前言

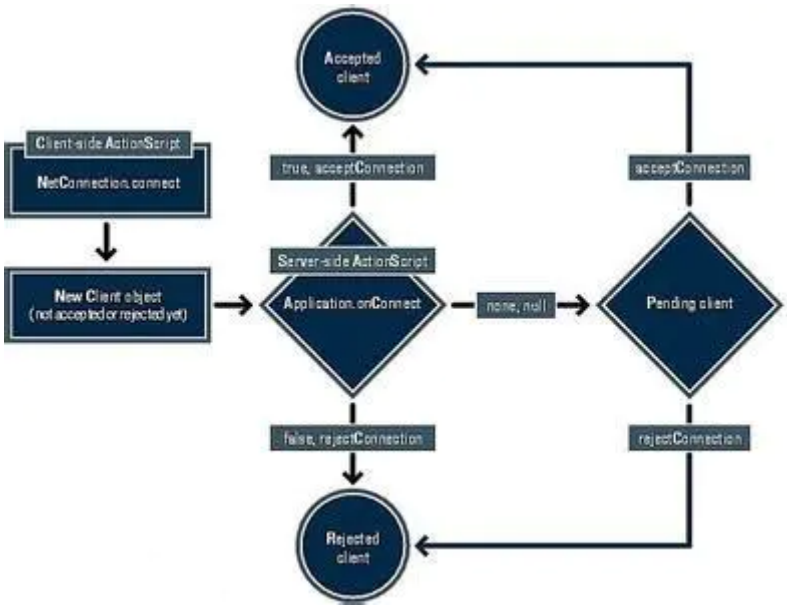
> 大家都知道很多视频应用的app中都是使用RTMP格式的协议，这个是国际上共同使用的协议，我自己虽然做过了直播类型的app，但是从没时间深入的了解这个协议的基础，从这一篇开始让我们逐步揭开RTMP协议的神秘面纱，从应用层逐步进入原理层和底层。

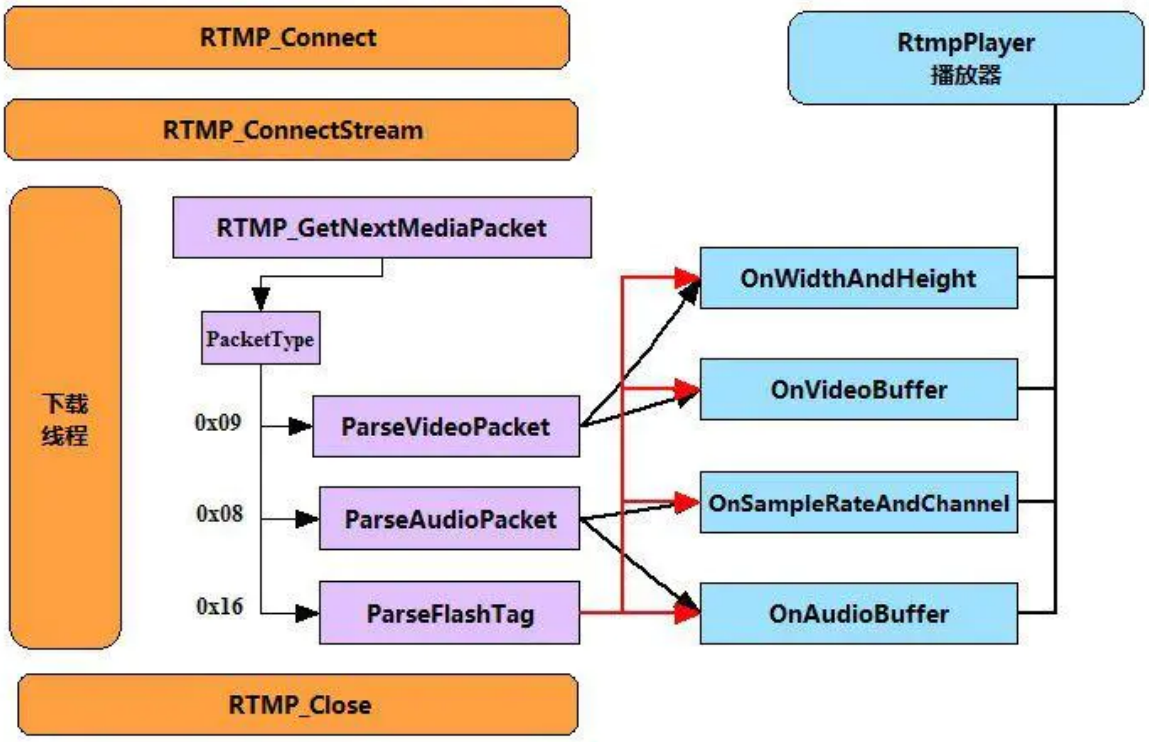
定义

RTMP是`Real Time Messaging Protocol`（实时消息传输协议）的首字母缩写。该协议基于TCP，是一个协议族，包括RTMP基本协议及RTMPT/RTMPS/RTMPE等多种变种。RTMP是一种设计用来进行实时数据通信的网络协议，主要用来在Flash/AIR平台和支持RTMP协议的流媒体/交互服务器之间进行音视频和数据通信。支持该协议的软件包括`Adobe Media Server/Ultrant Media Server/red5`等。

RTMP又是`Routing Table Maintenance Protocol`（路由选择表维护协议）的缩写。 在 AppleTalk 协议组中，路由选择表维护协议（`RTMP，Routing Table Protocol`）是一种传输层协议，它在 AppleTalk 路由器中建立并维护路由选择表。RTMP 基于路由选择信息协议（RIP）。正如 RIP 一样，RTMP 使用跳数作为路由计量标准。一个数据包从源 网络发送到目标网络，必须通过的路由器或其它中间介质节点数目的计算结果即为跳数。

下面我们看一下两张原理图理解一下。





协议概述

`RTMP(Real Time Messaging Protocol)`实时消息传送协议是Adobe Systems公司为Flash播放器和服务器之间音频、视频和数据传输 开发的开放协议。
它有多种变种：

- `RTMP`工作在TCP之上，默认使用端口1935；
- `RTMPE`在RTMP的基础上增加了加密功能；
- `RTMPT`封装在HTTP请求之上，可穿透防火墙；
- `RTMPS`类似RTMPT，增加了`TLS/SSL`的安全功能。

协议详细介绍

`RTMP`协议`(Real Time Messaging Protocol)`是被Flash用于对象，视频，音频的传输。这个协议建立在TCP协议或者轮询HTTP协议之上。

RTMP协议就像一个用来装数据包的容器,这些数据既可以是`AMF`格式的数据，也可以是`FLV`中的视/音频数据。

一个单一的连接可以通过不同的通道传输多路网络流，这些通道中的包都是按照固定大小的包传输的。
网络连接`(Connection)`一个`Actionscript`连接并播放一个流的简单代码:

```
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");
```

默认端口为`1935`

握手请求及应答

1. 握手过程

`Client → Server`：向服务器发出握手请求.这不属于协议包一部分,该握手请求第一个字节为(0×03),其后跟着1536个字节。尽管看上去这部分的内容对于RTMP协议来说并不是至关重要的,但也不可随意对待。

`Server → Client`：服务器向客户端回应握手请求，这部分的数据仍然不属于RTMP协议的部分。该回应的起始字节仍然为(0x03)，但是后边跟着两个长度为1536个字节(一共为3072字节)的包块。第一个1536块看上去似乎可以是任意内容，甚至好像可以是Null都没有关系。第二个1536的代码块，是上一步客户端向服务器端发送的握手请求的内容。

`Client→Server`：把上一步服务器向客户端回应的第二块1536个字节的数据块。

至此客户端与服务器端的握手结束，下面将发送RTMP协议的包内容。

`Client → Server`：向服务器发送连接包。
`Server → Client`：服务器回应。

... . 等等... ...

2. RTMP数据类型

```
0x01 Chunk Size changes the chunk size for packets
0x02 Unknown anyone know this one?
0x03 Bytes Read send every x bytes read by both sides
0x04 Ping ping is a stream control message, has subtypes
0x05 Server BW the servers downstream bw
0x06 Client BW the clients upstream bw
0x07 Unknown anyone know this one?
0x08 Audio Data packet containing audio
0x09 Video Data packet containing video data
0x0A - 0x11 Unknown anyone know?
0x12 Notify an invoke which does not expect a reply
0x13 Shared Object has subtypes
0x14 Invoke like remoting call, used for stream actions too.
Shared Object 数据类型
0x01 Connect
0x02 Disconnect
0x03 Set Attribute
0x04 Update Data
0x05 Update Attribute
0x06 Send Message
0x07 Status
0x08 Clear Data
0x09 Delete Data
0x0A Delete Attribute
0x0B
Initial Data
```

3. RTMP包结构

RTMP包 包含一个固定长度的包头和一个最长为128字节的包体，包头可以是下面4种长度的任意一种：`12, 8, 4, or 1 byte(s)`。
第一个字节的前两个Bit很重要，它决定了包头的长度，它可以用掩码0xC0进行"与"计算。下面罗列了可能的包头长度`Bits Header Length`。

```
00 12 bytes
01 8 bytes
10 4 bytes
11 1 byte
```

其实`RTMP`包结构就是使用了`AMF`格式.

下面是一个关于客户端向服务器端发送流的流程：

- `Client → Server`:发送一个创建流的请求
- `Server → Client`:返回一个表示流的索引号
- `Client → Server`:开始发送
- `Client → Server`:发送视音频数据包(这些包在同一个频道(channel)并用流的索引号来唯一标识)

4. RTMP Chunk Stream - RTMP块流

`Chunk Stream`是对传输`RTMP Chunk`的流的逻辑上的抽象，客户端和服务器之间有关RTMP的信息都在这个流上通信。这个流上的操作也是我们关注RTMP协议的重点

Message

`Message`是指满足该协议格式的、可以切分成`Chunk`发送的消息，消息包含的字段如下所示。

- `Timestamp`（时间戳）：消息的时间戳（但不一定是当前时间，后面会介绍），4个字节。
- `Length`(长度)：是指`Message Payload`（消息负载）即音视频等信息的数据的长度，3个字节。
- `TypeId`(类型Id)：消息的类型Id，1个字节。
- `Message Stream ID`（消息的流ID）：每个消息的唯一标识，划分成`Chunk`和还原Chunk为`Message`的时候都是根据这个ID来辨识是否是同一个消息的`Chunk`的，4个字节，并且以小端格式存储。

Chunking(Message分块)

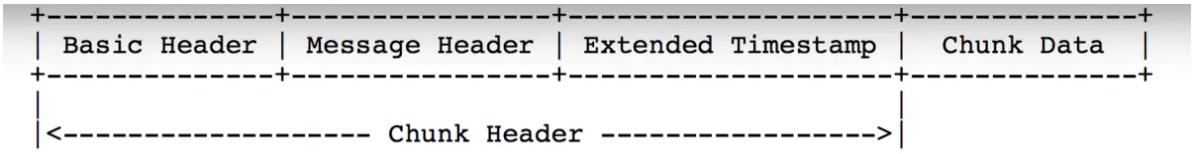
`RTMP`在收发数据的时候并不是以`Message`为单位的，而是把Message拆分成Chunk发送，而且必须在一个`Chunk`发送完成之后才能开始发送下一个Chunk。每个Chunk中带有MessageID代表属于哪个Message，接受端也会按照这个id来将chunk组装成Message。

为什么RTMP要将Message拆分成不同的Chunk呢？通过拆分，数据量较大的Message可以被拆分成较小的“Message”，这样就可以避免优先级低的消息持续发送阻塞优先级高的数据，比如在视频的传输过程中，会包括视频帧，音频帧和RTMP控制信息，如果持续发送音频数据或者控制数据的话可能就会造成视频帧的阻塞，然后就会造成看视频时最烦人的卡顿现象。同时对于数据量较小的Message，可以通过对Chunk Header的字段来压缩信息，从而减少信息的传输量。

Chunk的默认大小是128字节，在传输过程中，通过一个叫做Set Chunk Size的控制信息可以设置Chunk数据量的最大值，在发送端和接受端会各自维护一个Chunk Size，可以分别设置这个值来改变自己这一方发送的Chunk的最大大小。大一点的Chunk减少了计算每个chunk的时间从而减少了CPU的占用率，但是它会占用更多的时间在发送上，尤其是在低带宽的网络情况下，很可能会阻塞后面更重要信息的传输。小一点的Chunk可以减少这种阻塞问题，但小的Chunk会引入过多额外的信息（Chunk中的

Header），少量多次的传输也可能会造成发送的间断导致不能充分利用高带宽的优势，因此并不适合在高比特率的流中传输。在实际发送时应对要发送的数据用不同的Chunk Size去尝试，通过抓包分析等手段得出合适的Chunk大小，并且在传输过程中可以根据当前的带宽信息和实际信息的大小动态调整Chunk的大小，从而尽量提高CPU的利用率并减少信息的阻塞机率。

Chunk Format - Chunk格式



Chunk Format

下面就说一下快格式里面的组成。

- Basic Header：它是基本的头信息。

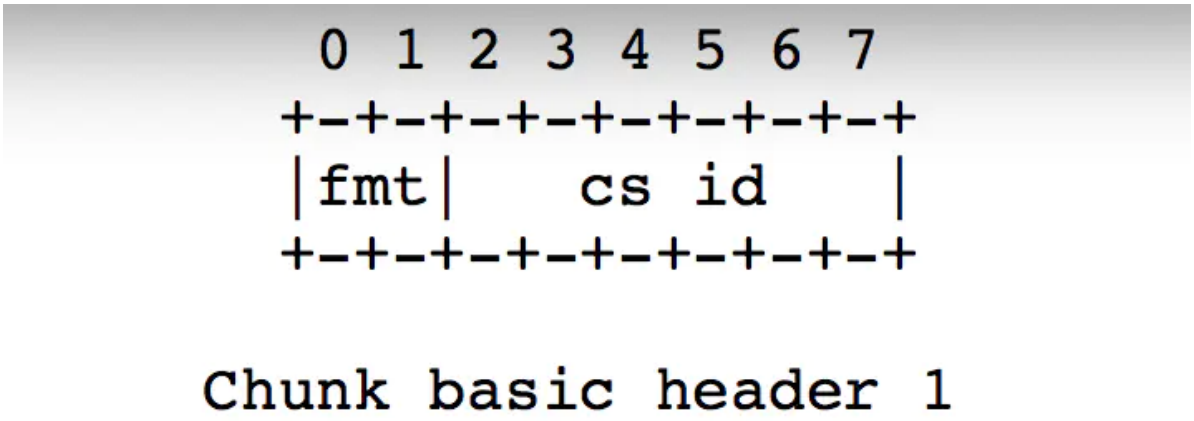
包含了`chunk stream ID`（流通道Id）和`chunk type`（chunk的类型），`chunk stream id`一般被简写为`CSID`，用来唯一标识一个特定的流通道，chunk type决定了后面Message Header的格式。Basic Header的长度可能是1，2，或3个字节，其中chunk type的长度是固定的（占2位，注意单位是位，bit），Basic Header的长度取决于CSID的大小，在足够存储这两个字段的前提下最好用尽量少的字节从而减少由于引入Header增加的数据量。

RTMP协议支持用户自定义［3，65599］之间的CSID，0，1，2由协议保留表示特殊信息。

0代表Basic Header总共要占用2个字节，CSID在［64，319］之间；
1代表占用3个字节，CSID在［64，65599］之间；
2代表该chunk是控制信息和一些命令信息，后面会有详细的介绍。

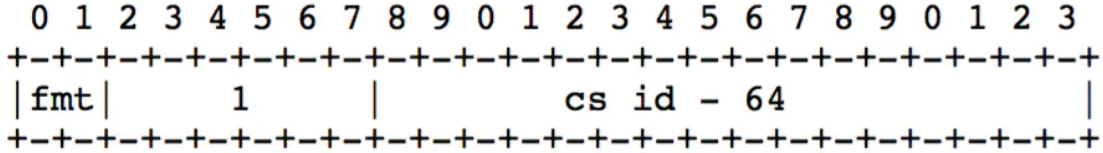
chunk type的长度固定为2位，因此CSID的长度是（6=8-2）、（14=16-2）、（22=24-2）中的一个。
当Basic Header为1个字节时，CSID占6位，6位最多可以表示64个数，因此这种情况下CSID在［0，63］之间，其中用户可自定义的范围为［3，63］。

下面看一下Basic Header不同字节时的字节示意图。



Chunk basic header 1

Basic Header为1个字节时



Chunk basic header 3

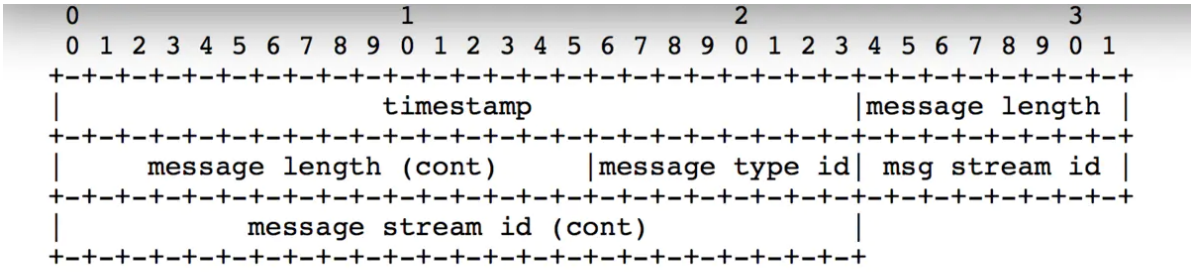
Basic Header为2或3个字节时

需要注意的是，Basic Header是采用小端存储的方式，越往后的字节数量级越高。可以看到2个字节和3个字节的Basic Header所能表示的CSID是有交集的`[64，319]`，但实际实现时还是应该秉着最少字节的原则使用2个字节的表示方式来表示`[64，319]`的`CSID`。

- Message Header

包含了要发送的实际信息（可能是完整的，也可能是一部分）的描述信息。Message Header的格式和长度取决于`Basic Header`的`chunk type`，共有4种不同的格式，由上面所提到的Basic Header中的fmt字段控制。其中第一种格式可以表示其他三种表示的所有数据，但由于其他三种格式是基于对之前chunk的差量化的表示，因此可以更简洁地表示相同的数据，实际使用的时候还是应该采用尽量少的字节表示相同意义的数据。以下按照字节数从多到少的顺序分别介绍这4种格式的`Message Header`。

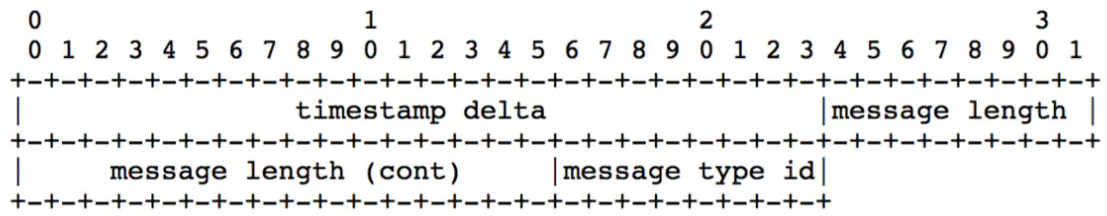
(1) type=0时Message Header占用11个字节，其他三种能表示的数据它都能表示，但在chunk stream的开始的第一个chunk和头信息中的时间戳后退（即值与上一个chunk相比减小，通常在回退播放的时候会出现这种情况）的时候必须采用这种格式。



Chunk Message Header - Type 0

Message Header type 0

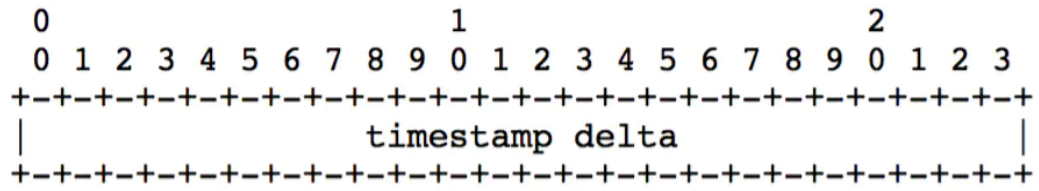
(2) type=1时Message Header占用7个字节，省去了表示msg stream id的4个字节，表示此chunk和上一次发的chunk所在的流相同，如果在发送端只和对端有一个流链接的时候可以尽量去采取这种格式。



Chunk Message Header - Type 1

Message Header type 1

(3) type=2时Message Header占用3个字节，相对于type = 1格式又省去了表示消息长度的3个字节和表示消息类型的1个字节，表示此chunk和上一次发送的chunk所在的流、消息的长度和消息的类型都相同。余下的这三个字节表示timestamp delta，使用同type = 1 。



Chunk Message Header - Type 2

Message Header type 2

(4) 0字节！！！好吧，它表示这个chunk的Message Header和上一个是完全相同的，自然就不用再传输一遍了。当它跟在Type = 0的chunk后面时，表示和前一个chunk的时间戳都是相同的。什么时候连时间戳都相同呢？就是一个Message拆分成了多个chunk，这个chunk和上一个chunk同属于一个Message。而当它跟在Type = 1或者Type = 2的chunk后面时，表示和前一个chunk的时间戳的差是相同的。比如第一个chunk的Type = 0，timestamp = 100，第二个chunk的Type = 2，timestamp delta = 20，表示时间戳为100+20=120，第三个chunk的Type = 3，表示timestamp delta = 20，时间戳为120+20=140。

- Extended Timestamp（扩展时间戳）

上面我们提到在chunk中会有时间戳timestamp和时间戳差timestamp delta，并且它们不会同时存在，只有这两者之一大于3个字节能表示的最大数值0xFFFFFF = 16777215时，才会用这个字段来表示真正的时间戳，否则这个字段为0。扩展时间戳占4个字节，能表示的最大数值就是0xFFFFFFFF = 4294967295。当扩展时间戳启用时，timestamp字段或者timestamp delta要全置为1，表示应该去扩展时间戳字段来提取真正的时间戳或者时间戳差。注意扩展时间戳存储的是完整值，而不是减去时间戳或者时间戳差的值。

- Chunk Data（块数据）：

用户层面上真正想要发送的与协议无关的数据，长度在`[0,chunkSize]`之间。

协议控制消息（Protocol Control Message）

在RTMP的chunk流会用一些特殊的值来代表协议的控制消息，它们的Message Stream ID必须为0（代表控制流信息），CSID必须为2，Message Type ID可以为1，2，3，5，6，具体代表的消息会在下面依次说明。控制消息的接受端会忽略掉chunk中的时间戳，收到后立即生效。

- `Set Chunk Size(Message Type ID=1)`：设置chunk中Data字段所能承载的最大字节数，默认为128B，通信过程中可以通过发送该消息来设置`chunk Size`的大小（不得小于128B），而且通信双方会各自维护一个chunkSize，两端的chunkSize是独立的。
- `Abort Message(Message Type ID=2)`：当一个Message被切分为多个chunk，接受端只接收到了部分chunk时，发送该控制消息表示发送端不再传输同Message的chunk，接受端接收到这个消息后要丢弃这些不完整的chunk。Data数据中只需要一个CSID，表示丢弃该CSID的所有已接收到的chunk。
- `Acknowledgement(Message Type ID=3)`：当收到对端的消息大小等于窗口大小（Window Size）时接受端要回馈一个ACK给发送端告知对方可以继续发送数据。窗口大小就是指收到接受端返回的ACK前最多可以发送的字节数量，返回的ACK中会带有从发送上一个ACK后接收到的字节数。
- `Window Acknowledgement Size(Message Type ID=5)`：发送端在接收到接受端返回的两个ACK间最多可以发送的字节数。
- `Set Peer Bandwidth(Message Type ID=6)`：限制对端的输出带宽。接受端接收到该消息后会通过设置消息中的Window ACK Size来限制已发送但未接受到反馈的消息的大小来限制发送端的发送带宽。如果消息中的Window ACK Size与上一次发送给发送端的size不同的话要回馈一个`Window Acknowledgement Size`的控制消息。

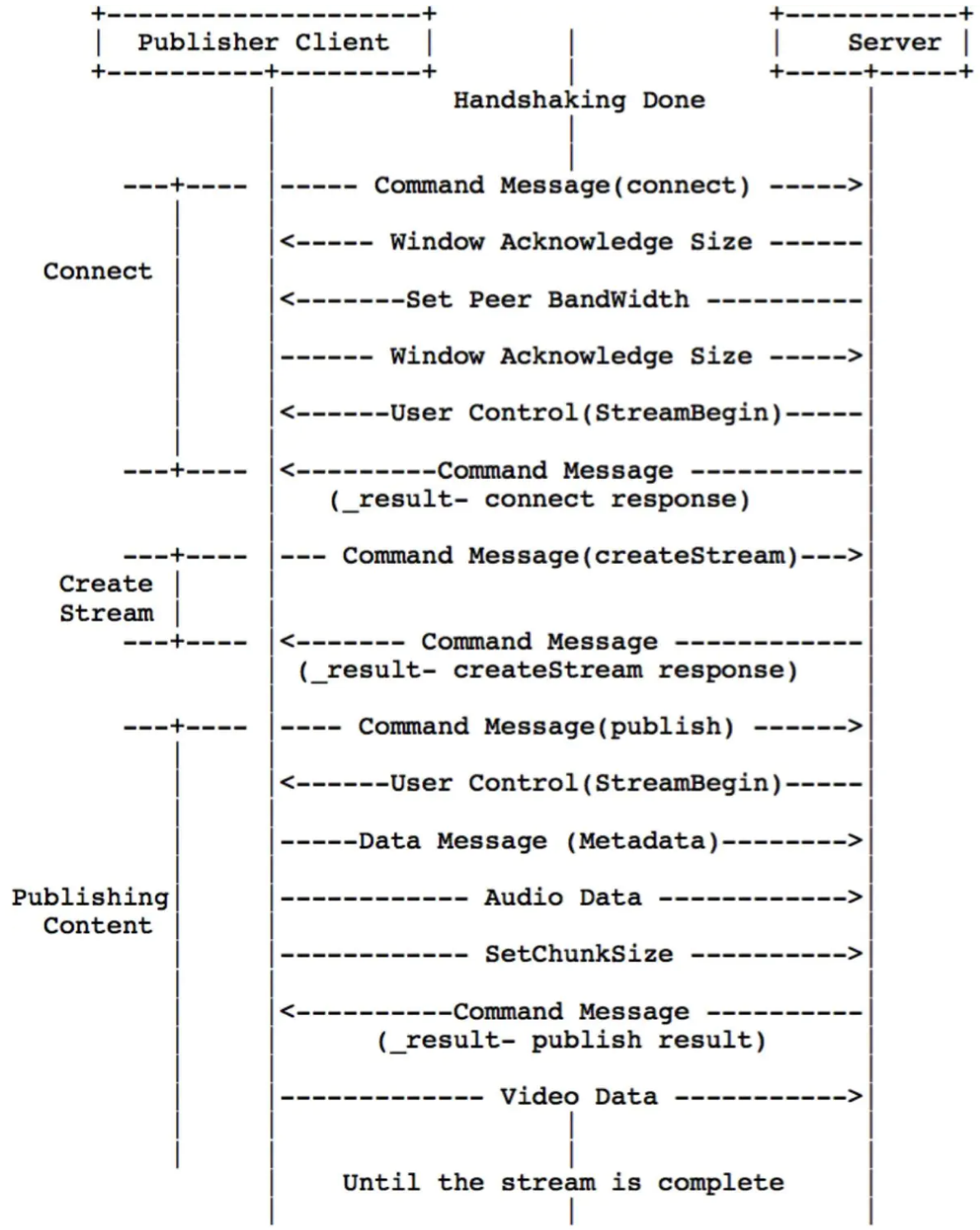
(1)`Hard(Limit Type=0)`：接受端应该将`Window Ack Size`设置为消息中的值
(2)`Soft(Limit Type=1)`：接受端可以讲`Window Ack Size`设为消息中的值，也可以保存原来的值（前提是原来的Size小与该控制消息中的`Window Ack Size`）
(3)`Dynamic(Limit Type=2)`：如果上次的`Set Peer Bandwidth`消息中的Limit Type为0，本次也按Hard处理，否则忽略本消息，不去设置`Window Ack Size`。

5. 不同类型的RTMP Message

- `Command Message`(命令消息，`Message Type ID=17或20`)：表示在客户端和服务器间传递的在对端执行某些操作的命令消息，如connect表示连接对端，对端如果同意连接的话会记录发送端信息并返回连接成功消息，publish表示开始向对方推流，接受端接到命令后准备好接受对端发送的流信息，后面会对比较常见的Command Message具体介绍。当信息使用AMF0编码时，Message Type ID = 20，AMF3编码时Message Type ID = 17。
- `Data Message`（数据消息，`Message Type ID=15或18`）：传递一些元数据（MetaData，比如视频名，分辨率等等）或者用户自定义的一些消息。当信息使用AMF0编码时，Message Type ID = 18，AMF3编码时Message Type ID = 15。
- `Shared Object Message`(共享消息，Message Type ID = 16或19)：表示一个Flash类型的对象，由键值对的集合组成，用于多客户端，多实例时使用。当信息使用AMF0编码时，`Message Type ID=19，AMF3` 编码时 `Message Type ID=16`。
- `Audio Message`（音频信息，`Message Type ID=8`）：音频数据。
- `Video Message`（视频信息，`Message Type ID=9`）：视频数据。
- `Aggregate Message`(聚集信息，`Message Type ID=22`)：多个RTMP子消息的集合 。
- `User Control Message Events`(用户控制消息，Message Type ID=4):告知对方执行该信息中包含的用户控制事件，比如Stream Begin事件告知对方流信息开始传输。和前面提到的协议控制信息（Protocol Control Message）不同，这是在RTMP协议层的，而不是在RTMP chunk流协议层的，这个很容易弄混。该信息在chunk流中发送时，`Message Stream ID=0,Chunk Stream Id=2,Message Type Id=4`。

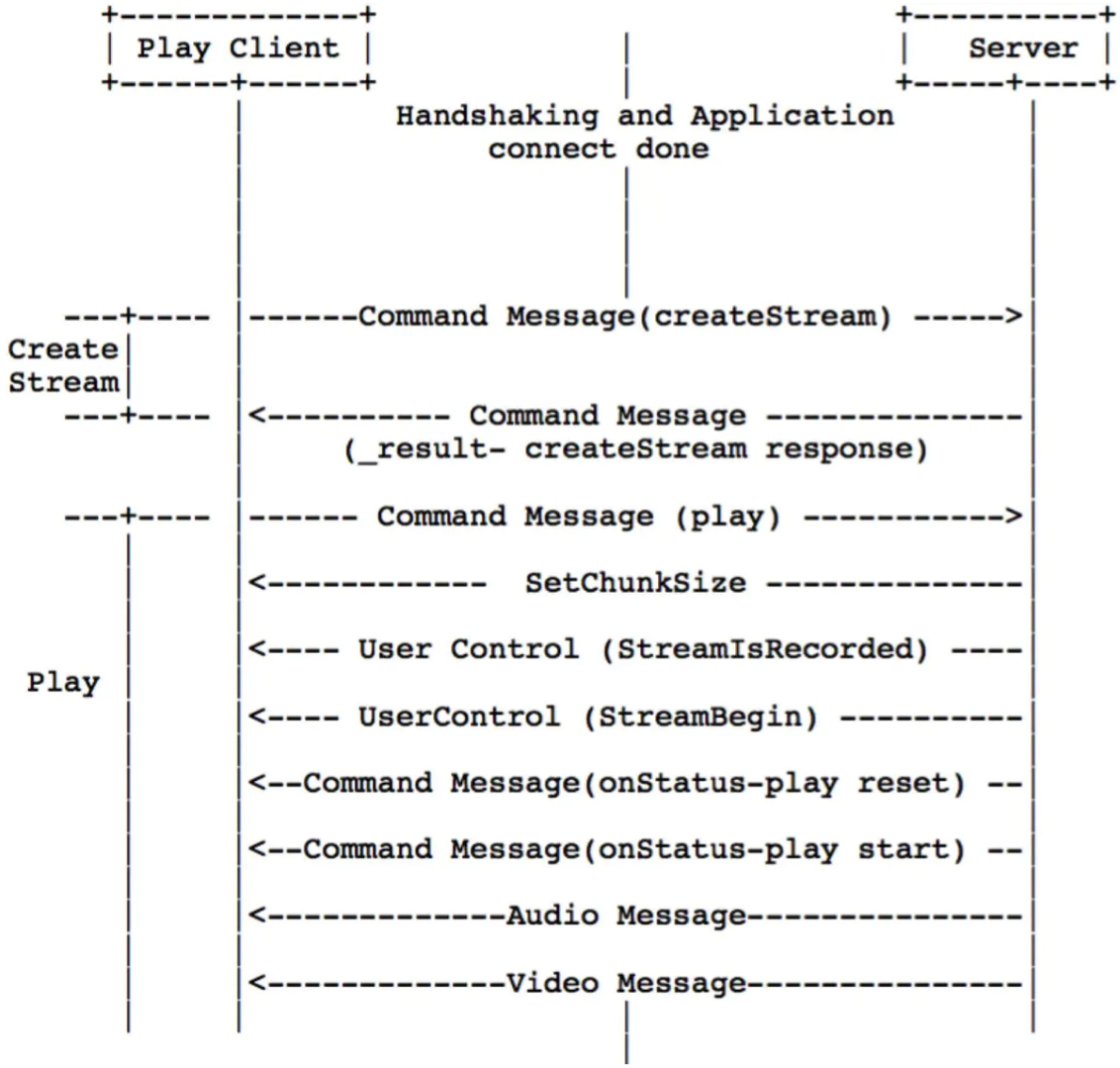
6. 基于RTMP协议的推流和拉流播放过程

先看一下推流过程。



推流过程

再看一下拉流过程



拉流播放过程