If a response is expected we give a transaction Id. Else we pass a value of 0
Command Object	Object	If there exists any command info this is set, else this is set to null type.
Optional Arguements	Object	Any optional arguments to be provided

The command structure of the response is as follows:

Field Name	Type	Description
Command Name	String	Name of the command.
Transaction ID	Number	ID of the command, to which the response belongs to
Command Object	Object	If there exists any command info this is set, else this is set to null type.
Response	Object	Response from the method that was called.

4.1.3. createStream

The client sends this command to the server to create a logical channel for message communication The publishing of audio, video, and metadata is carried out over stream channel created using the createStream command.

NetConnection is the default communication channel, which has a stream ID 0. Protocol and a few command messages, including createStream, use the default communication channel.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command. Set to “createStream”.
Transaction ID	Number	Transaction ID of the command.
Command Object	Object	If there exists any command info this is set, else this is set to null type.

The command structure from server to client is as follows:

Field Name	Type	Description
Command Name	String	_result or _error; indicates whether the response is result or error.
Transaction ID	Number	ID of the command that response belongs to.
Command Object	Object	If there exists any command info this is set, else this is set to null type.
Stream ID	Number	The return value is either a stream ID or an error information object.

4.2. NetStream commands

The NetStream defines the channel through which the streaming audio, video, and data messages can flow over the NetConnection that connects the client to the server. A NetConnection object can support multiple NetStreams for multiple data streams.

The following commands can be sent on the NetStream :

- play
- play2
- deleteStream
- closeStream
- receiveAudio
- receiveVideo
- publish
- seek
- pause

4.2.1. play

The client sends this command to the server to play a stream. A playlist can also be created using this command multiple times.

If you want to create a dynamic playlist that switches among different live or recorded streams, call play more than once and pass false for reset each time. Conversely, if you want to play the specified stream immediately, clearing any other streams that are queued for play, pass true for reset.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command. Set to “play”.
Transaction ID	Number	Transaction ID set to 0.
Command Object	Null	Command information does not exist. Set to null type.
Stream Name	String	Name of the stream to play. To play video (FLV) files, specify the name of the stream without a file extension (for example, "sample"). To play back MP3 or ID3 tags, you must precede the stream name with mp3: (for example, "mp3:sample". To play H.264/AAC files, you must precede the stream name with mp4: and specify the file extension. For example, to play the file sample.m4v,specify "mp4:sample.m4v"
Start	Number	An optional parameter that specifies the start time in seconds. The default value is -2, which means the subscriber first tries to play the live stream specified in the Stream Name field. If a live stream of that name is not found,it plays the recorded stream specified in the Stream Name field. If you pass -1 in the Start field, only the live stream specified in the Stream Name field is played. If you pass 0 or a positive number in the Start field, a recorded stream specified in the Stream Name field is played beginning from the time specified in the Start field. If no recorded stream is found, the next item in the playlist is played.
Duration	Number	An optional parameter that specifies the duration of playback in seconds. The default value is -1. The -1 value means

		a live stream is played until it is no longer available or a recorded stream is played until it ends. If u pass 0, it plays the single frame since the time specified in the Start field from the beginning of a recorded stream. It is assumed that the value specified in the Start field is equal to or greater than 0. If you pass a positive number, it plays a live stream for the time period specified in the Duration field. After that it becomes available or plays a recorded stream for the time specified in the Duration field. (If a stream ends before the time specified in the Duration field, playback ends when the stream ends.) If you pass a negative number other than -1 in the Duration field, it interprets the value as if it were -1.
Reset	Boolean	An optional Boolean value or number that specifies whether to flush any previous playlist.

The command structure from the server to the client is as follows:

Field Name	Type	Description
Command Name	String	Name of the command. If the play command is successful, the command name is set to onStatus.
Description	String	If the play command is successful, the client receives OnStatus message from server which is NetStream.Play.Start. If the specified stream is not found , NetStream.Play.StreamNotFound is received.

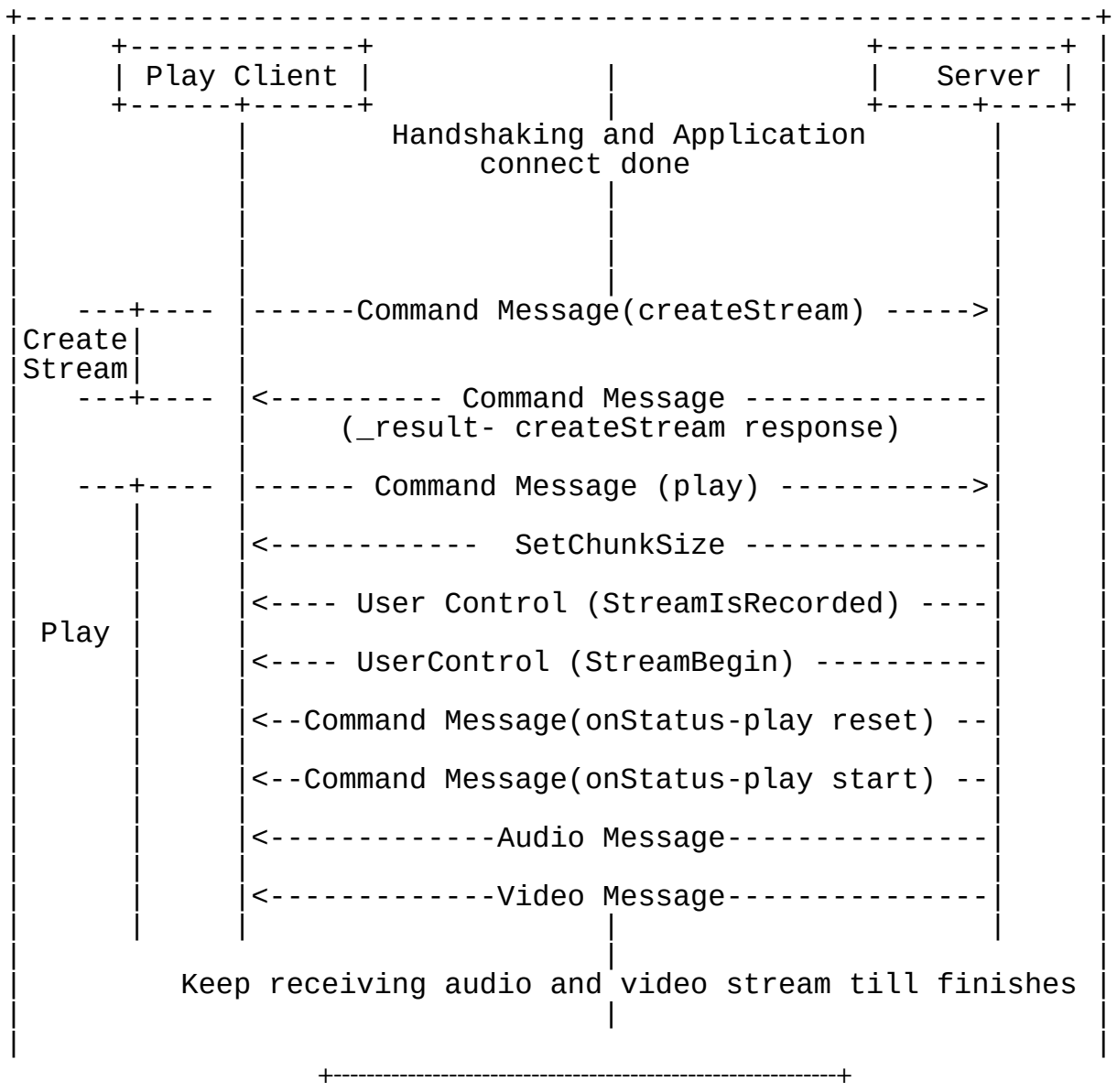


Figure 5 Message flow in the play command

The message flow during the execution of the command is:

- The client sends the play command after receiving result of the createStream command as success from the server.
- On receiving the play command, the server sends a protocol message to set the chunk size.

- The server sends another protocol message (user control) specifying the event ‘StreamIsRecorded’ and the stream ID in that message. The message carries the event type in the first 2 bytes and the stream ID in the last 4 bytes.

- The server sends another protocol message (user control) specifying the event ‘StreamBegin’, to indicate beginning of the streaming to the client.

- The server sends OnStatus command messages NetStream.Play.Start & NetStream.Play.Reset if the play command sent by the client is successful. NetStream.Play.Reset is sent by the server only if the play command sent by the client has set the reset flag. If the stream to be played is not found, the Server sends the onStatus message NetStream.Play.StreamNotFound.

After this, the server sends audio and video data, which the client plays.

4.2.2. play2

Unlike the play command, play2 can switch to a different bit rate stream without changing the timeline of the content played. The server maintains multiple files for all supported bitrates that the client can request in play2.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command, set to “play2”.
Transaction ID	Number	Transaction ID set to 0.
Start Time	Object	A AMF encoded object that stores a number value. The value in this field specifies the beginning position of the stream, in seconds. If 0 is passed in the Start Time field, the stream is played from the current timeline.
oldStreamName	Object	A AMF encoded object that stores a string value. Its value is a string containing the stream name parameter and the old stream name.
Stream Name	Object	A AMF encoded object that stores a string value. It stores the name of the stream that is played.
Duration	Object	A AMF encoded object that stores a number value. The value stored in it specifies the total duration of playing the stream.
Transition	Object	A AMF encoded object that stores a string value. Its value defines the playlist transition mode (switch or. swap mode)switch:Performs multi-bitrate streaming by switching 1-bit rate version of a stream to another. swap: Replaces the value in oldStreamName with the value in streamName, and stores the remaining playlist queue as is. However, in this case, the server does not make any assumptions about the content of the streams and treats them like different content. Hence, it either switches at the stream boundary or never.

The message flow for the command is shown in the following illustration.

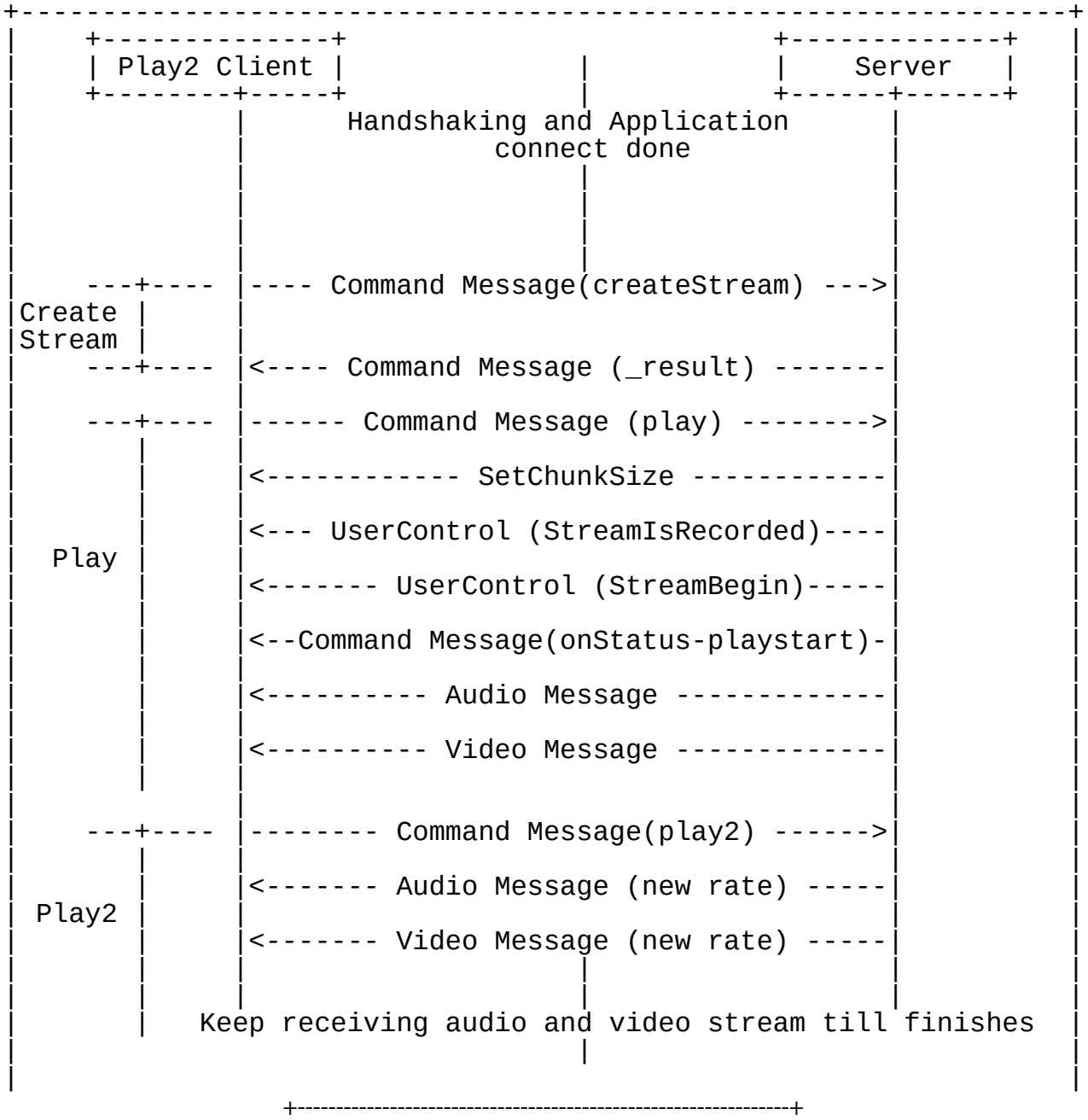


Figure 1 Message flow in the play2 command

4.2.3. deleteStream

NetStream sends the deleteStream command when the NetStream object is getting destroyed.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command, set to “deleteStream”.
Transaction ID	Number	Transaction ID set to 0.
Command Object	Null	Command information object does not exist. Set to null type.
Stream ID	Number	The ID of the stream that is destroyed on the server.

The server does not send any response.

4.2.4. receiveAudio

NetStream sends the receiveAudio message to inform the server whether to send or not to send the audio to the client.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command, set to “receiveAudio”.
Transaction ID	Number	Transaction ID set to 0.
Command Object	Null	Command information object does not exist. Set to null type.
Bool Flag	Boolean	true or false to indicate whether to receive audio or not.

The server does not send any response.

4.2.5. receiveVideo

NetStream sends the receiveVideo message to inform the server whether to send the video to the client or not.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command, set to “receiveVideo”.
Transaction ID	Number	Transaction ID set to 0.
Command Object	Null	Command information object does not exist. Set to null type.
Bool Flag	Boolean	true or false to indicate whether to receive video or not.

The server does not send any response.

4.2.6. Publish

The client sends the publish command to publish a named stream to the server. Using this name, any client can play this stream and receive the published audio, video, and data messages.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command, set to “publish”.
Transaction ID	Number	Transaction ID set to 0.
Command Object	Null	Command information object does not exist. Set to null type.
Publishing Name	String	Name with which the stream is published.
Publishing Type	String	Type of publishing. Set to “live”, “record”, or “append”. record: The stream is published and the data is recorded to a new file.The file is stored on the server in a subdirectory within the directory that contains the server application. If the file already exists, it is overwritten. append: The stream is published and the data is appended to a file. If no file is found, it is created. live: Live data is published without recording it in a file.

The server responds with the OnStatus command to mark the beginning of publish.

4.2.7. seek

The client sends the seek command to seek the offset (in milliseconds) within a media file or playlist.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command, set to “seek”.
Transaction ID	Number	Transaction ID set to 0.
Command Object	Null	There is no command information object for this command. Set to null type.
milliSeconds	Number	Number of milliseconds to seek into the playlist.

The server sends a status message NetStream.Seek.Notify when seek is successful. In failure, it returns an _error message.

4.2.8. pause

The client sends the pause command to tell the server to pause or start playing.

The command structure from the client to the server is as follows:

Field Name	Type	Description
Command Name	String	Name of the command, set to “pause”.
Transaction ID	Number	There is no transaction ID for this command. Set to 0.
Command Object	Null	Command information object does not exist. Set to null type.
Pause/Unpause Flag	Boolean	true or false, to indicate pausing or resuming play
milliSeconds	Number	Number of milliseconds at which the the stream is paused or play resumed. This is the current stream time at the Client when stream was paused. When the playback is resumed, the server will only send messages with timestamps greater than this value.

The server sends a status message NetStream.Pause.Notify when the stream is paused. NetStream.Unpause.Notify is sent when a stream in un-paused. In failure, it returns an _error message.

5. Message exchange example

Here are a few examples to explain message exchange using RTMP.

5.1. Publish recorded video

The example illustrates how a publisher can publish a stream and then stream the video to the server. Other clients can subscribe to this published stream and play the video.

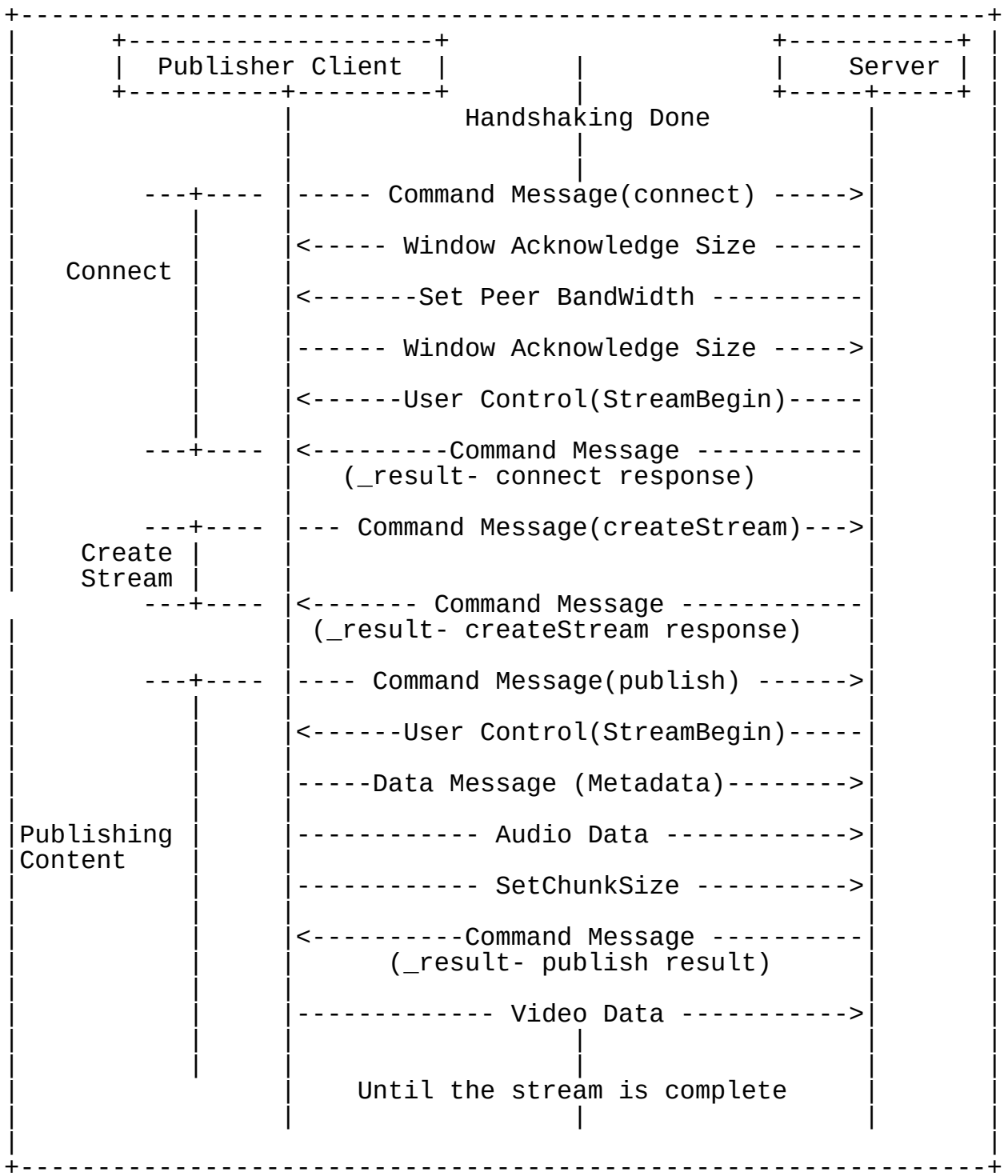


Figure 1 Message flow in publishing a video stream

5.2. Broadcasting a shared object message

The example illustrates the messages that are exchanged during the creation and changing of shared object. It also illustrates the process of shared object message broadcasting.

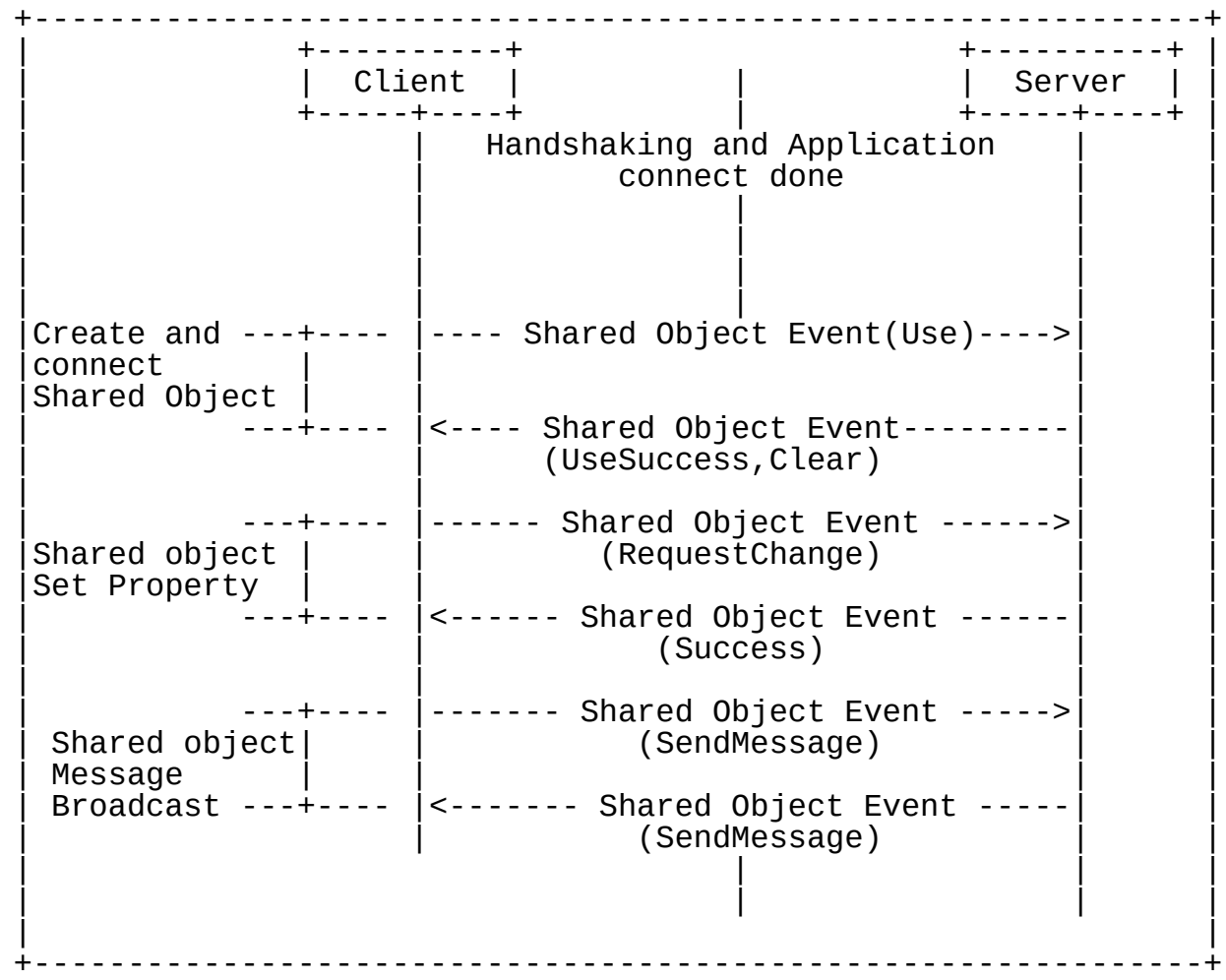


Figure 1 Shared object message broadcast

5.3. Publish MetaData from recorded stream

This example describes the message exchange for publishing metadata.

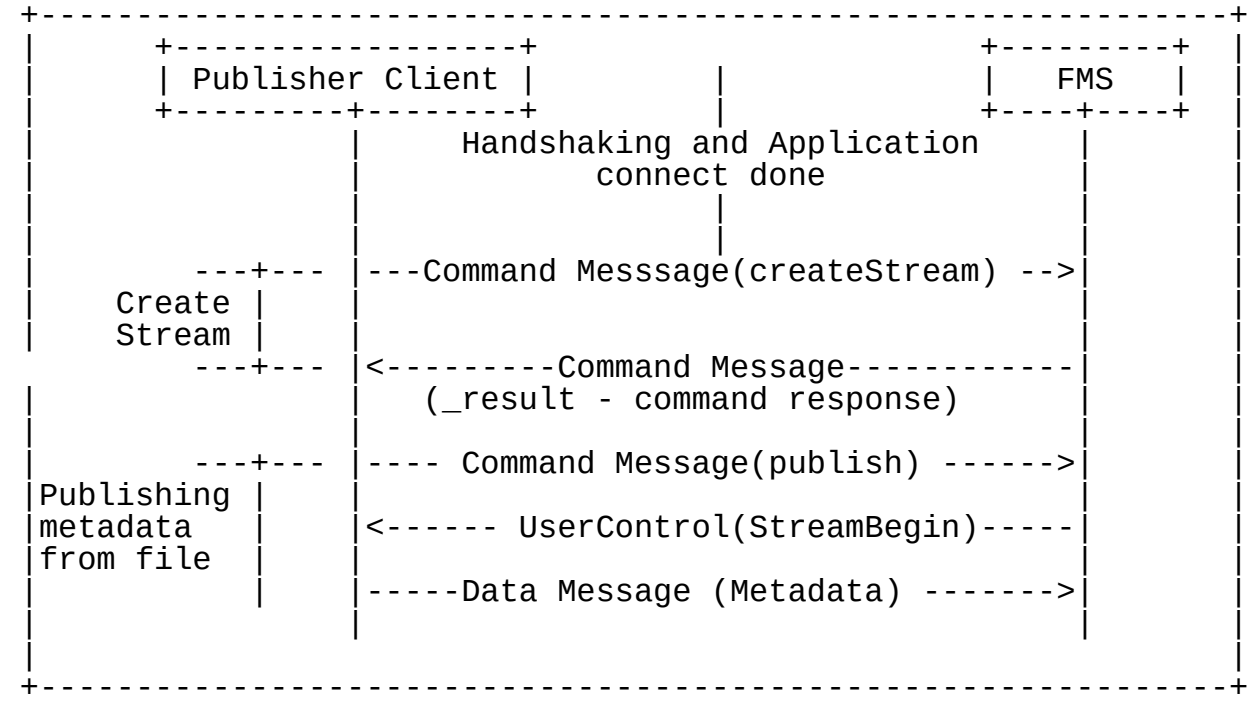


Figure 2 Publishing metadata

6. References

6.1. 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.

6.2. 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.

7. Acknowledgments

Address:

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