RTMP

- AMF0 , AMF3 인코딩 과정과 **Red5** 소스 분석

http://www.onnuristream.com

2016.01.26

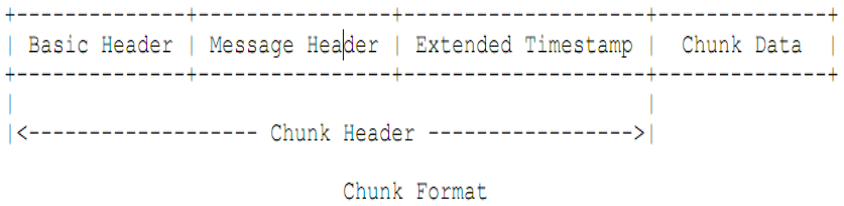
CONTENTS

1. Chunk format
2. RTMP Packet Header
3. Example) CreateStream
4. Invoke Message Structure
5. Ping Message Structure
6. Server Bw/Client Bw Message Structure
7. Set Chunk Size
8. AMF0 - data types
9. Example) AMF0
10. Red5 encode string - sequence diagram
11. Red5 AMF Output class diagram
12. Red5 AMF3 Output class diagram
13. 참고 자료

RTMP Chunk Stream

➢ Chunk Format

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

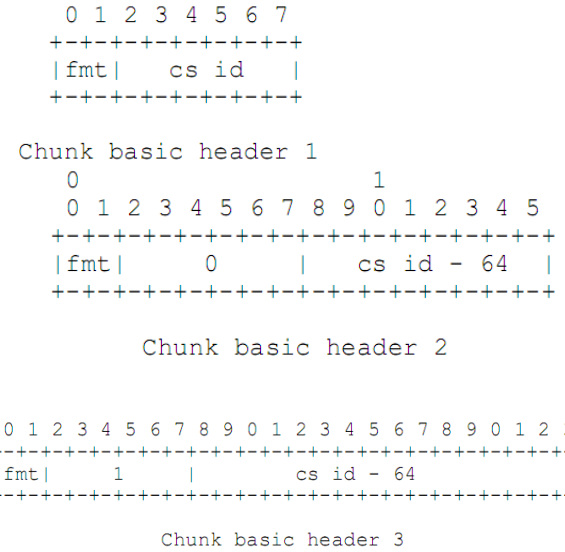


Chunk Format

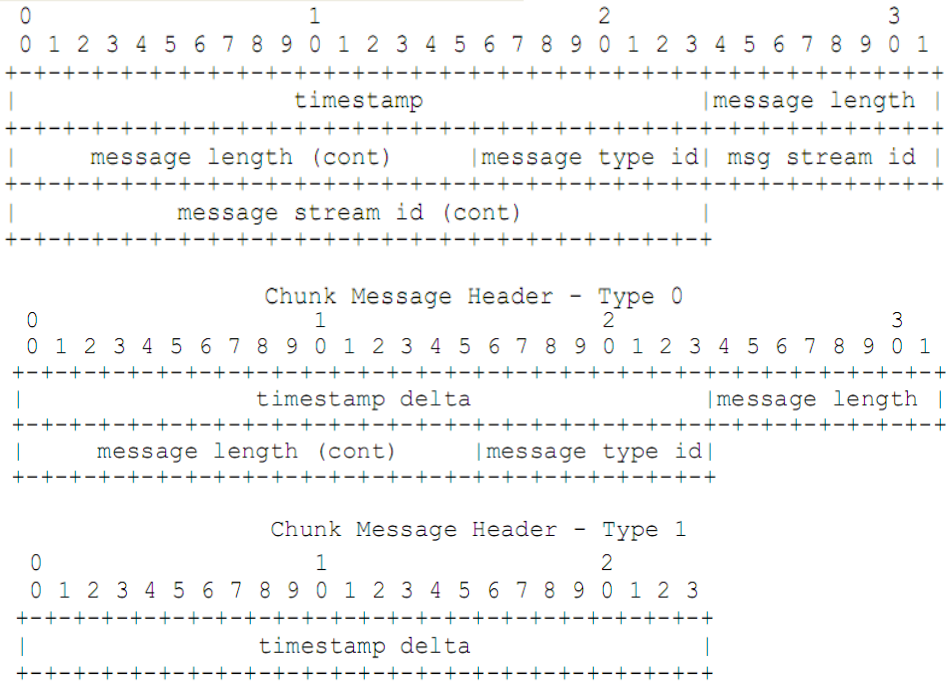
max header size: 18 bytes
Wikipedia 에서 말하는 full header size: 12 bytes

Basic Header (1 to 3 bytes)
Message Header (0,3,7, or 11 bytes)
Extended Timestamp (0 or 4 bytes)
Chunk Data (variable size)

> Basic Header
(3byte 까지 확장 가능)
- cs id: Chunk Stream ID



> Chunk Message Header
Type 0 chunk headers are 11 bytes long
Type 1 chunk headers are 7 bytes long
Type 2 chunk headers are 3 bytes long.
Type 3 chunks have no message header.



RTMP Chunk Stream

➢ Maximum Header size

red5-server-common\…\rtmp\codec\RTMPProtocolEncoder.java

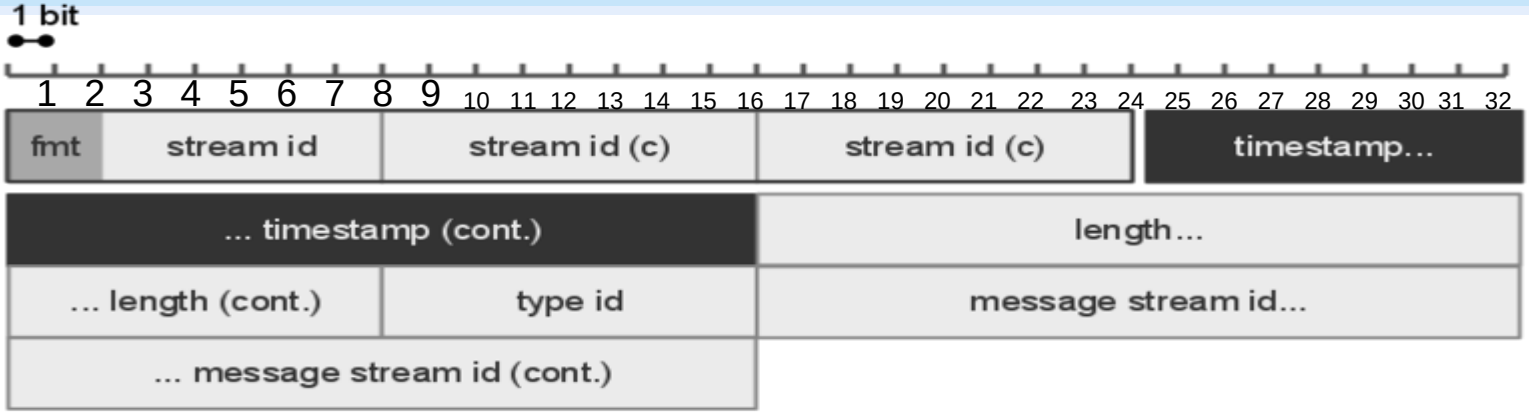
max header size: 18 bytes
Wikipedia 에서 말하는 full header size: 12 bytes

Basic Header (1 to 3 bytes)
Message Header (0,3,7, or 11 bytes)
Extended Timestamp (0 or 4 bytes)
Chunk Data (variable size)

```
116     /**
117      * Encode packet.
118      *
119      * @param packet      RTMP packet
120      * @return           Encoded data
121      */
122     public IoBuffer encodePacket(Packet packet) {
123         IoBuffer out = null;
124         final Header header = packet.getHeader();
125         final int channelId = header.getChannelId();
126         log.trace("Channel id: {}", channelId);
127         final IRTMPEvent message = packet.getMessage();
128         if (message instanceof ChunkSize) {
129             ChunkSize chunkSizeMsg = (ChunkSize) message;
130             ((RTMPConnection) Red5.getConnectionLocal()).getState().setWriteChunkSize(chunkSizeMsg.getSize());
131         }
132         // normally the message is expected not to be dropped
133         if (!dropMessage(channelId, message)) {
134             IoBuffer data = encodeMessage(header, message);
135             if (data != null) {
136                 RTMP rtmp = ((RTMPConnection) Red5.getConnectionLocal()).getState();
137                 if (data.position() != 0) {
138                     data.flip();
139                 } else {
140                     data.rewind();
141                 }
142                 int dataLen = data.limit();
143                 header.setSize(dataLen);
144                 // get last header
145                 Header lastHeader = rtmp.getLastWriteHeader(channelId);
146                 // maximum header size with extended timestamp (Chunk message header type 0 with 11 byte)
147                 int headerSize = 18;
148                 // set last write header
149                 rtmp.setLastWriteHeader(channelId, header);
150                 // set last write packet
151                 rtmp.setLastWritePacket(channelId, packet);
152                 int chunkSize = rtmp.getWriteChunkSize();
153                 // maximum chunk header size with extended timestamp
154                 int chunkHeaderSize = 7;
155                 int numChunks = (int) Math.ceil(dataLen / (float) chunkSize);
156                 int bufSize = dataLen + headerSize + (numChunks > 0 ? (numChunks - 1) * chunkHeaderSize : 0);
```

RTMP

➢ RTMP Packet Header



Header
- Basic Header : 처음 1 byte 부분 (최대 3byte 까지 확장가능)
  - fmt
  - stream id
- Chunk Message Header : 그 이외의 부분

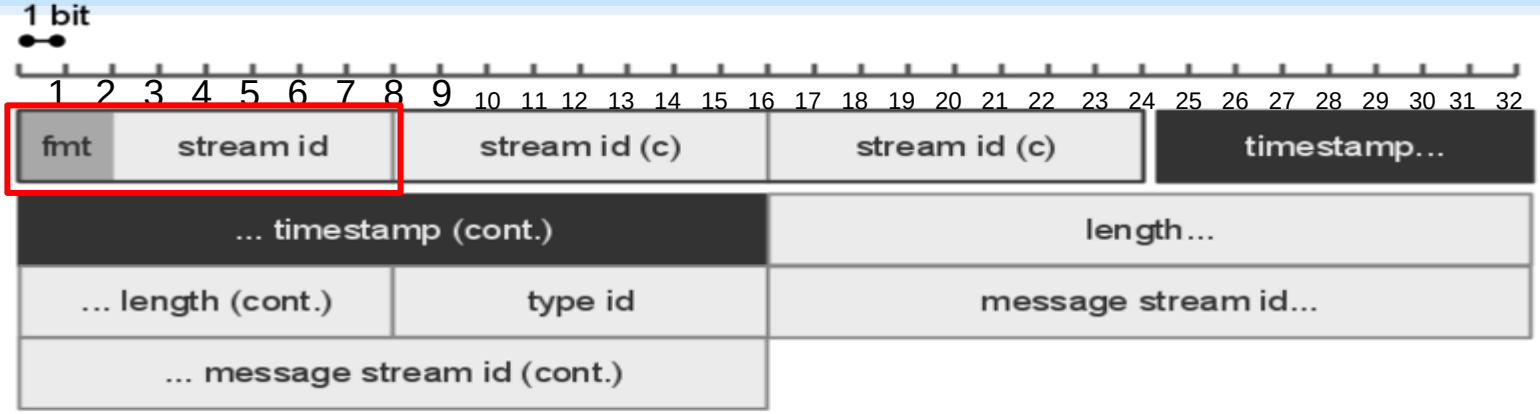
Basic Header

* 2 bit : Chuck type (fmt)
 - 00 : full header
 - 01 : 8 bytes - like type b00. not including message ID (4 last bytes).
 - 11 : Basic Header 만 가지고 있다.
 - 10 : Basic Header 와 timestamp 까지만 가지고 있다.

* 6 bit : stream ID
 - 1 : extended stream ID 라는 뜻이어서 뒤따라오는 2 byte 가 stream ID 가 된다.
 - 2 : Ping 이나, Set Client Bandwidth 메시지같은 low level message 라는 것을 의미한다.

RTMP

➢ RTMP Packet Header - Basic Header



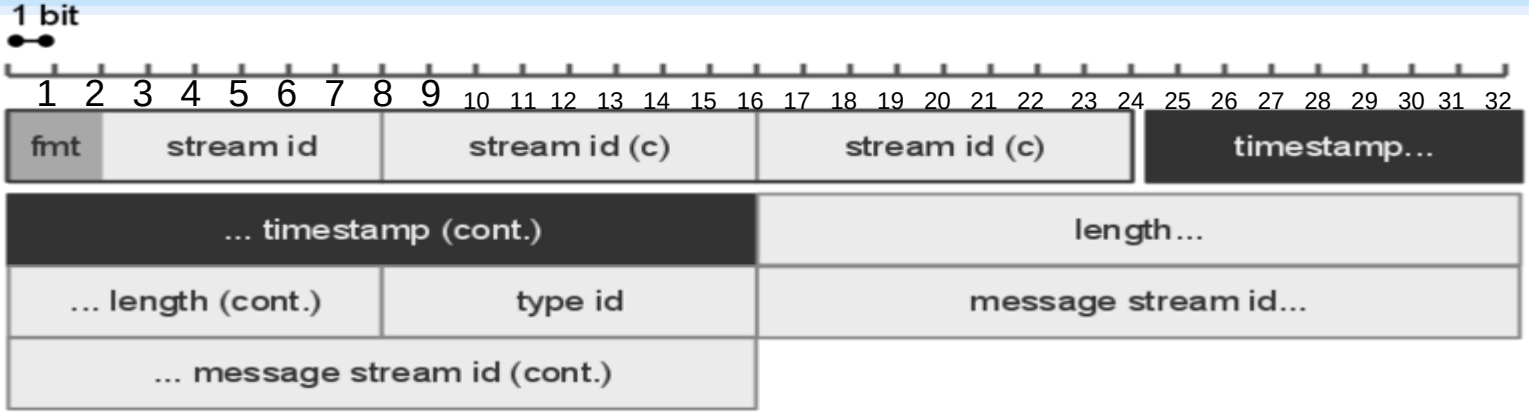
If **the value of the remaining 6 bits of the *Basic Header* (BH) (least significant) is 0** then the **BH is of 2 bytes** and represents from **Stream ID 64 to 319 (64+255)**; if **the value is 1**, then the **BH is of 3 bytes** (last 2 bytes encoded as 16bit Little Endian) and represents from **Stream ID 64 to 65599 (64+65535)**; if **the value is 2**, then BH is of 1 byte and is reserved for l**ow-level protocol control messages and commands**.

red5-server-common…\rtmp\RTMPUtils.java

```
161    /**
162     * Encodes header size marker and channel id into header marker.
163     * @param out output buffer
164     *
165     * @param headerSize         Header size marker
166     * @param channelId          Channel used
167     */
168    public static void encodeHeaderByte(IoBuffer out, byte headerSize, int channelId) {
169        if (channelId <= 63) {
170            out.put((byte) ((headerSize << 6) + channelId));
171        } else if (channelId <= 320) {
172            out.put((byte) (headerSize << 6));
173            out.put((byte) (channelId - 64));
174        } else {
175            out.put((byte) ((headerSize << 6) | 1));
176            channelId -= 64;
177            out.put((byte) (channelId & 0xff));
178            out.put((byte) (channelId >> 8));
179        }
180    }
```

RTMP

➢ RTMP Packet Header - Chunk Message Header



The **Chunk Message Header** contains meta-data information such as the message size (measured in bytes), the ***Timestamp Delta*** **and** ***Message Type***. This last value is a single byte and defines whether the packet is an audio, video, command or "low level" RTMP packet such as an RTMP Ping.

red5-server-common…\codec\RTMPProtocolEncoder.java

```
327    private byte getHeaderType(final Header header, final Header lastHeader) {
328        if (lastHeader == null) {
329            return HEADER_NEW;
330        }
331        final Integer lastFullTs = ((RTMPConnection) Red5.getConnectionLocal()).getState().getLastFullTimestampWritten(
332        if (lastFullTs == null) {
333            return HEADER_NEW;
334        }
335        final byte headerType;
336        final long diff = RTMPUtils.diffTimestamps(header.getTimer(), lastHeader.getTimer());
337        final long timeSinceFullTs = RTMPUtils.diffTimestamps(header.getTimer(), lastFullTs);
338        if (header.getStreamId() != lastHeader.getStreamId() || diff < 0 || timeSinceFullTs >= 250) {
339            // New header mark if header for another stream
340            headerType = HEADER_NEW;
341        } else if (header.getSize() != lastHeader.getSize() || header.getDataType() != lastHeader.getDataType()) {
342            // Same source header if last header data type or size differ
343            headerType = HEADER_SAME_SOURCE;
344        } else if (header.getTimer() != lastHeader.getTimer() + lastHeader.getTimerDelta()) {
345            // Timer change marker if there's time gap between header time stamps
346            headerType = HEADER_TIMER_CHANGE;
347        } else {
348            // Continue encoding
349            headerType = HEADER_CONTINUE;
350        }
351        return headerType;
```

0x00

0x01

0x02

0x03

➢ CreateStream - Basic Header

```
var stream:NetStream = new NetStream(connectionObject);
```

Header :12 bytes

Hex Code	ASCII
03 00 0b 68 00 00 19 14 00 00 00 00 02 00 0C 63 72 65 61 74 65 53 74 72	. . @ I . . . . . c r e a t e S t r
65 61 6D 00 40 00 00 00 00 00 00 00 05	e a m . @ . . . . .

The packet starts with a *Basic Header* of a single byte (0x03) where the 2 most significant bits (b**00**000011) define a chunk header type of 0 while the rest (b00**000011**) define a Chunk Stream ID of 3. The 4 possible values of the header type and their significance are:

- b00 = 12 byte header (full header).
- b01 = 8 bytes - like type b00. not including message ID (4 last bytes).
- b10 = 4 bytes - Basic Header and timestamp (3 bytes) are included.
- b11 = 1 byte - only the Basic Header is included.

➢ CreateStream - Chunk Message Header

An example is shown below as captured when a flash client executes the following code:

```
var stream:NetStream = new NetStream(connectionObject);
```

Hex Code	ASCII
01 02 03 04 05 06 07 08 09 10 11 12 03 00 0b 68 00 00 19 14 00 00 00 00 02 00 0C 63 72 65 61 74 65 53 74 72 65 61 6D 00 40 00 00 00 00 00 00 00 00 05 13 14 15 16 17 18 19 20 21 22 23 24 25	. . @ I . . . . . c r e a t e S t r e a m . @ . . . . .

The next bytes of the RTMP Header are decoded as follows:

- byte #1 (0x03) = Chunk Header Type.
- byte #2-4 (0x000b68) = Timestamp delta.
- byte #5-7 (0x000019) = Packet Length - in this case it is 0x000019 = 25 bytes.
- byte #8 (0x14) = Message Type ID - 0x14 (20) defines an **AMF0** encoded *command* message.
- byte #9-12 (0x00000000) = Message Stream ID. This (strangely) is in **little-endian** order

➢ CreateStream - Message Type ID

An example is shown below as captured when a flash client executes the following code:

```
var stream:NetStream = new NetStream(connectionObject);
```

Hex Code	ASCII
03 00 0b 68 00 00 19 14 00 00 00 00 02 00 0C 63 72 65 61 74 65 53 74 72 65 61 6D 00 40 00 00 00 00 00 00 00 00 05	. . @ I . . . . . c r e a t e S t r e a m . @ . . . . .

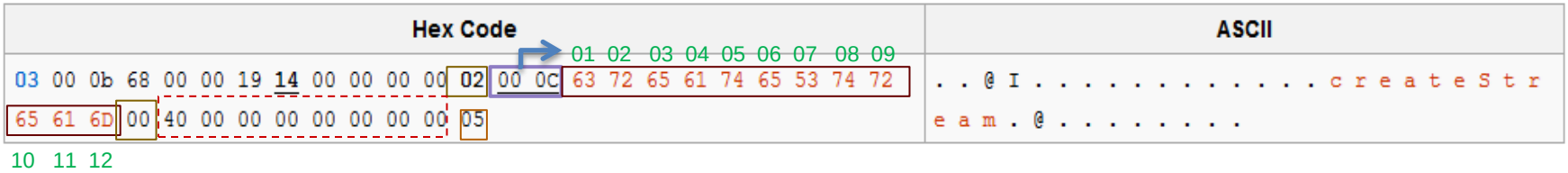
The Message Type ID byte defines whether the packet contains audio/video data, a remote object or a command. Some possible values for are:

- 0x01 = Set Packet Size Message.
- 0x04 = Ping Message.
- 0x05 = Server Bandwidth
- 0x06 = Client Bandwidth.
- 0x08 = Audio Packet.
- 0x09 = Video Packet.
- **0x11 = An AMF3 type command.**
- 0x12 = Invoke (onMetaData info is sent as such).
- **0x14 = An AMF0 type command**

➢ CreateStream - Message Type ID

An example is shown below as captured when a flash client executes the following code:

```
var stream:NetStream = new NetStream(connectionObject);
```



:double-precision floating point (8 bytes)

- Following the header, **0x02** denotes a **string** of size 0x000C and values 0x63 0x72 ... 0x6D ("createStream" command). Following that we have a **0x00 (number)** which is the transaction id of value 2.0. The last byte is **0x05 (null)** which means there are no arguments.

➢ Invoke Message Structure (0x14, 0x11)

Some of the message types shown above, such as Ping and Set Client/Server Bandwidth, are considered low level RTMP protocol messages which **do not use the AMF encoding format**. **Command messages** on the other hand, whether **AMF0 (Message Type of 0x14) or AMF3 (0x11),** use the format and have the general form shown below:

(String) <Command Name>
(Number) <Transaction Id>
(Mixed) <Argument> ex. Null, String, Object: {key1:value1, key2:value2 ... }

The transaction id is used for commands that can have a reply. The value can be either a string like in the example above or one or more objects, each composed of a set of key/value pairs where the keys are always encoded as strings while the values can be any AMF data type, including complex types like arrays.

➢ Ping Message Structure (0x04)

Ping messages are not AMF encoded. They start with a stream Id of **0x02** which implies a full (type 0) header and have a message type of 0x04. The header is followed by 6 bytes which are interpreted as such:

- #0-1 - Ping Type.
- #2-3 - Second Parameter (this has meaning in specific Ping Types)
- #4-5 - Third Parameter (same)

The first two bytes of the message body define the Ping Type which can apparently[11] take 6 possible values.

- Type 0 - Clear Stream: Sent when the connection is established and carries no further data
- Type 1 - Clear the Buffer.
- Type 3 - The client's buffer time. The third parameter holds the value in millisecond.
- Type 4 - Reset a stream.
- Type 6 - Ping the client from server. The second parameter is the current time.
- Type 7 - Pong reply from client. The second parameter is the time when the client receives the Ping.

Pong is the name for a reply to a Ping with the values used as seen above.

➢ ServerBw/ClientBw Message Structure (0x05, 0x06)

This relates to messages that have to do with the client up-stream and server down-stream bit-rate. The body is composed of 4 bytes showing the bandwidth value with a possible extension of one byte which sets the Limit Type. This can have **one of 3 possible values** which can be: **hard, soft or dynamic** (either soft or hard).

➢ Set Chunk Size (0x01)

The value received in the 4 bytes of the body. A default value of 128 bytes exists and the message is sent only when a change is wanted

➢ AMF0 - data types

The format specifies the various **data types** that can be used to encode data. Adobe states that AMF is mainly used to represent object graphs that include named properties in the form of key-value pairs, where the keys are encoded as strings and the values can be of any data type such as strings or numbers as well as arrays and other objects. XML is supported as a native type. Each type is denoted by a single byte preceding the actual data. The values of that byte are as below (for AMF0):

- **Number - 0x00** (Encoded as IEEE 64-bit double-precision floating point number)
- Boolean - 0x01 (Encoded as a single byte of value 0x00 or 0x01)
- **String - 0x02** (16-bit integer string length with UTF-8 string)
- **Object - 0x03 (Set of key/value pairs)**
- Null - 0x05
- ECMA Array - 0x08 (32-bit entry count)
- **Object End - 0x09 (preceded by an empty 16-bit string length)**
- Strict Array - 0x0a (32-bit entry count)
- Date - 0x0b (Encoded as IEEE 64-bit double-precision floating point number with 16-bit integer timezone offset)
- Long String - 0x0c (32-bit integer string length with UTF-8 string)
- XML Document - 0x0f (32-bit integer string length with UTF-8 string)
- Typed Object - 0x10 (16-bit integer name length with UTF-8 name, followed by entries)
- Switch to AMF3 - 0x11

➢ AMF0 - objects

AMF objects begin with a **(0x03)** followed by a set of key-value pairs and end with a **(0x09)** as value (preceded by 0x00 0x00 as empty key entry). Keys are encoded as strings with the (0x02) 'type-definition' byte being implied (not included in the message). Values can be of any type including other objects and whole object graphs can be serialized in this way. Both object keys and strings are preceded by **two bytes** denoting their **length** in number of bytes. This means that **strings** are preceded by **a total of three bytes which includes the 0x02 type byte**. Null types only contain their type-definition (0x05). **Numbers** are encoded as double-precision floating point and are composed of **eight bytes.**

➢ example - AMF0

As an example, when encoding the object below in actionscript 3 code.

```
var person:Object = {name:'Mike', age:'30', alias:'Mike'};
var stream:ByteArray = new ByteArray();
stream.objectEncoding = ObjectEncoding.AMF0; // ByteArray defaults to AMF3
stream.writeObject(person);
```

The data held in the ByteArray is:

Hex code	ASCII
03 00 04 6e 61 6d 65 02 00 04 4d 69 6b 65 00 03 61 67 65 00 40 3e 00 00 00 00 00 00 00 05 61 6c 69 61 73 02 00 04 4d 69 6b 65 00 00 09	. . . n a m e . . . M i k e . . a g e . @ > . . . . a l i a s . . . M i k e . . .

legend: **object start/end** object keys **object values** ecma_array

:length

:double-precision floating point (8 bytes)

➢ example (2) - AMF0

(command) "_result"
(transaction id) 1
(value)
[1] { fmsVer: "FMS/3,5,5,2004"
capabilities: 31.0
mode: 1.0 },
[2] { level: "status",
code: "NetConnection.Connect.Success",
description: "Connection succeeded",
data: (array) {
version: "3,5,5,2004" },
clientId: 1584259571.0,
objectEncoding: 3.0 }

The AMF message starts with a **0x03** which denotes an RTMP packet with Header Type of 0, so **12 bytes** are expected to follow. It is of Message Type **0x14,** which denotes a **command** in the form of a string of value "_result" and two serialized objects as arguments. The message can be decoded as follows:

Hex code	ASCII
03 00 00 00 00 01 05 14 00 00 00 00 02 00 07 5F 72 65	. . . . . _ r e
73 75 6C 74 00 3F F0 00 00 00 00 00 00 03 00 06 66 6D	s u l t . ? . . . . . f m
73 56 65 72 02 00 0E 46 4D 53 2F 33 2C 35 2C 35 2C 32	s V e r . . . F M S / 3 , 5 , 5 , 2
30 30 34 00 0C 63 61 70 61 62 69 6C 69 74 69 65 73 00	0 0 4 . . c a p a b i l i t i e s .
40 3F 00 00 00 00 00 00 00 04 6D 6F 64 65 00 3F F0 00	@ ? . . . . . m o d e . ? .
00 00 00 00 00 00 00 09 03 00 05 6C 65 76 65 6C 02 00	. . . . . l e v e l . .
06 73 74 61 74 75 73 00 04 63 6F 64 65 02 00 1D 4E 65	. s t a t u s . . c o d e . . N e
74 43 6F 6E 6E 65 63 74 69 6F 6E 2E 43 6F 6E 6E 65 63	t C o n n e c t i o n . C o n n e c
74 2E 53 75 63 63 65 73 73 00 0B 64 65 73 63 72 69 70	t . S u c c e s s . . d e s c r i p
74 69 6F 6E 02 00 15 43 6F 6E 6E 65 63 74 69 6F 6E 20	t i o n . . . C o n n e c t i o n
73 75 63 63 65 65 64 65 64 2E 00 04 64 61 74 61 08 00	s u c c e e d e d . . d a t a . .
00 00 01 00 07 76 65 72 73 69 6F 6E 02 00 0A 33 2C 35	. . . . . v e r s i o n . . . 3 , 5
2C 35 2C 32 30 30 34 00 00 09 00 08 63 6C 69 65 6E 74	, 5 , 2 0 0 4 . . . . . c l i e n t
69 64 00 41 D7 9B 78 7C C0 00 00 00 0E 6F 62 6A 65 63	i d . A . . x . . . . . o b j e c
74 45 6E 63 6F 64 69 6E 67 00 40 08 00 00 00 00 00 00	t E n c o d i n g . @ . . . . .
00 00 09	. . .

08 :ECMA Array

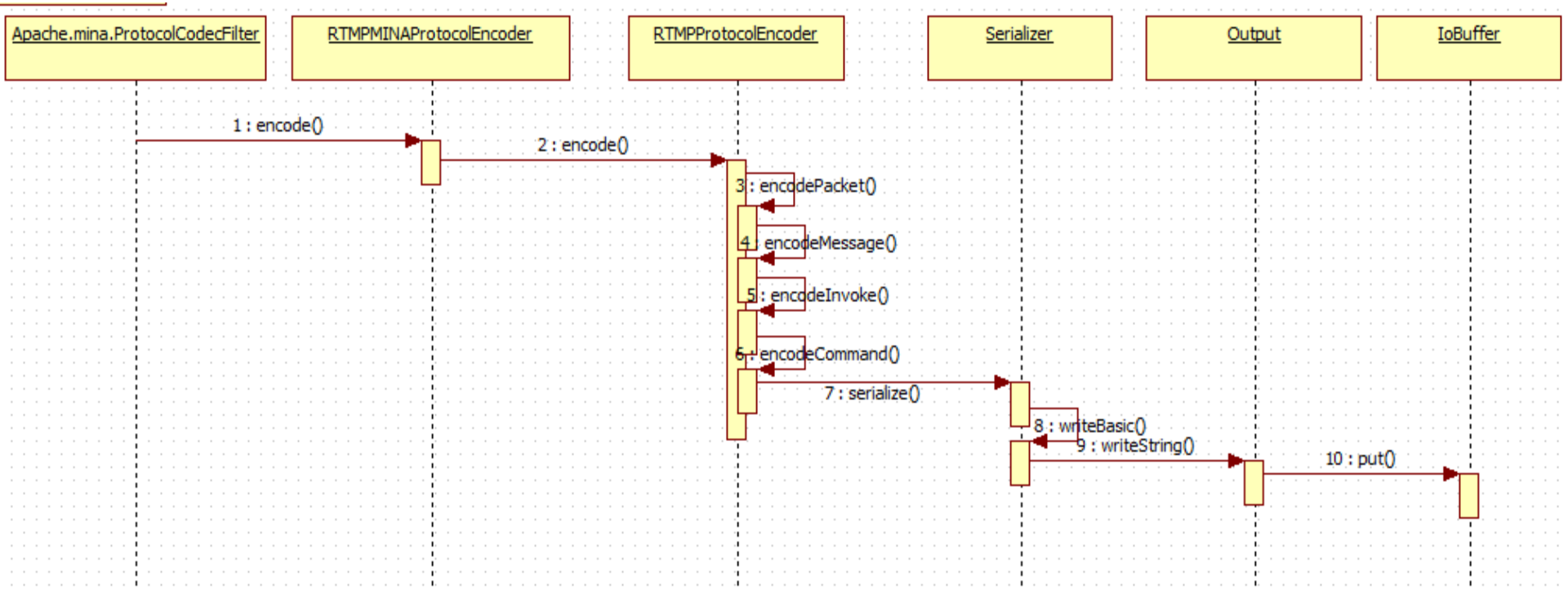
:length

:double-precision floating point (8 bytes)

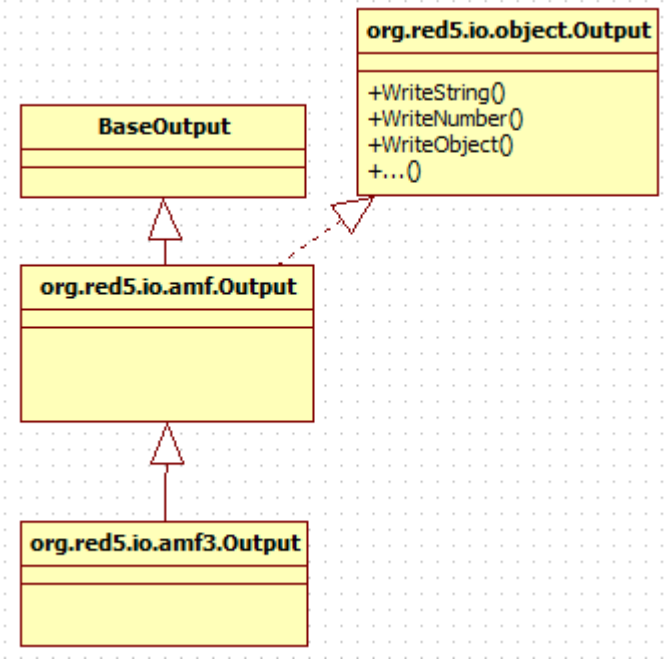
legend: **object start/end** object keys **object values** ecma_array

RTMP

➢ Red5: Encode string 과정 sequence diagram



➢ Red5: AMF Output class diagram



Red5-io/src/main/java/org/red5/io/amf/Output.java

```
486    public void writeString(String string) {
487        final byte[] encoded = encodeString(string);
488        final int len = encoded.length;
489        if (len < AMF.LONG_STRING_LENGTH) {
490            buf.put(AMF.TYPE_STRING);
491            buf.putShort((short) len);
492        } else {
493            buf.put(AMF.TYPE_LONG_STRING);
494            buf.putInt(len);
495        }
496        buf.put(encoded);
497    }
```

0x02

length

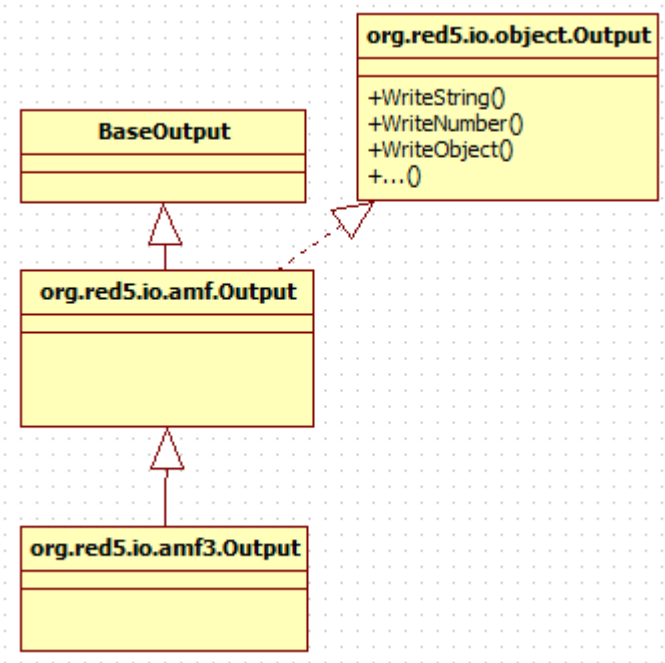
Encoded str.

```
286    public void writeNumber(Number num) {
287        buf.put(AMF.TYPE_NUMBER);
288        buf.putDouble(num.doubleValue());
289    }
```

0x00

double-precision floating point.

➢ Red5: AMF3 Output class diagram



Red5-io/src/main/java/org/red5/io/amf3/Output.java

```
204    @Override
205    public void writeString(String string) {
206        writeAMF3();
207        buf.put(AMF3.TYPE_STRING);
208        if ("".equals(string)) {
209            putInteger(1);
210        } else {
211            final byte[] encoded = encodeString(string);
212            putString(string, encoded);
213        }
214    }
215
216    /** {@inheritDoc} */
```

0x06

Console | Tasks | Search | Declaration

void org.red5.io.amf3.Output.putString(String str, byte[] encoded)

```
protected void putString(String str, byte[] encoded) {
    final int len = encoded.length;
    Integer pos = stringReferences.get(str);
    if (pos != null) {
        // Reference to existing string
        putInteger(pos << 1);
        return;
    }
    putInteger(len << 1 | 1);
    buf.put(encoded);
```

length

Encoded str.

```
188    public void writeNumber(Number num) {
189        writeAMF3();
190        if (num.longValue() < AMF3.MIN_INTEGER_VALUE || num.longValue() > AMF3.MAX_INTEGER_VALUE) {
191            // Out of range for integer encoding
192            buf.put(AMF3.TYPE_NUMBER);
193            buf.putDouble(num.doubleValue());
194        } else if (num instanceof Long || num instanceof Integer || num instanceof Short || num instanceof Byte) {
195            buf.put(AMF3.TYPE_INTEGER);
196            putInteger(num.longValue());
197        } else {
198            buf.put(AMF3.TYPE_NUMBER);
199            buf.putDouble(num.doubleValue());
200        }
201    }
```

0x05

double-precision floating point.

➢ 참고 자료

- rtmp spec.pdf

  https://www.adobe.com/content/dam/Adobe/en/devnet/rtmp/pdf/rtmp_specification_1.0.pdf

- AMF0 spec.pdf

  http://download.macromedia.com/pub/labs/amf/amf0_spec_121207.pdf

- AMF3 spec.pdf

  http://download.macromedia.com/pub/labs/amf/amf3_spec_121207.pdf

- Red5 in github

  https://github.com/Red5

- Action Message Format

  https://en.wikipedia.org/wiki/Action_Message_Format

Thank You!