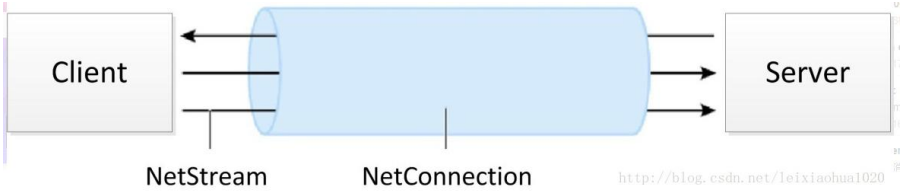
https://blog.csdn.net/leixiaohua1020/article/details/11704355
https://blog.csdn.net/leixiaohua1020/article/details/11694129
http://www.cnblogs.com/Kingfans/p/7083100.html

rtmp协议分析:

媒体数据 --> 消息 --> 块 --> tcp发送 <----> tcp接受 --> 块 --> 消息 --> 媒体数据

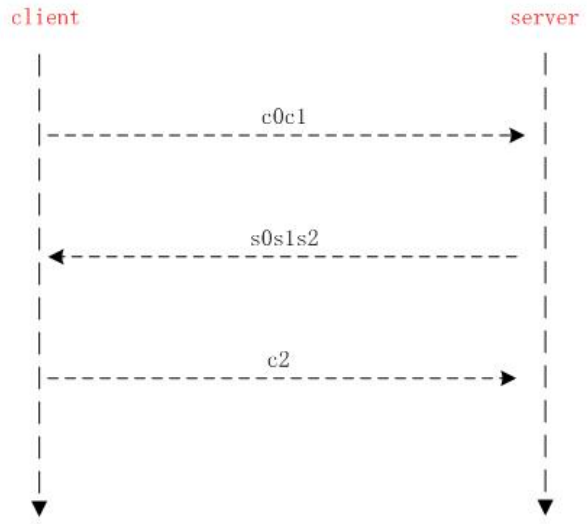


Message stream (消息流)：通信中消息流通的一个逻辑通道；
Message stream ID (消息流 ID)：每个消息有一个关联的 ID，使用 ID 可以识别出流通中的消息流。
Chunk (块)：消息的一段。消息在网络发送之前被拆分成很多小的部分。块可以确保端到端交付所有消息有序 timestamp，即使有很多不同的流。
Chunk stream (块流)：通信中允许块流向一个特定方向的逻辑通道。块流可以从客户端流向服务器，也可以从服务器流向客户端。
Chunk stream ID (块流 ID)：每个块有一个关联的 ID，使用 ID 可以识别出流通中的块流。

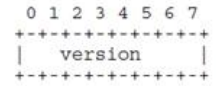
RTMP协议规定，播放一个流媒体有两个前提步骤：
第一步，建立一个网络连接（NetConnection）；
第二步，建立一个网络流（NetStream）；
其中，网络连接代表服务器端应用程序和客户端之间基础的连通关系；网络流代表了发送多媒体数据的通道。服务器和一个客户端之间只能建立一个网络连接，但是基于该连接可以创建很多网络流；

播放一个RTMP协议的流媒体需要经过以下几个步骤：握手，建立连接，建立流，播放。RTMP连接都是以握手作为开始的。建立连接阶段用于建立客户端与服务器之间的“网络连接”；建立流阶段用于建立客户端与服务器之间的“网络流”；播放阶段用于传输视音频数据。

1、握手:
rtmp的握手协商，其实也可以看做是三次握手，客户端首先发送c0c1给服务器；服务器收到c0c1后，根据c0c1的内容，生成s0s1s2，将s0s1s2发送给客户端；客户端再根据收到的s0s1s2生成c2，将c2发送给服务器，服务器接受c2完成握手。握手过程简图如下:

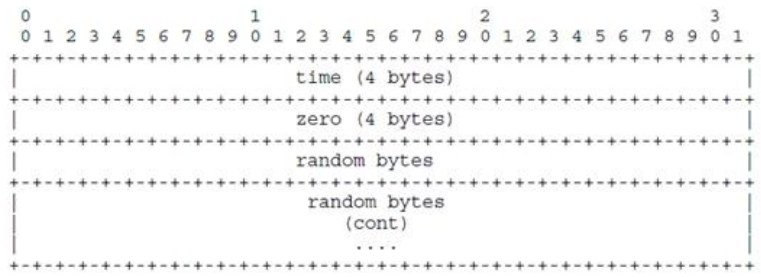


c0: 只占一个字节，表示版本信息，其值为0x03;



C0 and S0 bits

c1,s1: c1和s1的格式相同，占1536个字节，详细结构如下：



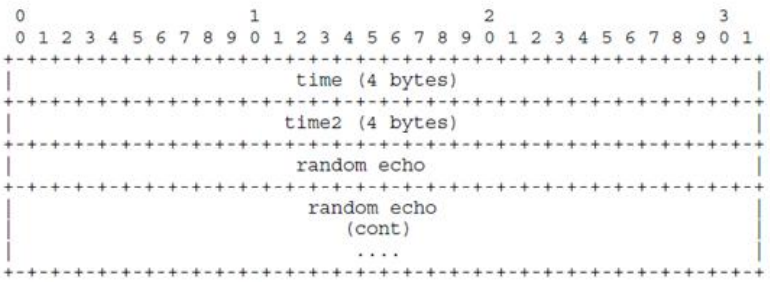
C1 and S1 bits

time：c1或者s1发送时，所在服务器的系统时间；

zero：简单握手时，该值为0。加密握手时，该值为版本信息；

random：简单握手时为1528字节的随机值。加密握手时，该字段由两部分组成：

digest(764 bytes) + key(764 bytes)；



C2 and S2 bits

c2,s2：c2和s2的格式相同，占1536字节；

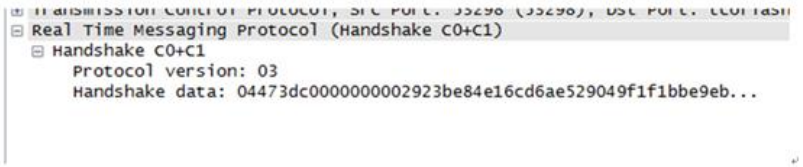
time：发送c2或者s2时的系统时间。加密握手时，可用随机值填充；

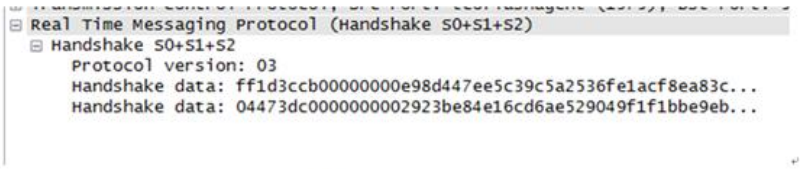
time2：c2时为s1中time的值，s2时为c1中time的值。加密握手时，可用随机值填充；

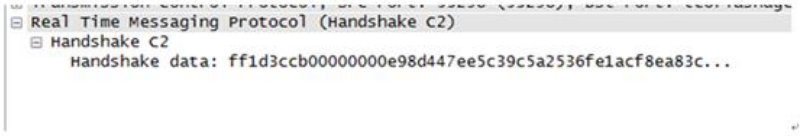
random：简单握手时，为1528字节的随机值。加密握手时，该字段由两部分组成：

rand(1496 byte) + digest-data(32 byte);

简单未加密握手：未加密的握手非常简单，客户端生成c0c1，c0值为0x03，c1的time字段为发送c0c1时的客户端机器的系统时间，zero字段填零，后面的1528字节用随机数填充，构造完成c0c1后，发送给服务端；服务端只要收到了1537个字节的数据，并且第一字节为0x03，就认为收到了c0c1。然后构造s0s1s2，发送给客户端，s0s1的生成过程与c0c1相同，s2就是c1的拷贝；客户端收到s0s1s2，向服务端发送c2，c2为s1的拷贝，服务端收到1536字节长度的c2，就完成了rtmp的简单握手。抓包截图如下：







加密握手：对明文加密，进行发送，主要就是对c1/s1的一个32字节长的字段用openssl进行加密。

c0: 0x03；

c1: time(4byte) + version(4byte) + digest(764byte) + key(764byte)；

time: 系统当前时间;

version: 版本信息;

digest: random_4(4byte)+random_offset(byte) + _digest(32byte) + random_764-4-offset-32(byte)

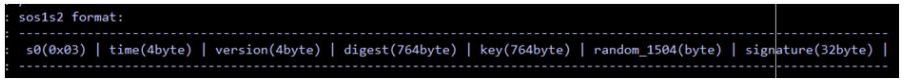
random_4: 4字节的随机值，根据这4字节计算random_offset的长度;

random_offset: 随机值填充;

_digest: 这个是32字节的加密字段，通过openssl生成;

random_764-4-offset-32: 随机值填充;

key: 764字节的随机值填充;



s0: 0x03；

s1: time(4byte) + version(4byte) + digest(764byte) + key(764byte)；

time:发送s1时的系统时间;

version: 版本信息;

digest: random_4(4byte)+random_offset(byte)+_digest(32byte)+random_764-4-offset-32(byte)

random_4: 4字节的随机值，根据这4字节计算random_offset的长度;

random_offset: 随机值填充;

_digest: 这个是32字节的加密字段，通过openssl生成;

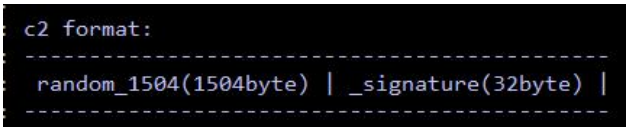
random_764-4-offset-32: 随机值填充;

key: 764字节的随机值填充, srs代码里128字节的_key，是由c1的key变换生成的，但是这个值貌似没有什么用，用随机值填充也是可以的，librtmp就是用随机值填充的；

s2: random_1504(byte) + _signature(32byte)

random_1504: 可以用1504个字节的随机数填充；也可以拷贝c1的前1504字节(librtmp做法)；

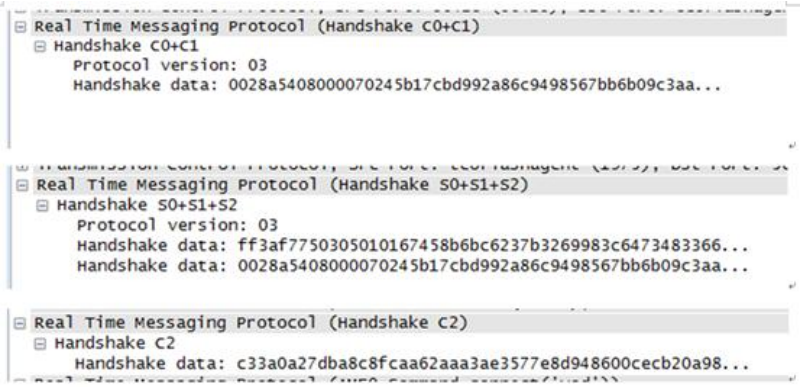
_signature: 根据c1的_digest字段生成temp_key, 再根据temp_key和1504字节的random生成s2的_digest；客户端不会校验s2；



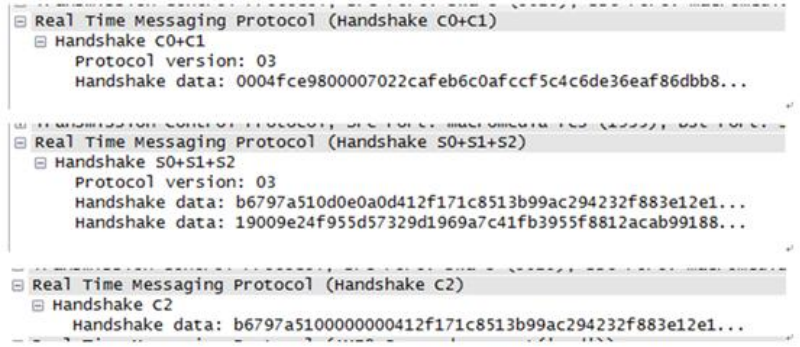
c2: random_1504(byte) + _signature(32byte)

random_1504: 1504个字节的随机数填充；

_signature: 根据s1的_digest字段生成temp_key, 再根据temp_key和1504字节的random生成c2的_digest；



上图是flash player与librtmp服务器交互数据包的截图，可以看见s2的前面部分与c1相同，而c2为随机生成；下图为flash player与nginx交互数据包的截图，s2为随机数据填充，c2与s1存在相同数据。不同的服务程序，s2和c2的生成是可以不同的，但是c1和s1的生成规则则是固定的，也是最关键的。



2、建立连接:

a)、客户端发送命令消息中的“连接”(connect)到服务器，请求与一个服务应用实例建立连接；

b)、服务器接收到连接命令消息后，发送确认窗口大小(Window Acknowledgement Size)协议消息到客户端，同时连接到连接命令中提到的应用程序；

c)、服务器发送设置带宽()协议消息到客户端；

d)、客户端处理设置带宽协议消息后，发送确认窗口大小(Window Acknowledgement Size)协议消息到服务器端；

e)、服务器发送用户控制消息中的“流开始”(Stream Begin)消息到客户端；

f)、 服务器发送命令消息中的“结果”(_result)，通知客户端连接的状态；

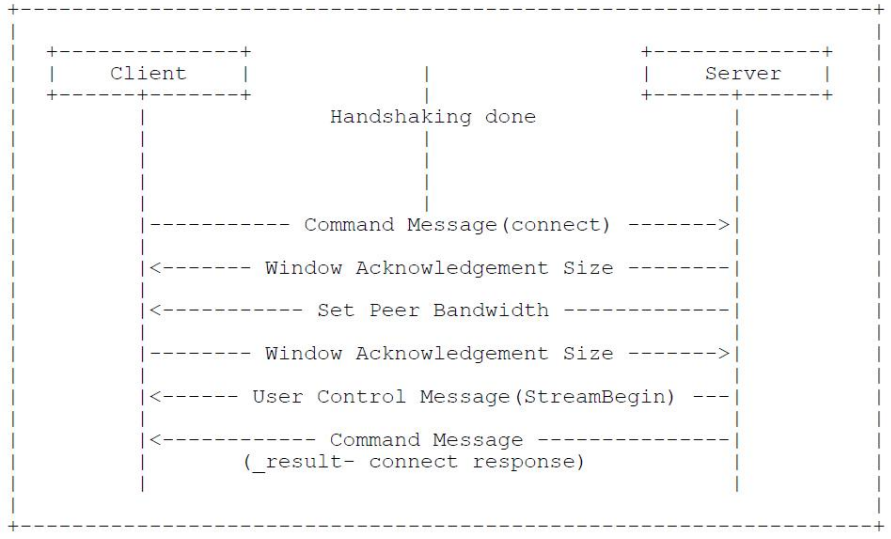
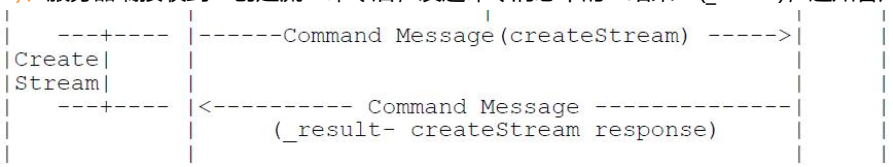


Figure 4 Message flow in the connect command

3、建立网络:

a)、客户端发送命令消息中的“创建流” （createStream）命令到服务器端;

b)、服务器端接收到“创建流”命令后，发送命令消息中的“结果”(_result)，通知客户端流的状态;



4、播放:

a)、客户端发送命令消息中的“播放” （play）命令到服务器;

b)、接收到播放命令后，服务器发送设置块大小（ChunkSize）协议消息;

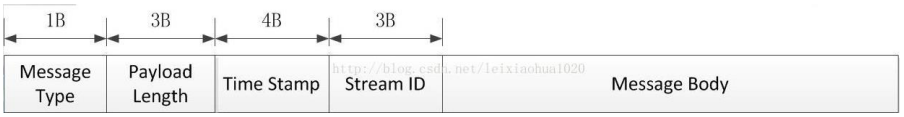
c)、服务器发送用户控制消息中的“streambegin”，告知客户端流ID;

d)、播放命令成功的话，服务器发送命令消息中的“响应状态” NetStream.Play.Start & NetStream.Play.reset，告知客户端“播放”命令执行成功;

e)、在此之后服务器发送客户端要播放的音频和视频数据;

==========

消息(Message)：消息是rtmp协议中基本数据单元，客户端和服务端通过消息传递音频、视频和其他数据；消息的具体格式如下:



Message Type (消息类型)：一个字节的字段来表示消息类型;

Payload Length (负载长度)：三个字节的字段来表示有效负载的字节数，以大端格式保存;

Time Stamp(时间戳)：四个字节的字段包含了当前消息的 timestamp，四个字节也以大端格式保存;

Stream Id (消息流 ID)：三个字节的字段以指示出当前消息的流，这三个字节以大端格式保存;

rtmp有很多消息类型，网络收发时，必须说明该message是那种类型的消息，不然客户端接受到数据也无法继续处理，所以Message Header中必须携带Message Type字段，用以说明发送的是那种消息；Message Type说明了消息类型，客户端接受时，还必须知道该消息的长度，才能正确的组装该消息，所以消息传递时，必须负载长度字段Payload Length；NetConnection网络连接中可以传输多个NetStream，所以消息在发送时，还必须标明属于哪一个NetStream，Stream ID字段用来标注该消息的归属；Time Stamp为消息的发送时间;

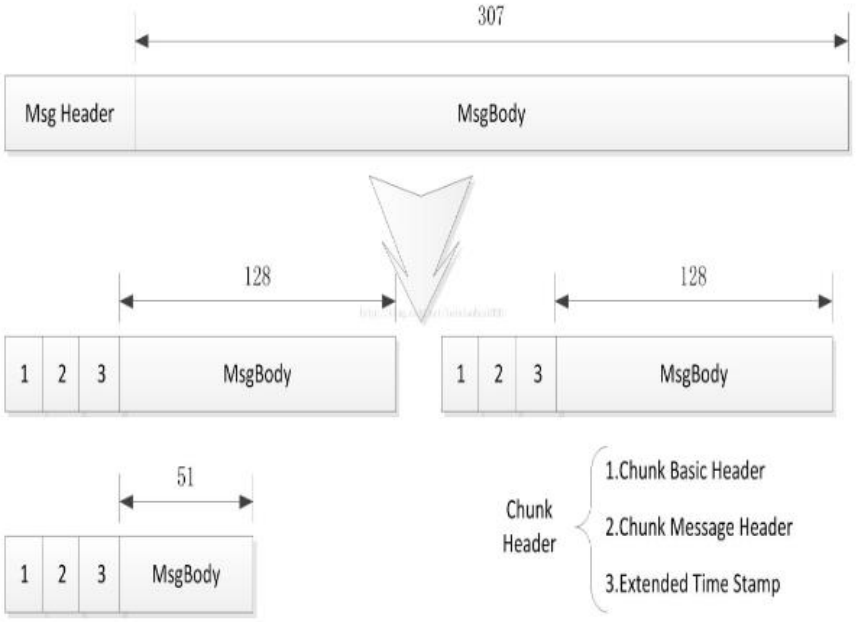
块(chunk)： 网络传输时，消息被分割成很多的块，块才是网络传输时的基本单位；每个块都有一个唯一 ID，这个 ID 叫做 chunk stream ID (块流 ID)；块通过网络进行传输时，每个块必须被完全发送才可以发送下一块；在接收端，这些块被根据块流 ID 被组装成消息；块的大小是可以配置的，它可以使用一个设置块大小的控制消息进行设置 (SetChunkSize)；更大的块大小可以降低 CPU 开销，但在低带宽连接时因为它的大量的写入也会延迟其他内容的传递，更小的块不利于高比特率的流化。所以块的大小设置取决于具体情况，一般设置为4096;

为什么要分块：

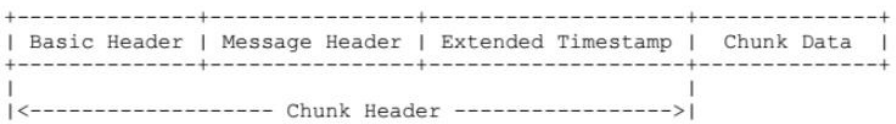
1、分块允许上层协议将大的消息分解为更小的消息，防止体积大的但优先级小的消息 (比如视频) 阻碍体积较小但优先级高的消息 (比如音频或者控制命令)；

2、分块也让我们能够使用较小开销发送小消息，因为块头包含包含在消息内部的信息压缩提示?

在消息被分割成几个消息块的过程中，消息负载部分(Message Body)被分割成大小固定的数据块(默认是128字节，最后一个数据块可以小于该固定长度)，并在其首部加上消息块首部(Chunk Header)，就组成了相应的消息块。消息分块过程如下图，一个大小为307字节的消息被分割成128字节的消息块（除了最后一个）。



消息块格式如下：



Chunk Format

块基本头：共有三种格式，分别占用1byte，2bytes or 3bytes；

fmt：块类型，不同的块类型对应不同的块消息头类型(Chunk Message Header)；

cs id：块流ID，从2开始；

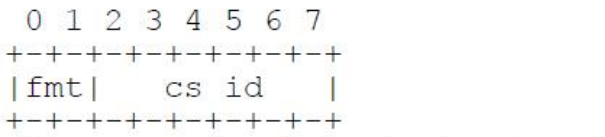


Figure 6 Chunk basic header 1

格式1：cs id占6位，对应值范围 2 - 63；

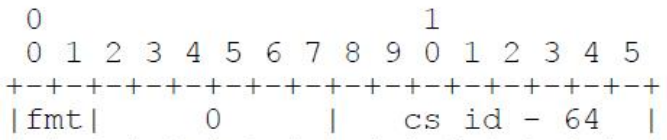


Figure 7 Chunk basic header 2

格式2：cs id占8位，对应值范围 64 - 319；

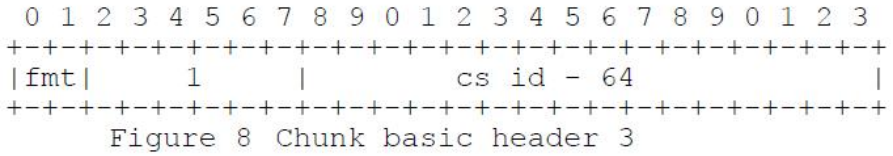


Figure 8 Chunk basic header 3

格式2：cs id占16位，对应值范围 64 - 65599；

块消息头：共有4种格式，长度分别为11、7、3和0个字节；

fmt = 0时，此时chunk message header长11字节，必须出现在块流的起始位置和时间戳出现回滚的位置，结构如下：

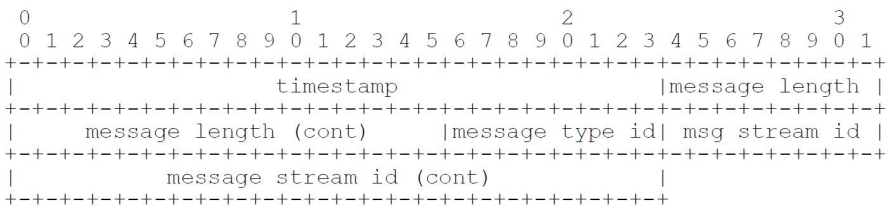


Figure 9 Chunk Message Header - Type 0

tmiestamp：长3字节，存储绝对时间戳；当时间戳值大于等于0xFFFFFF时，该字段填0xFFFFFF，此时扩展时间戳出现在块消息头中，在扩展字段中填绝对时间；

fmt = 1时，此时chunk message header长7字节，对比类型0的chunk message header缺少了msg stream id字段，这一块使用前一块一样的流 ID；可变长度消息的流 (例如，一些视频格式) 应该在第一块之后使用这一格式表示之后的每个新消息；

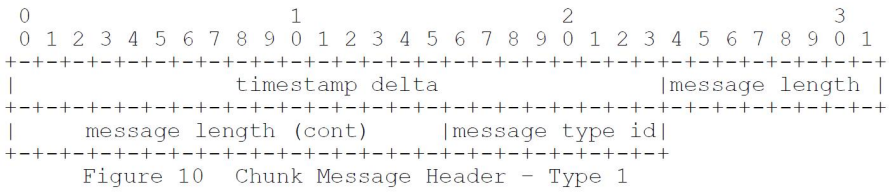


Figure 10 Chunk Message Header - Type 1

fmt = 2时，长度为 3 个字节。既不包含流 ID 也不包含消息长度；这一块具有和前一块相同的流 ID 和消息长度。具有不变长度的消息 (例如，一些音频和数据格式) 应该在第一块之后使用这一格式表示之后的每个新消息。

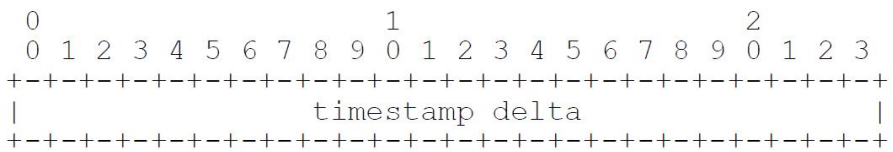


Figure 11 Chunk Message Header - Type 2

fmt = 3时，流 ID、消息长度以及 timestamp delta 等字段都不存在；这种类型的块使用前面块一样的块流 ID、消息长度以及 timestamp delta 等字段。当单一一个消息被分割为多块时，除了第一块的其他块都应该使用这种类型。参考例 2 ，组成流的消息具有同样的大小，流 ID 和时间间隔应该在类型 2 之后的所有块都使用这一类型。参考例 1 ，如果第一个消息和第二个消息之间的 delta 和第一个消息的 timestamp 一样的话，那么在类型 0 的块之后要紧跟一个类型 3 的块，因为无需再来一个类型 2 的块来注册 delta 了。如果一个类型 3 的块跟着一个类型 0 的块，那么这个类型 3 块的 timestamp delta 和类型 0 块的 timestamp 是一样的。

扩展时间戳：

扩展 timestamp 字段用于对大于 16777215 (0xFFFFFF) 的 timestamp 或者 timestamp delta 进行编码；也就是，对于不适合于在 24 位的类型 0、1 和 2 的块里的 timestamp 和 timestamp delta 编码。这一字段包含了整个 32 位的 timestamp 或者 timestamp delta 编码。可以通过设置类型 0 块的 timestamp 字段、类型 1 或者 2 块的 timestamp delta 字段 16777215 (0xFFFFFF) 来启用这一字段。当最近的具有同一块流的类型 0、1 或 2 块指示扩展 timestamp 字段出现时，这一字段才会在类型为 3 的块中出现。

5.3.2.1. 例子 1

这个例子演示了一个简单地音频消息流。这个例子演示了信息的冗余。

	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

下一个表格演示了这个流所产生的块。从消息 3 起，数据传输得到了最佳化利用。每条消息的开销在这一点之后都只有一个字节。

	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. 例子 2

这一例子阐述了一条消息太大，无法装在一个 128 字节的块里，被分割为若干块。

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

Sample Message to be broken to chunks

这是传输的块：

	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

接受端如何解析chunk?

1、先解析第一个字节低6位的值，根据该值获取chunk basic head：

a、低6位的值大于等于2，则该低6位的值为chunk stream id；

b、低6位的值等于0，则接下来的一个字节为chunk stream id；

c、低6位的值等于1，则接下来的两个字节为chunk stream id；

2、根据第一个字节的高两位的值，获取chunk message head的类型：

a、高两位的值为0，则chunk message head长度为11字节；

b、高两位的值为1，则chunk message head长度为7字节；

c、高两位的值为2，则chunk message head长度为3字节；

d、高两位的值为3，则chunk message head长度为0字节；

3、根据chunk message head获取chunk data的长度，解析出数据

==========

http://bbs.chinaffmpeg.com/forum.php?mod=viewthread&tid=236

librtmp推流4.5小时断流问题：填充扩展时间戳