基于 RTMP 协议的直播摄像头

目　录

修订信息：

日期	备注
2015/10/1	摄像头推流技术概要
2015/11/20	音频内容的支持与注意事项

背　景

流媒体技术的主要特点是以 “流(Streaming)” 的形式在基于 IP 协议的互联网中进行多媒体数据的实时、连续传播,客户端在播放前并不需要下载 整个媒体文件,而是在将缓存区中已经收到的媒体数据进行播放。同时, 媒体流的剩余部分仍持续不断地从服务器递送到客户端,即所谓的“边下 载,边播放”。 移动互联网是传统桌面互联网向移动通信网络的延伸。作为移动互联网中具有代表性的典型应用,移动流媒体业务主要是利用互联网和3G、4G通信网络平台,为以手机为主的嵌入式终端设备提供基于音视频的流式多媒体服务。

流媒体协议是支撑流媒体业务运行的关键核心技术之一。在传统桌面互联网时代,常用的流媒体协议主要有 HTTP 渐进式下载和基于 RTSP/RTP 的实时流媒体协议栈等等,这些流媒体协议大多数可以平移到移动流媒体 中继续应用。然而由于移动互联网及其终端设备的一些独有特性,传统流 媒体协议在移动互联网中的应用在功能、性能的提供和用户体验等方面都 会受到不同程度的约束和限制,于是一些新的流媒体协议应运而生。例如, 苹果公司的 HTTP Live Streaming 就是其中具有代表性且得到较为广泛应 用的一个。

本篇内容主要针对上述几种流媒体协议进行综述,并对这几种协议的优缺点以及适用范围进行较为深入的分析和比较,以辅助工程人员对流媒体实现的技术选型有较为全面的认知。

1 摄像头平台

以下为 3 家主流摄像头平台解决方案厂商的介绍：

TI 是美国德州仪器的简称，总部位于美国德克萨斯州的达拉斯，是全球知名的半导体企业，主要从事模拟电路和数字信号处理技术的研究，其具有代表性的 DaVinci- DM3x ARM9 视频处理器解决方案在安防行业有着广泛的应用。TI 成立之初是一家使用地震信号处理技术勘探原油的地质勘探公司，1951 年更名为现用名的德州仪器公司，其后逐渐进入半导体市场。当安防领域的视频监控从模拟阶段发展到数字压缩处理阶段后，TI 在安防视频压缩领域逐渐暂居主导地位。高清视频监控领域有代表性的解决方案是 DM355、DM365、DM368。

海思(Hisilicon)，海思半导体有限公司成立于 2004 年 10 月，前身是创建于 1991 年的华为集成电路设计中心，总部位于深圳。 海思的产品覆盖无线网络、固定网络、数字媒体等领域的芯片及解决方案，成功应用在全球 100 多个国家和地区;在数字媒体领域，已推出网络监控芯片及解决方案、可视电话芯片及解决方案、DVB 芯片及解决方案和 IPTV 芯片及解决方案。在 2009-2012 年，DVR 芯片可谓风生水起。在 IPC 领域有代表性的解决方案是 Hi3516、Hi3517、Hi3518A、Hi3518C、Hi3518E。

安霸(Ambarella)于 2004 年由 C-Cube Microsystems 公司三位元老组建，总部位于加州的圣克拉拉市,公司中文名称为“安霸”。安霸是高清视频业界的技术领导者，主要提供低功耗、高清视频压缩与图像处理的解决方案。在电视广播和广电行业市场，安霸技术也得到广泛应用，在 H.264 高清专业广播编码设备市场拥有近 90%的市场，大量来自世界各地的电视节目都经安霸芯片压缩后传送。在消费摄像机市场，安霸推动开创出一块多功能摄像机新领域。众多世界顶尖品牌的数码相机、DV、多功能运动摄像机品牌包括 Kodak, SONY, Samsung，GoPro 等已经正在使用安霸芯片。在安防监控市场，安霸利用超高画质视频的领先技术，推动高清监控摄像机在安防的发展。安霸在业界率先推出了基于最新 H.264 视频压缩标准的高集成 SoC 芯片，集成了各种关键系统功能，提供高性价比的高清整体解决方案。安霸最初将 A2 系列引入到安防视频监控领域，发展到成熟应用的 A5s 系列，目前已发展到 S2 系列 UHD 超高清安防 IPC 解决方案。由于技术难度较高，在国内能够从 A2、A5 到 S2 系列都有深入开发，有深厚技术积累，并能规模转化为实用的高清视频监控产品的公司屈指可数。

1.1 硬件平台解决方案

音视频的编码算法在本文档不展开，本文主要描述软硬件规格，以及如何利用七牛流媒体服务进行推流。

以下是主流 3 个平台的 IPcam 硬件配置情况：

方案类型	安霸A5s	TI DM368	海思Hi3516
方案架构	ARM+视频压缩单元	ARM+视频压缩单元	
处理器	ARM1136J-S	ARM926EJ-S	ARM Cortex A9
主频	528 MHz	432MHz	800MHz
制造工艺	45纳米	65纳米	未知
编码算法	H.264 BP/MP/HP Level 5.0 MJPEG	H.264 BP/MP/HP MPEG-4,MPEG-2,MJPEG,VC1/WMV9	H.264 BP/MP/HP MPEG4,MJPEG/JPEG
最高压缩分辨率	10MP@3fps,5MP@12fps,3MP@20fps 1080P@30fps + D1@30fps + CVBS/HDMI	1080p,30fps	1080P@30fps + D1@30fps
码流控制	1080P/30fps≤4Mbps,CBR/VBR/ConstantQP rate control	1080P/30fps≤6Mbps	1080P/30fps≤6Mbps
接口	主流接口都支持	主流接口都支持	主流接口都支持
3A控制	广播级3A	支持	支持
功耗	1080P模式下≤1W	未知	未知

1.1.1 视频编码器

对于七牛的流媒体服务器，目前支持 H.264 编码，并同时支持 baseline, main, high profile,不过对于主流的摄像头输出，多数为 baseline 或者 main profile 并且少有 B 帧编码（参看 https://en.wikipedia.org/wiki/Video_compression_picture_types 对于 B 帧简介），B 帧对于编码效率有极大的帮助但同时也需要更多的编码处理时间，因此对于多数的摄像头平台的编码器，虽然同时支持三种 profile 但是在摄像头实际的开发过程中，考虑到嵌入式平台的硬件限制，因此采用的是 baseline/main 的基本功能。当然，如果能够采用 high profile 的编码算法将极大提高同等带宽条件下的视频质量，但是对于流媒体服务器以及协议层的封装则需要相应作出改动以兼容相应编码算法。

1.1.2 音频编码器

七牛流媒体服务器支持 AAC 音频编码，能够将 AVC 与 AAC 转存为不同的封装格式，对于多数摄像头，会采用 PCM 系的音频编码，由于 PCM 的音频需要进行一次转码处理，当流媒体服务器在对编码进行封装时将需要统一转码为 H.264/AAC 的编码来做进一步的多媒体处理。对于 PCM 的音频编码，如果能够正确地在协议层通过 RTMP 进行封包，则在播放端

可以正常播放。

1.1.3 处理器 / 内存

处理器规格要求：ARM9 以上

内存要求：最低保证 32MB 以上

摄像设备一般是嵌入式平台为主，在内存方面需要保证有足够地空间来进行协议层的工作，比如可能的发包缓冲队列以及 RTMP 协议本身的二进制库。对于 CPU 的要求则不需要有太高要求，原则是对于性能不高的配置情况下，尽量能够将数据处理的部分外移，如转码、压缩、格式封装等，能够满足编码器(codec)至协议层(RTMP)的实现即可，过多的处理会导致很多不可预料的错误，如延时，音视频不同步，发包性能下降导致的画面卡顿等。

1.2 软件平台解决方案

软件平台是指提供摄像头工作的软件环境以及开发工具链以及第三方的库文件。

1.2.1 操作系统

操作系统为经过剪裁的嵌入式 32 位 Linux 系统，一般对于海思、安霸等平台有多种可选的开放操作系统。对于海思 3516、3518 等平台都有提供相应的开发组件，因此对于厂商提供的解决方案，都可以直接进行开发。

1.2.2 编译环境

我们提供了两种解决方案，但都是基于开源的 RTMP 协议库 librtmp，因此只要能够在相应平台提供的 toolchain 编译链接通过则可以保证程序正常运行。

编译环境操作演示：

安装完毕后的开发环境：

```
gnueabi/lib$ ls
a7                      libasan.la          libgomp.so
a7_hard_neon-vfpv4      libasan_preinit.o   libgomp.so.1
a7_soft                 libasan.so          libgomp.so.1.0.0
a7_softfp_neon-vfpv4    libasan.so.0        libgomp.spec
a7_softfp_vfpv4         libasan.so.0.0.0    libstdc++.a
a9                      libatomic.a         libstdc++.la
a9_soft                 libatomic.la        libstdc++.so
a9_softfp_vfp           libatomic.so        libstdc++.so.6
a9_softfp_vfpv3-d16     libatomic.so.1      libstdc++.so.6.0.19
armv5te_arm9            libatomic.so.1.0.0  libstdc++.so.6.0.19-gdb.py
armv5te_arm9_soft       libgcc_s.so         libsupc++.a
armv5te_arm9_vfp        libgcc_s.so.1       libsupc++.la
ldscripts               libgomp.a
libasan.a               libgomp.la
```

注意在项目的 build 过程中如果发生缺少运行库的问题的时候，比如 stdc 等标准库的问题，请先确认当前交叉编译环境的操作系统是 64 位还是 32 位，如果是 64 位系统，则需要安装相应标准库的 32 位版本，比如：

```
sudo apt-get install lib32stdc++6
```

另外对于编译时找不到第三方库的情况，请检查/etc/ld.so.conf 配置，确保相应库的链接目录存在于系列配置中。

1.2.3 关联库

对于编码层，无需使用软件实现如 x264，各相应的解决方案平台就已经能够实现硬件编码，因此与七牛流媒体对接所需要的只是协议层的实现，目前我们提供 2 种不同的开放接口方式：

- 基于 libRTMP 实现的 demo 程序
- 基于 libRTMP 封装的七牛 C-SDK

如果开发条件允许，我们建议直接基于 libRTMP 进行开发对后期的维护工作更有利，如何进行协议的封装与开发，可以参考后一章的内容。

2 协议层

从音视频栈的组成来看可分为 3 层，编码层、封装层和协议层，实际的数据交互则封装在网络协议栈中，其中 RTMP 协议属于应用层协议，基于 TCP/IP。

本章将会对 RTMP 协议以及 FLV 封装格式进行简介。

2.1 RTMP 协议简介

RTMP 协议 (Real Time Messaging Protocol)最初为 Macromedia 公司开发的互联网音视频传输协议，主要用于在 Flash 服务端与播放端的通信，目前该协议属于 Adobe 公司并发布了一份并不完整的公开文档。

2.2 FLV 文件分析

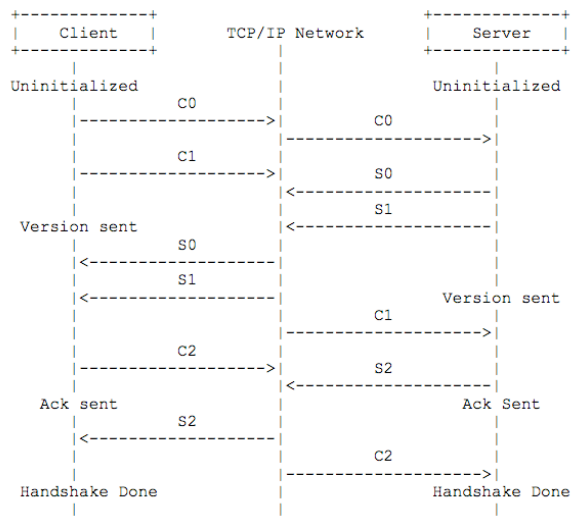
FLV 的文件的格式可参考 Adobe 官方发布的”Adobe flash video file format specification version 10.1”， 由于 RTMP 的应用层数据报文的内容与 FLV 各类 tag 的格式一致，比如 Video Tags 或 Audio Tags。因此，分析 FLV 文件的的格式对于了解 RTMP 协议有很大帮助。

2.3 RTMP 推流协议分析

RTMP 协议是双向协议，意味着服务端－客户端的数据流向是双向的。

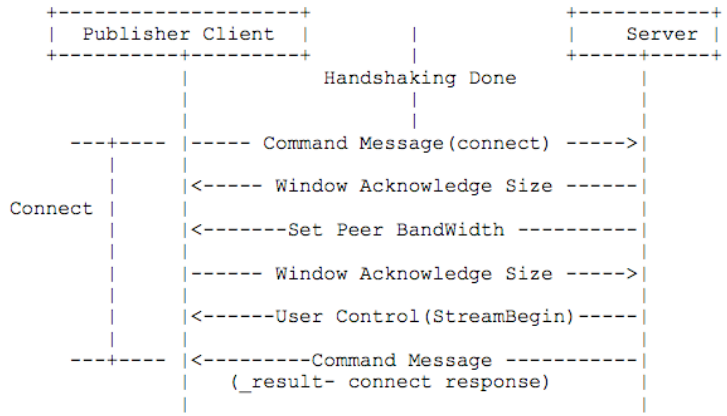
2.3.1 连接发起

握手阶段：

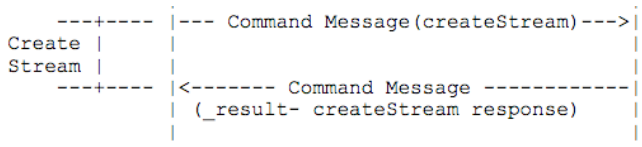


C0,C1 为客户端发起的定长请求，需等到 S0，S1 回复后才能继续发送最后的 C2 定长请求，之后等待 S2 返回完成握手。

建立连接阶段：



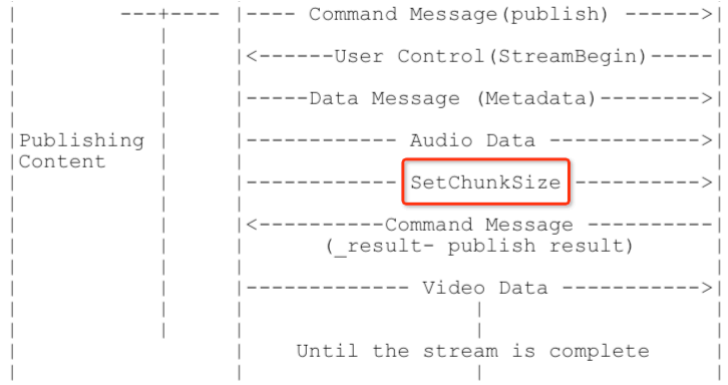
创建流：



这部分是握手之后确认传输控制的流程，细节可在 adobe 的文档中找到，虽然整篇文档的细节部分并不多，建议可参考 libRTMP 或 ffmpeg 源码 connect 部分。

2.3.2 协议控制

关于调整 ChunkSize，文档定义如下：



根据”RTMP specification 1.0”文档描述，设置 chunk 大小需要单独发送控制包告知服务端，之后将以协商的 chunk 大小进行传输。

2.3.3 数据交互

发布流：

发布流主要是实际音频与视频的分发。可看到数据主要通过 Audio data 与 Video data 的数据包进行分发，音视频在发送第一个数据包时需要发送 Sequence Header。在实际的操作过程中，发包的时机需要自己控制，也就是说明 Frame Rate 的控制是由推流客户端

进行。

可以简单地分析一下数据交互的过程：

2.3.3.1 音频

AAC sequence header:

```
Real Time Messaging Protocol (Audio Data)
  RTMP Header
      00.. .... = Format: 0
      ..00 0100 = Chunk Stream ID: 4
      Timestamp: 0
      Body size: 4
      Type ID: Audio Data (0x08)
      Stream ID: 1
  RTMP Body
    Control: 0xaf (HE-AAC 44 kHz 16 bit stereo)
        1010 .... = Format: HE-AAC (10)
        .... 11.. = Sample rate: 44 kHz (3)
        .... ..1. = Sample size: 16 bit (1)
        .... ...1 = Channels: stereo (1)
      Audio data: 001210
```

音频需要首先发送 sequence header 来告知服务器相关的音频配置。

Sequence header 分为两部分:

Decoder Specific 和 Audio Specific Configuration(ASC)，其中需要将 AAC ADTS 头部的字段解析至 ASC 中，要注意其中：

E 2 profile, the MPEG-4 Audio Object Type minus 1

profile 指代的 Audio ObjectType 字段在 ADTS 中为定义的常量值减去 1，因此解析至 ASC 时需要相应加上 1。

而 sequence header 之后的一般 AAC 数据帧都会带有 ADTS 头部，当推送时，使用的推送工具比如 ffmpeg 警告称需要进行 adts_to_asc 转换，意味着推流的数据包将 ADTS 头部包含至 RTMP packet 中以至于报错。

2.3.3.2 视频

视频部分相对比较简单，同样也需要发送 sequence header，但是只需要将设备编码时拿到的 SPS 以及 PPS 打包发送即可。

这部分的实现将围绕 libRTMP 的功能作简要介绍。请看下一节内容。

2.4 libRTMP 使用介绍

2.4.1 建立连接

libRTMP 是开源的 RTMP 协议库，按照 Adobe Flash Server 的实际通信方式以及公开的 RTMP 协议文档作为参考而制作的，使用者可以方便地调用库中的接口进行 RTMP 各个阶段的通信要求，如下是实现简单 RTMP 推流的具体流程：

新建 RTMP 对象：

```
m_pRtmp = RTMP_Alloc();
RTMP_Init(m_pRtmp);
```

建立连接以及创建流：

```
if (RTMP_SetupURL(m_pRtmp, (char *)_url) == 0) {
        return false;
}
RTMP_EnableWrite(m_pRtmp);
if (RTMP_Connect(m_pRtmp, nullptr) == 0) {
        return false;
}
if (RTMP_ConnectStream(m_pRtmp, 0) == 0) {
        return false;
}
```

这里要注意 RTMP_开头的函数多数为返回 1 为成功，返回 0 为失败。

2.4.2 AAC 音频包的发送

```
char *body = new char[_nSize + 2];
unsigned int i = 0;

// decoder spec info bytes
body[0] |= _chSoundType & 0x01;
body[0] |= (_chSoundSize << 1) & 0x02;
body[0] |= (_chSoundRate << 2) & 0x0c;
body[0] |= (SOUND_FORMAT_AAC << 4) & 0xf0;
// this is a sequence header
body[1] = SOUND_AAC_TYPE_RAW;

// send aac frame
memcpy(&body[2], _pData, _nSize);
bool bStatus = SendPacket(RTMP_PACKET_TYPE_AUDIO, body, _nSize + 2, _nTimeStamp);
delete[] body;
return bStatus;
```

AAC 音频包的发送顺序为：先发 sequence header, 之后连续发送普通音频数据包，sequence header 的发送流程可参照 demo，普通音频包的发送只需在发送前加上 2 字节的头部即可。

2.4.3 H264 视频包的发送

发送 H264 configuration：

```
// header
body[i++] = 0x17; // 1:keyframe  7:AVC
body[i++] = 0x00; // AVC sequence header

body[i++] = 0x00;
body[i++] = 0x00;
body[i++] = 0x00; // fill in 0;

// AVCDecoderConfigurationRecord.
body[i++] = 0x01; // configurationVersion
body[i++] = m_spsData[1]; // AVCProfileIndication
body[i++] = m_spsData[2]; // profile_compatibility
body[i++] = m_spsData[3]; // AVCLevelIndication
body[i++] = 0xff; // lengthSizeMinusOne

// sps nums
body[i++] = 0xE1; //&0x1f
// sps data length
body[i++] = m_sps.size >> 8;
body[i++] = m_sps.size & 0xff;
// sps data
memcpy(&body[i], m_sps.data, m_sps.size);
i = i + m_sps.size;

// pps nums
body[i++] = 0x01; //&0x1f
// pps data length
body[i++] = m_pps.size >> 8;
body[i++] = m_pps.size & 0xff;
// sps data
memcpy(&body[i], m_pps.data, m_pps.size);
i = i + m_pps.size;
```

需要填入 SPS 与 PPS，一般以获得的最近保留的 SPS 与 PPS 为主。

之后便可以发送 H.264 编码的内容，根据是否是关键帧来填充不同的 FLV tag 头部：

```
if (_bIsKeyFrame) {
        body[i++] = 0x17; // 1:Iframe 7:AVC
} else {
        body[i++] = 0x27; // 2:Pframe 7:AVC
}
body[i++] = 0x01; // AVC NALU

// composition time adjustment
body[i++] = 0x00;
body[i++] = 0x00;
body[i++] = 0x00;

// NALU size
body[i++] = _size >> 24;
body[i++] = _size >> 16;
body[i++] = _size >> 8;
body[i++] = _size & 0xff;

// NALU data
memcpy(&body[i], _data, _size);
```

如果是关键帧则填入 0x17，非关键帧则为 0x27

最后组成 RTMP 报文发送至服务端：

```
RTMPPacket packet;
RTMPPacket_Reset(&packet);
RTMPPacket_Alloc(&packet, _size);

packet.m_packetType = _nPacketType;
packet.m_nChannel = 0x04;
packet.m_headerType = RTMP_PACKET_SIZE_LARGE;
packet.m_nTimeStamp = _nTimestamp;
packet.m_nInfoField2 = m_pRtmp->m_stream_id;
packet.m_nBodySize = _size;
memcpy(packet.m_body, _data, _size);

int nRet = RTMP_SendPacket(m_pRtmp, &packet, 0);
```

参考 https://github.com/qiniudemo/Publishing/blob/master/pub/pub.cpp

3 兼容性

七牛的流媒体服务是由 Go 语言重写的，支持 RTMP，HLS 等多种主流多媒体协议，并且还在不断加入更多支持，但往往厂商在协议层，甚至编码层会自己重新开发，因此有些情况下会发生不兼容情况而导致无法播放、推流失败、无法转码的状况，该篇将主要描述这类情况。

3.1 编解码部分

3.1.1 H.265 HVEC

H.265 的发展是大势所趋，不过在现实中，如果要大面积推广新编码则需要软硬件厂商的支持，比如编码器，解码程序等，目前对于大部分的设备，可能的情况是即使提供了 H.265 的硬编码能力却无法适应播放端。七牛目前暂时不支持对 H.265 的转码，因此如果能够以 H.264 进行编码，将会有更好的兼容效果。

3.1.2 PCM 音频

RTMP 支持主流的几项音频：

0 = Linear PCM, platform endian
1 = ADPCM
2 = MP3
3 = Linear PCM, little endian
4 = Nellymoser 16 kHz mono
5 = Nellymoser 8 kHz mono
6 = Nellymoser
7 = G.711 A-law logarithmic PCM
8 = G.711 mu-law logarithmic PCM
9 = reserved
10 = AAC
11 = Speex
14 = MP3 8 kHz
15 = Device-specific sound

对于大多数互联网项目，使用 G711,ADPCM 和 AAC 更多。虽然目前主流的 HLS 视频播放不支持 G711 与 ADPCM 音频，但是可以将这两种音频通过转码至 AAC 后进行 HLS 的切片存储以提高播放体验。

在 RTMP 层面 PCM 是支持的一种音频编码，同样七牛 PILI 直播服务会正常转发 PCM 数据包，不过暂时不支持官方的直接存储服务，如果需要进行存储，需要进行一次转码服务，使音频以 AAC 的编码格式保存。

3.1.3 Profile 适配

Profile 对于 H.264 来说是功能方面的扩展，我们支持 baseline,main,以及 high 三种不同方式的编码，只是作为开发者，如果要通过 main, 或者 high 进行封装，需要了解其中的一些细节，比如 B 帧的处理部分，以及 CTS 的处理(可参看”3.2 协议层”部分)。

CompositionTime	IF CodecID == 7 SI24	IF AVCPacketType == 1 Composition time offset ELSE 0 See ISO 14496-12, 8.15.3 for an explanation of composition times. The offset in an FLV file is always in milliseconds.

参考 flash flv 文件 spec，flv_v10.1 E.4.3.1

3.1.4 码率适配

这个与摄像头的设置有关，对于一般的摄像头的输出，无论是 CBR 还是 VBR 都不会影响到流媒体服务，只是对于一般的视频输出码流会有一定的推荐标准，比如保持一定的码率，在更高的分辨率下 CBR 的情况下帧率必然会有所减低，在 VBR 的情况下，将会使得画面质量变差，因此关于码率的设置，不会影响到直播服务本身，但会实际影响到用户的观看体验。

3.2 RTMP 协议层部分

3.2.1 ParameterSet 信息

SPS，PPS 以及 SEI 一般是与关键帧一同发送，从缓冲区获取的 SPS 与 PPS 以及 SEI 应该与 IDR 帧一起发送，在 libRTMP 中则应该在一次 SendPacket 中发送出去，如果单独发送会造成播放端拿到无用帧而丢弃，具体视播放器实现而定，但一般两种策略皆可：其一是利用 libRTMP 与关键帧一同发送，其二是除了 configuration 中的 SPS 与 PPS 需要发送到服务端，其余的 SPS 与 PPS 可不发送至服务端。

3.2.2 时间戳

CTS：目前大部分平台摄像头不带有 B frame 输出，因此 CTS 都可填 0，如果不是，需要参考摄像头 API 提供的 PTS 以及 DTS，并且正确填写 CTS(参照 flv specification 10.1)。

PTS/DTS：PTS/DTS 是外部提供的时间戳，并非在编码层，因此单纯的 H264 文件编码流是无法直接获得时间戳的，ffmpeg 则有一套机制来计算时间戳。

一般摄像头平台提供的 API 都能够获得 H264 的时间戳，对于 RTMP 协议层面发送音视频包时可按照以下规则：

对于任意第 n 个音频包，它的 DTS(无 B frame 情况下 DTS=PTS)为 T，则它的 RTMP 时间戳则定义为：

TsAudio(1)=0

TsAudio(n)=T(n)-T(1)

对于任意第 n 个音频包，它的 PTS 为 T，则它的 RTMP 时间戳则定义为：

TsVideo(1)=0

TsVideo(n)=T(n)-T(1)

参考材料

[1] ADOBE FLASH VIDEO FILE FORMAT SPECIFICATION VERSION 10.1

[2] Real Time Messaging Chunk Stream Protocol 1.0

[3] MPEG-4 Audio:ISO/IEC 14496-3:2009