RTMP Protocol [DRAFT]

Introduction

RTMP is a protocol used by the Flash Player to deliver real time objects, video, and audio to clients using a binary TCP connection or polling HTTP tunnel.

The protocol is a container for data packets which may be AMF or raw audio/video data like found in the FLV File Format.

A single connection is capable of multiplexing many net streams using different channels. Within these channels packets are split up into fixed size body chunks.

Connection

Sample ActionScript for connecting and playing a stream:

```
var videoInstance:Video = your_video_instance;
var nc:NetConnection = new NetConnection();
var connected:Boolean = nc.connect("rtmp:/localhost/myapp");
var ns:NetStream = new NetStream(nc);
videoInstance.attachVideo(ns);
ns.play("flvName");
```

Default port is 1935

Handshake

Client → Server : Sends Handshake Request. This is not a protocol packet but a single byte (0×03) followed by 1536 bytes. This content does not seem to be vital for the protocol, but is not random either. See 1)

Server → Client : Sends a Handshake Response. This is not a RTMP packet but a single byte (0×03) followed by two 1536 byte chunks (so a total of 3072 raw bytes). The second chunk of bytes is the original client request bytes sent in handshake request. The first chunk can be anything. Use null bytes it doesnt seem to matter.

Client→Server: Sends 1536 raw bytes which are the second 1536 chunk of server generated handshake.

At this time, handshake is done and further packets are RTMP ones.

Client → Server : send the Connect RTMP packet.

Server → Client : Server responds

...and so on...

RTMP Datatypes

0×01	Chunk Size	changes the chunk size for packets
0×02	Unknown	anyone know this one?
0×03	Bytes Read	send every x bytes read by both sides
0×04	Ping	ping is a stream control message, has subtypes
0×05	Server BW	the servers downstream bw
0×06	Client BW	the clients upstream bw
0×07	Unknown	anyone know this one?
0×08	Audio Data	packet containing audio
0×09	Video Data	packet containing video data
0x0A - 0xE	Unknown	anyone know?
0x0F	Flex Stream	Stream with variable length
0×10	Flex Shared Object	Shared object with variable length

0×11	Flex Message	Shared message with variable length
0×12	Notify	an invoke which does not expect a reply
0×13	Shared Object	has subtypes
0×14	Invoke	like remoting call, used for stream actions too.

Shared Object DataTypes

0×01 Connect
0×02 Disconnect
0×03 Set Attribute
0×04 Update Data
0×05 Update Attribute
0×06 Send Message
0×07 Status
0×08 Clear Data
0×09 Delete Data
0x0A Delete Attribute
0x0B Initial Data

RTMP Packet Structure

RTMP Packets consist of a fixed-length header and a variable length body that has a default of 128 bytes. The header can come in one of four sizes: 12, 8, 4, or 1 byte(s).

The two most significant bits of the first byte of the packet (which also counts as the first byte of the header) determine the length of the header. They can be extracted by ANDing the byte with the mask 0xC0. The possible header lengths are specified in the table below:

Bits	Header Length
00	12 bytes
01	8 bytes
10	4 bytes
11	1 byte

The header excludes information in the shorter version and implies that the information that is excluded is the same as the last time that information was explicitly included in the header.

In a full 12 byte header is broken down as follows:

The first byte has the header size and the object id. The first two bits are the size of the header and the following 6 bits are the object id. This limits a RTMP packets to a maximum of 64 objects in one packet. This byte is always sent no matter the size of the header.

The next three bytes are the time stamp. This is an integer (is is big-endian or little-endian?) and it is sent whenever the header size is 4 bytes or larger.

The next three bits are the length of the object body. This is an integer and appears to be big-endian. The length of the object is the size of the AMF in the RTMP packet without the RTMP headers, so you need to remove any RTMP headers before this number matches properly. These bytes are sent whenever the header size is 8 or more.

The next single byte is the content type. The content types are listed in another section this page and are only included when the header is 8 bytes or longer.

The final 4 bytes of the header is a stream id. This is a 32 bit integer that is little-endian encoded. These bytes are only included when the header is a full 12 bytes.

Also, checkout the document below. Its mostly accurate, although where it talks about AMF its really RTMP. AMF is used in RTMP, but its not covered in the following document.

Mick’s Breakdown of RTMP: JPG / PDF

Streaming

For basic publish cycle this is what happens :

Client→Server : sends a CreateStream request *(is it a single RTMP packet ?)*

The createstream request is a single AFM0 function call (remote method invocation) whose high-level equivalent function signature would be "createstream(double ClientStream, NULL)" The ClientStream variable starts at 1 and is incremented by one for every stream that is created in a connection. It is NOT used by the server to route data for multimedia streams.

Server→Client : sends a response with a streamIndex number

The response the server sends back is also an AMF0 call, this time targeted to the client-side function "_result(double ClientStream, NULL, double ServerStream)" where the ClientStream value is the same one provided in the request to create the stream supplied by the client, above, and the ServerStream value is generated by the server to identify the stream over which data will be routed. Most servers that work with Flash clients appear to simply increment a value starting from one, just like the client does, but the ClientStream does not have to match the ServerStream.

Client→Server : does a publish *(what does it means in this context ?)*

Yet another AMF0 call, this time to a server method "publish(double 0, NULL, "resource_name", "options")" The first variable has always been zero in the sessions I have captured, but I have no idea what it is intended to reflect. Ditto for the NULL in the second parameter position. The resource_name parameter is a string that identifies the resource to which the stream is to be attached. For FMS-like servers, this appears to simply be a file name in in a directory on the disk, although for real-time video stream reflection it is probably irrelevant as long as the server can associate it between connections.

Client→Server : send the audio video packets (the packets are sent from the source as indicated on the streamIndex via the same channel as the publish request)

Now, this is what's really confusing about this god-awful RTMP protocol. The authors of the protocol were either intentionally attempting to obfuscate it to prevent compatibility-related reverse engineering with packet sniffers (which is all that is legal), or they were simply using some sort of higher-level representation for the protocol that resulted in very inconsistent low-level data going out the wire (much more likely). The way that AV data is associated with a stream once a publish() command has succeeded, for example, is by virtue of an integer field in the RTMP packet header. That's where the hilarity begins.

The server tells the client what stream to use by sending a double-precision floating-point value in its _result() call to the client, as described above. Fine so far. The client, for live streams at least, then sends an RTMP-level set_buffer(S, T) command back to the server, where I'm using 'S' to represent the stream being targeted and 'T' as the buffering time desired. Both of these parameters are sent as 32-bit big-endian integers.

Finally, the server replies with an RTMP-level reset(S) command, also using a big-endian integer to identify the stream, and an AMF0-formatted status message of the following form:

```
[C:-:S](RMI) onStatus(0, NULL, struct {
    "level" => "status",
    "code" => "NetStream.Publish.Start",
    "description" => "<resource> is now published.",
    "cliendid" => <some_double-precision_value>
});
```

where I'm using the shorthand [C:T:S] to indicate the channel, time-index and stream values in the RTMP header, respectively. Here the stream identifier is sent in the header instead of as a call parameter (this is also how actual raw AV data is tagged). Guess what: that stream field in the header is a LITTLE-ENDIAN 32-bit integer.

So, the RTMP protocol uses 64-bit floats, 32-bit little-endian integers, and 32-bit big-endian integers to identify the exact same stream in three different contexts, in back-to-back messages sent between client and server.

If anyone is still interested, I may write up an annotated packet trace and post a PDF of it here. I wound up going back to the protocol analyzer because it turned out to be easier than trying to read the RED5 source  I'm just not a Java affictionado–actually, I'm writing up a real-time, media-only reflector in Erlang for an application that I cannot make public, but I am willing and able to provide any information I glean about RTMP back to the community (although it may not be much, in contrast to what RED5 appears to be capable of doing, which is why it was easier to analyze the wire rather than read the source). –ovrlod3

The PDF would be great to help on my Erlang version of this -SimpleEnigma

(this needs more detailed description I think, maybe adding an introduction explaining in general how streams are identified and that RTMP allow stream interleaving over the same connection)