Copyright Adobe Systems Incorporated

H. Parmar, Ed.
M. Thornburgh, Ed.
Adobe
December 21, 2012

Adobe's Real Time Messaging Protocol

Abstract

This memo describes Adobe's Real Time Messaging Protocol (RTMP), 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 3
  1.1. Terminology 3
2. Contributors 3
3. Definitions 3
4. Byte Order, Alignment, and Time Format 5
5. RTMP Chunk Stream 5
  5.1. Message Format 6
  5.2. Handshake 7
    5.2.1. Handshake Sequence 7
    5.2.2. C0 and S0 Format 7
    5.2.3. C1 and S1 Format 8
    5.2.4. C2 and S2 Format 8
    5.2.5. Handshake Diagram 10
  5.3. Chunking 11
    5.3.1. Chunk Format 11
      5.3.1.1. Chunk Basic Header 12
      5.3.1.2. Chunk Message Header 13
        5.3.1.2.1. Type 0 14
        5.3.1.2.2. Type 1 14
        5.3.1.2.3. Type 2 15
        5.3.1.2.4. Type 3 15
        5.3.1.2.5. Common Header Fields 15
      5.3.1.3. Extended Timestamp 16
    5.3.2. Examples 16
      5.3.2.1. Example 1 16
      5.3.2.2. Example 2 17
  5.4. Protocol Control Messages 18
    5.4.1. Set Chunk Size (1) 19
    5.4.2. Abort Message (2) 19
    5.4.3. Acknowledgement (3) 20

IPR2024-01305
SportsCastr Inc. EX-2024

5.4.4. Window Acknowledgement Size (5) 20
5.4.5. Set Peer Bandwidth (6) 21
6. RTMP Message Formats 21
6.1. RTMP Message Format 22
6.1.1. Message Header 22
6.1.2. Message Payload 22
6.2. User Control Messages (4) 23
7. RTMP Command Messages 23
7.1. Types of Messages 24
7.1.1. Command Message (20, 17) 24
7.1.2. Data Message (18, 15) 24
7.1.3. Shared Object Message (19, 16) 24
7.1.4. Audio Message (8) 26
7.1.5. Video Message (9) 26
7.1.6. Aggregate Message (22) 26
7.1.7. User Control Message Events 27
7.2. Types of Commands 28
7.2.1. NetConnection Commands 29
7.2.1.1. connect 29
7.2.1.2. Call 35
7.2.1.3. createStream 36
7.2.2. NetStream Commands 37
7.2.2.1. play 38
7.2.2.2. play2 42
7.2.2.3. deleteStream 43
7.2.2.4. receiveAudio 44
7.2.2.5. receiveVideo 45
7.2.2.6. publish 45
7.2.2.7. seek 46
7.2.2.8. pause 47
7.3. Message Exchange Examples 48
7.3.1. Publish Recorded Video 48
7.3.2. Broadcast a Shared Object Message 50
7.3.3. Publish Metadata from Recorded Stream 50
8. References 51
Authors’ Addresses 52

1. Introduction

Adobe’s Real Time Messaging Protocol (RTMP) provides a bidirectional message multiplex service over a reliable stream transport, such as TCP [RFC0793], intended to carry parallel streams of video, audio, and data messages, with associated timing information, between a pair of communicating peers. Implementations typically assign different priorities to different classes of messages, which can effect the order in which messages are enqueued to the underlying stream transport when transport capacity is constrained.

This memo describes the syntax and operation of the Real Time Messaging Protocol.

1.1. Terminology

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

2. Contributors

Rajesh Mallipeddi, formerly of Adobe Systems, was the original editor of this specification, and provided most of its original text.

Mohit Srivastava of Adobe Systems contributed to the development of this specification.

3. 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 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.

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.

Remote Procedure Call (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. AMF has two versions: AMF 0 [AMF0] and AMF 3 [AMF3].

4. 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 Internet Protocol [RFC0791]. 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 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 streams (especially from separate hosts) requires some additional mechanism outside of RTMP.

Because timestamps are 32 bits long, they roll over every 49 days, 17 hours, 2 minutes and 47.296 seconds. Because streams are allowed to run continuously, potentially for years on end, an RTMP application SHOULD use serial number arithmetic [RFC1982] when processing timestamps, and SHOULD be capable of handling wraparound. For example, an application assumes that all adjacent timestamps are within 2^31 - 1 milliseconds of each other, so 10000 comes after 4000000000, and 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.

5. RTMP Chunk Stream

This section 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 (Section 6), 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 [RFC0793], 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.

5.1. Message Format

The format of a message that can be split into chunks to support multiplexing depends on a 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.

5.2. 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.

5.2.1. 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.

5.2.2. C0 and S0 Format

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

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|   version     |
+-+-+-+-+-+-+-+-+
```

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.

5.2.3. C1 and S1 Format

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        time (4 bytes)                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        zero (4 bytes)                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        random bytes                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         random bytes                          |
|                            (cont)                             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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.

5.2.4. 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:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        time (4 bytes)                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       time2 (4 bytes)                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        random echo                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         random echo                           |
|                            (cont)                             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 or 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.

5.2.5. Handshake Diagram

```
        +-------------+                            +-------------+
        |    Client   |       TCP/IP Network       |    Server   |
        +-------------+            |               +-------------+
              |                    |                      |
        Uninitialized              |                Uninitialized
              |          C0        |                      |
              |------------------->|         C0           |
              |                    |--------------------->|
              |          C1        |                      |
              |------------------->|         S0           |
              |                    |<---------------------|
              |                    |         S1           |
         Version sent              |<---------------------|
              |          S0        |                      |
              |<-------------------|                      |
              |          S1        |                      |
              |<-------------------|                 Version sent
              |                    |         C1           |
              |                    |--------------------->|
              |          C2        |                      |
              |------------------->|         S2           |
              |                    |<---------------------|
            Ack sent               |                  Ack Sent
              |          S2        |                      |
              |<-------------------|                      |
              |                    |         C2           |
              |                    |--------------------->|
         Handshake Done            |                Handshake Done
              |                    |                      |
```

Pictorial Representation of Handshake

The following describes the states mentioned in the handshake diagram:

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.

Handshake Done: The client and the server exchange messages.

5.3. 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 into smaller messages, for example to prevent large low-priority messages (such as video) from blocking smaller high-priority messages (such as audio or control).

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 Set Chunk Size control message (Section 5.4.1). Larger chunk sizes 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.

5.3.1. Chunk Format

Each chunk consists of a header and data. The header itself has three parts:

```
+--------------+----------------+--------------------+--------------+
| Basic Header | Message Header | Extended Timestamp |  Chunk Data  |
+--------------+----------------+--------------------+--------------+
|                                                    |
|<------------------- Chunk Header ------------------>|
```

Chunk Format

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.

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 is present in certain circumstances depending on the encoded timestamp or timestamp delta field in the Chunk Message header. See Section 5.3.1.3 for more information.

Chunk Data (variable size): The payload of this chunk, up to the configured maximum chunk size.

5.3.1.1. 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 2 byte form and an ID in the range of 64-319 (the second byte + 64). Value 1 indicates the 3 byte form and an ID in the range of 64-65599 ((the third byte)*256 + the second byte + 64). Values in the range of 3-63 represent the complete stream ID. Chunk Stream ID with value 2 is reserved for low-level protocol control messages and commands.

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.

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|fmt|   cs id   |
+-+-+-+-+-+-+-+-+
```

Chunk basic header 1

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

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|fmt|     0     |   cs id - 64  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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).

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|fmt|     1     |        cs id - 64             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 either the 2-byte or 3-byte form of the header.

5.3.1.2. 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.

5.3.1.2.1. Type 0

Type 0 chunk headers 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).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   timestamp                   |message length |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     message length (cont)     |message type id| msg stream id |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           message stream id (cont)            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 0xFFFFFF), this field MUST be 16777215, indicating the presence of the Extended Timestamp field to encode the full 32 bit timestamp. Otherwise, this field SHOULD be the entire timestamp.

5.3.1.2.2. Type 1

Type 1 chunk headers 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.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                timestamp delta                |message length |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     message length (cont)     |message type id|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Chunk Message Header - Type 1

5.3.1.2.3. Type 2

Type 2 chunk headers 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.

```
 0                   1                   2
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               timestamp delta                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Chunk Message Header - Type 2

5.3.1.2.4. Type 3

Type 3 chunks have no message header. The stream ID, message length and timestamp delta fields are not present; chunks of this type take values from the preceding chunk for the same Chunk Stream ID. 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 (Section 5.3.2.2). A stream consisting of messages of exactly the same size, stream ID and spacing in time SHOULD use this type for all chunks after a chunk of Type 2. Refer to Example 1 (Section 5.3.2.1). If the delta between the first message and the second message is same as the timestamp of the first message, then a chunk of Type 3 could immediately follow the chunk of Type 0 as there is no need for a chunk of Type 2 to register the delta. If a Type 3 chunk follows a Type 0 chunk, then the timestamp delta for this Type 3 chunk is the same as the timestamp of the Type 0 chunk.

5.3.1.2.5. Common Header Fields

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 0xFFFFFF), this field MUST be 16777215, indicating the presence of the Extended Timestamp field to encode the full 32 bit delta. Otherwise, this field SHOULD be the actual 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 the benefits 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.

5.3.1.3. Extended Timestamp

The Extended Timestamp field is used to encode timestamps or timestamp deltas that are greater than 16777215 (0xFFFFFF); that is, for timestamps or timestamp deltas that don’t fit in the 24 bit fields of Type 0, 1, or 2 chunks. This field encodes the complete 32-bit timestamp or timestamp delta. The presence of this field is indicated by setting the timestamp field of a Type 0 chunk, or the timestamp delta field of a Type 1 or 2 chunk, to 16777215 (0xFFFFFF). This field is present in Type 3 chunks when the most recent Type 0, 1, or 2 chunk for the same chunk stream ID indicated the presence of an extended timestamp field.

5.3.2. Examples

5.3.2.1. Example 1

This example 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

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

Format of each of the chunks of audio messages

5.3.2.2. Example 2

This example illustrates a message that is too long to fit in a 128-byte chunk and is broken into several chunks.

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

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

Format of each of the chunks

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.

5.4. Protocol Control Messages

RTMP Chunk Stream uses message type IDs 1, 2, 3, 5, and 6 for protocol control messages. These messages contain information needed by the RTMP Chunk Stream protocol.

These protocol control messages MUST have message stream ID 0 (known as the control stream) and be sent in chunk stream ID 2. Protocol control messages take effect as soon as they are received; their

timestamps are ignored.

5.4.1. Set Chunk Size (1)

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

The maximum chunk size defaults to 128 bytes, but the client or the server can change this value, and updates its peer using this message. 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 message to the server to notify it that the chunk size is now 131 bytes. The client can then send the audio data in a single chunk.

The maximum chunk size SHOULD be at least 128 bytes, and MUST be at least 1 byte. The maximum chunk size is maintained independently for each direction.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|                     chunk size (31 bits)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Payload for the ‘Set Chunk Size’ protocol message

0: This bit MUST be zero.

chunk size (31 bits): This field holds the new maximum chunk size, in bytes, which will be used for all of the sender’s subsequent chunks until further notice. Valid sizes are 1 to 2147483647 (0x7FFFFFFF) inclusive; however, all sizes greater than 16777215 (0xFFFFFF) are equivalent since no chunk is larger than one message, and no message is larger than 16777215 bytes.

5.4.2. 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. 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.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      chunk stream id (32 bits)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Payload for the ‘Abort Message’ protocol message

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

5.4.3. Acknowledgement (3)

The client or the server MUST send an 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. This message specifies the sequence number, which is the number of the bytes received so far.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       sequence number (4 bytes)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Payload for the ‘Acknowledgement’ protocol message

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

5.4.4. Window Acknowledgement Size (5)

The client or the server sends this message to inform the peer of the window size to use between sending acknowledgments. The sender expects acknowledgment from its peer after the sender sends window size bytes. The receiving peer MUST send an Acknowledgement (Section 5.4.3) after receiving the indicated number of bytes since the last Acknowledgement was sent, or from the beginning of the session if no Acknowledgement has yet been sent.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Acknowledgement Window size (4 bytes)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Payload for the ‘Window Acknowledgement Size’ protocol message

5.4.5. Set Peer Bandwidth (6)

The client or the server sends this message to limit the output bandwidth of its peer. The peer receiving this message limits its output bandwidth by limiting the amount of sent but unacknowledged data to the window size indicated in this message. The peer receiving this message SHOULD respond with a Window Acknowledgement Size message if the window size is different from the last one sent to the sender of this message.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Acknowledgement Window size                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Limit Type   |
+-+-+-+-+-+-+-+-+
```

Payload for the ‘Set Peer Bandwidth’ protocol message

The Limit Type is one of the following values:

0 - Hard: The peer SHOULD limit its output bandwidth to the indicated window size.

1 - Soft: The peer SHOULD limit its output bandwidth to the the window indicated in this message or the limit already in effect, whichever is smaller.

2 - Dynamic: If the previous Limit Type was Hard, treat this message as though it was marked Hard, otherwise ignore this message.

6. RTMP Message Formats

The section specifies the format of RTMP messages that are transferred between entities on a network using a lower level transport layer, such as RTMP Chunk Stream.

While RTMP was designed to work with the RTMP Chunk Stream, 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.

6.1. 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.

6.1.1. Message Header

The message header contains the following:

Message Type: One byte field to represent the message type. A range of type IDs (1-6) 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.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Message Type  |                Payload length                 |
|   (1 byte)    |                 (3 bytes)                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Timestamp                               |
|                       (4 bytes)                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Stream ID                      |
|                (3 bytes)                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Message Header

6.1.2. Message Payload

The other part of the message is the payload, which is the actual data 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.

6.2. User Control Messages (4)

RTMP uses message type ID 4 for User Control messages. These messages contain information used by the RTMP streaming layer. Protocol messages with IDs 1, 2, 3, 5, and 6 are used by the RTMP Chunk Stream protocol (Section 5.4).

User Control messages SHOULD use message stream ID 0 (known as the control stream) and, when sent over RTMP Chunk Stream, be sent on chunk stream ID 2. User Control messages are effective at the point they are received in the stream; their timestamps are ignored.

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.

```
+-------------------------------+-------------------------
|     Event Type (16 bits)      | Event Data
+-------------------------------+-------------------------
```

Payload for the ‘User Control’ protocol message

The first 2 bytes of the message data are used to identify the Event type. Event type is followed by Event data. The size of Event Data field is variable. However, in cases where the message has to pass through the RTMP Chunk Stream layer, the maximum chunk size (Section 5.4.1) SHOULD be large enough to allow these messages to fit in a single chunk.

Event Types are and their Event Data formats are enumerated in Section 7.1.7.

7. RTMP Command Messages

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

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 object 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.

7.1. 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.

7.1.1. Command Message (20, 17)

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.

7.1.2. Data Message (18, 15)

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.

7.1.3. Shared Object Message (19, 16)

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 19 for AMF0 and 16 for AMF3 are reserved for shared object events. Each message can contain multiple events.

```
+------+------+-------+-----+-----+------+-----+ +-----+------+-----+
|Header|Shared|Current|Flags|Event|Event |Event|.|Event|Event |Event|
|      |Object|Version|     |Type |data  |data |.|Type |data  |data |
|      |Name  |       |     |     |length|     |.|     |length|     |
+------+------+-------+-----+-----+------+-----+ +-----+------+-----+
       |                                                            |
       |<- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - >|
       |             AMF Shared Object Message body                 |
```

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.

7.1.4. Audio Message (8)

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.

7.1.5. Video Message (9)

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.

7.1.6. Aggregate Message (22)

An aggregate message is a single message that contains a series of RTMP sub-messages using the format described in Section 6.1. Message type 22 is used for aggregate messages.

```
+---------+-------------------------+
| Header  | Aggregate Message body  |
+---------+-------------------------+
```

The Aggregate Message format

```
+--------+-------+---------+--------+-------+---------+ - - - -
|Header 0|Message|Back     |Header 1|Message|Back     |
|        |Data 0 |Pointer 0|        |Data 1 |Pointer 1|
+--------+-------+---------+--------+-------+---------+ - - - -
```

The Aggregate Message body format

The message stream ID of the aggregate message overrides the message stream IDs of the sub-messages inside the aggregate.

The difference between the timestamps of the aggregate message and the first sub-message is the offset used to renormalize the timestamps of the sub-messages to the stream timescale. The offset is added to each sub-message’s timestamp to arrive at the normalized stream time. The timestamp of the first sub-message SHOULD be the same as the timestamp of the aggregate message, so the offset SHOULD be zero.

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.

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

7.1.7. User Control Message Events

The client or the server sends this message to notify the peer about the user control events. For information about the message format, see Section 6.2.

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 PingResponse on receiving MsgPingRequest.
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 PingRequest request.

7.2. 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.

7.2.1. 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 :

- o connect
- o call
- o close
- o createStream

7.2.1.1. 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 Arguments	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://local host:1935/test app/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
videoFunct-ion	Number	Indicates what special video functions are supported.	SUPPORT_VID _CLIENT_SEEK
pageUrl	String	URL of the web page from where the SWF file was loaded.	http:// somehost/ sample.html
object Encoding	Number	AMF encoding method.	AMF3

Flag values for the audioCodecs property:

Codec Flag	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

Flag values for the videoCodecs Property:

Codec Flag	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

Flag values for the videoFunction property:

Function Flag	Usage	Value
SUPPORT_VID_CLIENT_SEEK	Indicates that the client can perform frame-accurate seeks.	1

Values for the object encoding property:

Encoding Type	Usage	Value
AMF0	AMF0 object encoding supported by Flash 6 and later	0
AMF3	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 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.

```
+--------------+                              +-------------+
|    Client    |             |                |    Server   |
+------+-------+             |                +------+------+
       |              Handshaking done               |
       |                     |                       |
       |                     |                       |
       |                     |                       |
       |                     |                       |
       |----------- Command Message(connect) ------->|
       |                                             |
       |<------- Window Acknowledgement Size --------|
       |                                             |
       |<----------- Set Peer Bandwidth -------------|
       |                                             |
       |-------- Window Acknowledgement Size ------->|
       |                                             |
       |<------ User Control Message(StreamBegin) ---|
       |                                             |
       |<------------ Command Message ---------------|
       |       (_result- connect response)           |
       |                                             |
```

Message flow in the connect command

The message flow during the execution of the command is:

1. Client sends the connect command to the server to request to connect with the server application instance.

2. 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.

3. The server sends the protocol message 'Set Peer Bandwidth' to the client.

4. The client sends the protocol message 'Window Acknowledgement Size' to the server after processing the protocol message 'Set Peer Bandwidth'.

5. The server sends an another protocol message of type User Control Message(StreamBegin) to the client.

6. 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). In addition it specificies other connection response related information like level (string), code (string), description (string), objectencoding (number), etc.

7.2.1.2. 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 Arguments	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.
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.

7.2.1.3. 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.

7.2.2. 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 by the client to the server:

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

The server sends NetStream status updates to the client using the "onStatus" command:

Field Name	Type	Description
Command Name	String	The command name "onStatus".
Transaction ID	Number	Transaction ID set to 0.
Command Object	Null	There is no command object for onStatus messages.
Info Object	Object	An AMF object having at least the following three properties: "level" (String): the level for this message, one of "warning", "status", or "error"; "code" (String): the message code, for example "NetStream.Play.Start"; and "description" (String): a human-readable description of the message. The Info object MAY contain other properties as appropriate to the code.

Format of NetStream status message commands.

7.2.2.1. 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 of the same name. If there is no recorded stream with that name, the subscriber waits for a new live stream with that name and plays it when available. 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 you 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.

```
          +-------------+                          +----------+
          | Play Client |             |            |   Server |
          +------+------+             |            +-----+----+
                 |        Handshaking and Application       |
                 |             connect done                 |
                 |                    |                     |
                 |                    |                     |
                 |                    |                     |
                 |                    |                     |
        ---+---- |------Command Message(createStream) ----->|
     Create|     |                                          |
     Stream|     |                                          |
        ---+---- |<---------- Command Message --------------|
                 |     (_result- createStream response)     |
                 |                                          |
        ---+---- |------ Command Message (play) ----------->|
           |     |                                          |
           |     |<------------  SetChunkSize --------------|
           |     |                                          |
           |     |<---- User Control (StreamIsRecorded) ----|
      Play |     |                                          |
           |     |<---- UserControl (StreamBegin) ----------|
           |     |                                          |
           |     |<--Command Message(onStatus-play reset) --|
           |     |                                          |
           |     |<--Command Message(onStatus-play start) --|
           |     |                                          |
           |     |<-------------Audio Message---------------|
           |     |                                          |
           |     |<-------------Video Message---------------|
           |     |                    |                     |
                                      |
              Keep receiving audio and video stream till finishes
```

Message flow in the play command

The message flow during the execution of the command is:

1. The client sends the play command after receiving result of the createStream command as success from the server.

2. On receiving the play command, the server sends a protocol message to set the chunk size.

3. 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.

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

5. The server sends an 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.

7.2.2.2. 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.
Command Object	Null	Command information does not exist. Set to null type.
Parameters	Object	An AMF encoded object whose properties are the public properties described for the flash.net.NetStreamPlayOptions ActionScript object.

The public properties for the NetStreamPlayOptions object are described in the ActionScript 3 Language Reference [AS3].

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

```
            +--------------+                        +-------------+
            | Play2 Client |              |         |    Server   |
            +--------+-----+              |         +------+------+
                     |      Handshaking and Application      |
                     |               connect done            |
                     |                    |                  |
                     |                    |                  |
                     |                    |                  |
                     |                    |                  |
            ---+---- |---- Command Message(createStream) --->|
        Create |     |                                       |
        Stream |     |                                       |
            ---+---- |<---- Command Message (_result) -------|
                     |                                       |
            ---+---- |------ Command Message (play) -------->|
               |     |                                       |
               |     |<------------ SetChunkSize ------------|
               |     |                                       |
               |     |<--- UserControl (StreamIsRecorded)----|
          Play |     |                                       |
               |     |<------- UserControl (StreamBegin)-----|
               |     |                                       |
               |     |<--Command Message(onStatus-playstart)-|
               |     |                                       |
               |     |<---------- Audio Message -------------|
               |     |                                       |
               |     |<---------- Video Message -------------|
               |     |                                       |
               |     |                                       |
            ---+---- |-------- Command Message(play2) ------>|
               |     |                                       |
               |     |<------- Audio Message (new rate) -----|
         Play2 |     |                                       |
               |     |<------- Video Message (new rate) -----|
               |     |                    |                  |
               |     |                    |                  |
               |  Keep receiving audio and video stream till finishes
                                          |
```

Message flow in the play2 command

7.2.2.3. 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.

7.2.2.4. 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, if the receiveAudio command is sent with the bool flag set as false. If this flag is set to true, server responds with status messages NetStream.Seek.Notify and NetStream.Play.Start

7.2.2.5. 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, if the receiveVideo command is sent with the bool flag set as false. If this flag is set to true, server responds with status messages NetStream.Seek.Notify and NetStream.Play.Start

7.2.2.6. 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.

7.2.2.7. 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.

7.2.2.8. 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.

7.3. Message Exchange Examples

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

7.3.1. Publish Recorded Video

This 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.

```
        +--------------------+                     +-----------+
        |  Publisher Client  |        |            |    Server |
        +----------+---------+        |            +-----+-----+
                   |           Handshaking Done           |
                   |                  |                   |
                   |                  |                   |
          ---+---- |----- Command Message(connect) ----->|
             |     |                                      |
             |     |<----- Window Acknowledge Size ------|
      Connect |     |                                      |
             |     |<-------Set Peer BandWidth ----------|
             |     |                                      |
             |     |------ Window Acknowledge Size ----->|
             |     |                                      |
             |     |<------User Control(StreamBegin)-----|
             |     |                                      |
          ---+---- |<---------Command Message -----------|
                   |   (_result- connect response)        |
                   |                                      |
          ---+---- |--- Command Message(createStream)--->|
       Create |     |                                      |
       Stream |     |                                      |
          ---+---- |<------- Command Message ------------|
                   | (_result- createStream response)     |
                   |                                      |
          ---+---- |---- Command Message(publish) ------>|
             |     |                                      |
             |     |<------User Control(StreamBegin)-----|
             |     |                                      |
             |     |-----Data Message (Metadata)-------->|
             |     |                                      |
    Publishing|     |------------ Audio Data ------------>|
      Content |     |                                      |
             |     |------------ SetChunkSize ---------->|
             |     |                                      |
             |     |<----------Command Message ----------|
             |     |      (_result- publish result)       |
             |     |                                      |
             |     |------------- Video Data ----------->|
             |     |                  |                   |
             |     |                  |                   |
                   |    Until the stream is complete      |
                   |                  |                   |
```

Message flow in publishing a video stream

7.3.2. Broadcast a Shared Object Message

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

```
                  +----------+                       +----------+
                  |  Client  |           |           |  Server  |
                  +-----+----+           |           +-----+----+
                        |   Handshaking and Application    |
                        |          connect done            |
                        |                |                 |
                        |                |                 |
                        |                |                 |
                        |                |                 |
     Create and ---+---- |---- Shared Object Event(Use)---->|
     connect       |     |                                  |
     Shared Object |     |                                  |
                   ---+---- |<---- Shared Object Event---------|
                        |       (UseSuccess,Clear)         |
                        |                                  |
                   ---+---- |------ Shared Object Event ------>|
     Shared object |     |         (RequestChange)          |
     Set Property  |     |                                  |
                   ---+---- |<------ Shared Object Event ------|
                        |            (Success)             |
                        |                                  |
                   ---+---- |------- Shared Object Event ----->|
      Shared object|     |           (SendMessage)          |
      Message      |     |                                  |
      Broadcast ---+---- |<------- Shared Object Event -----|
                        |           (SendMessage)          |
                                       |                 |
                                       |                 |
```

Shared object message broadcast

7.3.3. Publish Metadata from Recorded Stream

This example describes the message exchange for publishing metadata.

```
            +------------------+                        +---------+
            | Publisher Client |         |              |   FMS   |
            +---------+--------+         |              +----+----+
                      |     Handshaking and Application     |
                      |            connect done             |
                      |                  |                  |
                      |                  |                  |
              ---+--- |---Command Messsage(createStream) -->|
          Create |    |                                     |
          Stream |    |                                     |
              ---+--- |<---------Command Message------------|
                      |   (_result - command response)      |
                      |                                     |
              ---+--- |---- Command Message(publish) ------>|
      Publishing |    |                                     |
        metadata |    |<------ UserControl(StreamBegin)-----|
       from file |    |                                     |
                 |    |-----Data Message (Metadata) ------->|
                      |                                     |
```

Publishing metadata

8. References

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1982] Elz, R. and R. Bush, "Serial Number Arithmetic", RFC 1982, August 1996.

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

[AS3] Adobe Systems, Inc., "ActionScript 3.0 Reference for the Adobe Flash Platform", 2011, <http://www.adobe.com/devnet/actionscript/documentation.html>.

[AMF0] Adobe Systems, Inc., "Action Message Format -- AMF 0", December 2007, <http://opensource.adobe.com/wiki/download/attachments/1114283/amf0_spec_121207.pdf>.

[AMF3] Adobe Systems, Inc., "Action Message Format -- AMF 3", May 2008, <http://opensource.adobe.com/wiki/download/attachments/1114283/amf3_spec_05_05_08.pdf>.

Authors' Addresses

Hardeep Singh Parmar (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110-2704
US

Phone: +1 408 536 6000
Email: hparmar@adobe.com
URI: http://www.adobe.com/

Michael C. Thornburgh (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110-2704
US

Phone: +1 408 536 6000
Email: mthornbu@adobe.com
URI: http://www.adobe.com/