原 RTMPdump（libRTMP） 源代码分析 10： 处理各种消息（Message）

2013年10月24日 00:24:15

===

RTMPdump(libRTMP) 源代码分析系列文章：

===

函数调用结构图

RTMPDump (libRTMP)的整体的函数调用结构图如下图所示。

单击查看大图

详细分析

已经连续写了一系列的博客了，其实大部分内容都是去年搞RTMP研究的时候积累的经验，回顾一下过去的知识，其实RTMPdump（libRTMP）主要的功能也都分析的差不多了，现在感觉还需要一些查漏补缺。主要就是它是如何处理各种消息（Message）的这方面还没有研究的特明白，在此需要详细研究一下。

再来看一下RTMPdump（libRTMP）的“灵魂”函数RTMP_ClientPacket()，主要完成了各种消息的处理。

[cpp]

```cpp
//处理接收到的数据
int
RTMP_ClientPacket(RTMP *r, RTMPPacket *packet)
{
  int bHasMediaPacket = 0;
  switch (packet->m_packetType)
    {
    //RTMP消息类型ID=1,设置块大小
    case 0x01:
      /* chunk size */
        //----------------
        r->dlg->AppendCInfo("处理收到的数据。消息 Set Chunk Size (typeID=1)。");
        //-----------------------------
        RTMP_LogPrintf("处理消息 Set Chunk Size (typeID=1)\n");
      HandleChangeChunkSize(r, packet);
      break;
    //RTMP消息类型ID=3，致谢
    case 0x03:
      /* bytes read report */
      RTMP_Log(RTMP_LOGDEBUG, "%s, received: bytes read report", __FUNCTION__);
      break;
    //RTMP消息类型ID=4，用户控制
    case 0x04:
      /* ctrl */
        //----------------
        r->dlg->AppendCInfo("处理收到的数据。消息 User Control (typeID=4)。");
        //-----------------------------
        RTMP_LogPrintf("处理消息 User Control (typeID=4)\n");
      HandleCtrl(r, packet);
```

```
      break;
    //RTMP消息类型ID=5
    case 0x05:
      /* server bw */
        //----------------
        r->dlg->AppendCInfo("处理收到的数据。消息 Window Acknowledgement Size (typeID=5)。");
        //-----------------------------
        RTMP_LogPrintf("处理消息 Window Acknowledgement Size (typeID=5)\n");
      HandleServerBW(r, packet);
      break;
    //RTMP消息类型ID=6
    case 0x06:
      /* client bw */
        //----------------
        r->dlg->AppendCInfo("处理收到的数据。消息 Set Peer Bandwidth (typeID=6)。");
        //-----------------------------
        RTMP_LogPrintf("处理消息 Set Peer Bandwidth (typeID=6)\n");
      HandleClientBW(r, packet);
      break;
    //RTMP消息类型ID=8，音频数据
    case 0x08:
      /* audio data */
      /*RTMP_Log(RTMP_LOGDEBUG, "%s, received: audio %lu bytes", __FUNCTION__, packet.m_nBodySize); */
      HandleAudio(r, packet);
      bHasMediaPacket = 1;
      if (!r->m_mediaChannel)
    r->m_mediaChannel = packet->m_nChannel;
      if (!r->m_pausing)
    r->m_mediaStamp = packet->m_nTimeStamp;
      break;
    //RTMP消息类型ID=9，视频数据
    case 0x09:
      /* video data */
      /*RTMP_Log(RTMP_LOGDEBUG, "%s, received: video %lu bytes", __FUNCTION__, packet.m_nBodySize); */
      HandleVideo(r, packet);
      bHasMediaPacket = 1;
      if (!r->m_mediaChannel)
    r->m_mediaChannel = packet->m_nChannel;
      if (!r->m_pausing)
    r->m_mediaStamp = packet->m_nTimeStamp;
      break;
    //RTMP消息类型ID=15，AMF3编码，忽略
    case 0x0F:          /* flex stream send */
      RTMP_Log(RTMP_LOGDEBUG,
      "%s, flex stream send, size %lu bytes, not supported, ignoring",
      __FUNCTION__, packet->m_nBodySize);
      break;
    //RTMP消息类型ID=16，AMF3编码，忽略
    case 0x10:          /* flex shared object */
      RTMP_Log(RTMP_LOGDEBUG,
      "%s, flex shared object, size %lu bytes, not supported, ignoring",
      __FUNCTION__, packet->m_nBodySize);
      break;
    //RTMP消息类型ID=17，AMF3编码，忽略
    case 0x11:          /* flex message */
      {
    RTMP_Log(RTMP_LOGDEBUG,
        "%s, flex message, size %lu bytes, not fully supported",
        __FUNCTION__, packet->m_nBodySize);
    /*RTMP_LogHex(packet.m_body, packet.m_nBodySize); */

    /* some DEBUG code */
#if 0
       RTMP_LIB_AMFObject obj;
       int nRes = obj.Decode(packet.m_body+1, packet.m_nBodySize-1);
       if(nRes < 0) {
       RTMP_Log(RTMP_LOGERROR, "%s, error decoding AMF3 packet", __FUNCTION__);
       /*return; */
       }

       obj.Dump();
#endif

    if (HandleInvoke(r, packet->m_body + 1, packet->m_nBodySize - 1) == 1)
      bHasMediaPacket = 2;
    break;
      }
    //RTMP消息类型ID=18，AMF0编码，数据消息
    case 0x12:
      /* metadata (notify) */

      RTMP_Log(RTMP_LOGDEBUG, "%s, received: notify %lu bytes", __FUNCTION__,
      packet->m_nBodySize);
      //处理元数据,暂时注释
      /*
      if (HandleMetadata(r, packet->m_body, packet->m_nBodySize))
    bHasMediaPacket = 1;
      break;
      */
    //RTMP消息类型ID=19，AMF0编码，忽略
    case 0x13:
```

```
        RTMP_Log(RTMP_LOGDEBUG, "%s, shared object, not supported, ignoring",
        __FUNCTION__);
        break;
      //RTMP消息类型ID=20，AMF0编码，命令消息
      //处理命令消息！
      case 0x14:
          //----------------
          r->dlg->AppendCInfo("处理收到的数据。消息 命令 (AMF0编码) (typeID=20)。");
          //----------------------------
        /* invoke */
        RTMP_Log(RTMP_LOGDEBUG, "%s, received: invoke %lu bytes", __FUNCTION__,
        packet->m_nBodySize);
        RTMP_LogPrintf("处理命令消息 (typeID=20，AMF0编码)\n");
        /*RTMP_LogHex(packet.m_body, packet.m_nBodySize); */

        if (HandleInvoke(r, packet->m_body, packet->m_nBodySize) == 1)
      bHasMediaPacket = 2;
        break;
      //RTMP消息类型ID=22
      case 0x16:
        {
      /* go through FLV packets and handle metadata packets */
      unsigned int pos = 0;
      uint32_t nTimeStamp = packet->m_nTimeStamp;

      while (pos + 11 < packet->m_nBodySize)
        {
          uint32_t dataSize = AMF_DecodeInt24(packet->m_body + pos + 1);   /* size without header (11) and prevTagSize (4) */

          if (pos + 11 + dataSize + 4 > packet->m_nBodySize)
            {
          RTMP_Log(RTMP_LOGWARNING, "Stream corrupt?!");
          break;
            }
          if (packet->m_body[pos] == 0x12)
            {
          HandleMetadata(r, packet->m_body + pos + 11, dataSize);
            }
          else if (packet->m_body[pos] == 8 || packet->m_body[pos] == 9)
            {
          nTimeStamp = AMF_DecodeInt24(packet->m_body + pos + 4);
          nTimeStamp |= (packet->m_body[pos + 7] << 24);
            }
          pos += (11 + dataSize + 4);
        }
      if (!r->m_pausing)
        r->m_mediaStamp = nTimeStamp;

      /* FLV tag(s) */
      /*RTMP_Log(RTMP_LOGDEBUG, "%s, received: FLV tag(s) %lu bytes", __FUNCTION__, packet.m_nBodySize); */
      bHasMediaPacket = 1;
      break;
        }
      default:
        RTMP_Log(RTMP_LOGDEBUG, "%s, unknown packet type received: 0x%02x", __FUNCTION__,
        packet->m_packetType);
#ifdef _DEBUG
        RTMP_LogHex(RTMP_LOGDEBUG, (const uint8_t *)packet->m_body, packet->m_nBodySize);
#endif
    }

  return bHasMediaPacket;
}
```

前文已经分析过当消息类型ID为0x14（20）的时候，即AMF0编码的命令消息的时候，会调用HandleInvoke()进行处理。

参考：RTMPdump（libRTMP） 源代码分析 7： 建立一个流媒体连接 （NetStream部分 2）

这里就不再对这种类型ID的消息进行分析了，分析一下其他类型的消息，毕竟从发起一个RTMP连接到接收视音频数据这个过程中是要处理很多消息的。

参考：RTMP流媒体播放过程

下面我们按照消息ID从小到大的顺序，看看接收到的各种消息都是如何处理的。

消息类型ID是0x01的消息功能是“设置块（Chunk）大小”，处理函数是HandleChangeChunkSize()，可见函数内容很简单。

[cpp]

```cpp
static void
HandleChangeChunkSize(RTMP *r, const RTMPPacket *packet)
{
  if (packet->m_nBodySize >= 4)
    {
      r->m_inChunkSize = AMF_DecodeInt32(packet->m_body);
      RTMP_Log(RTMP_LOGDEBUG, "%s, received: chunk size change to %d", __FUNCTION__,
      r->m_inChunkSize);
    }
}
```

消息类型ID是0x03的消息功能是“致谢”，没有处理函数。

消息类型ID是0x04的消息功能是“用户控制（UserControl）”，处理函数是HandleCtrl()，这类的消息出现的频率非常高，函数体如下所示。具体用户控制消息的作用这里就不多说了，有相应的文档可以参考。

注：该函数中间有一段很长的英文注释，英语好的大神可以看一看

[cpp]

```cpp
//处理用户控制(UserControl)消息。用户控制消息是服务器端发出的。
static void
HandleCtrl(RTMP *r, const RTMPPacket *packet)
{
  short nType = -1;
  unsigned int tmp;
  if (packet->m_body && packet->m_nBodySize >= 2)
    //事件类型(2B)
    nType = AMF_DecodeInt16(packet->m_body);
  RTMP_Log(RTMP_LOGDEBUG, "%s, received ctrl. type: %d, len: %d", __FUNCTION__, nType,
      packet->m_nBodySize);
  /*RTMP_LogHex(packet.m_body, packet.m_nBodySize); */

  if (packet->m_nBodySize >= 6)
    {
    //不同事件类型做不同处理
      switch (nType)
    {
    //流开始
    case 0:
    //流ID
      tmp = AMF_DecodeInt32(packet->m_body + 2);
      RTMP_Log(RTMP_LOGDEBUG, "%s, Stream Begin %d", __FUNCTION__, tmp);
      break;
    //流结束
    case 1:
    //流ID
      tmp = AMF_DecodeInt32(packet->m_body + 2);
      RTMP_Log(RTMP_LOGDEBUG, "%s, Stream EOF %d", __FUNCTION__, tmp);
      if (r->m_pausing == 1)
        r->m_pausing = 2;
      break;
    //流枯竭
    case 2:
    //流ID
      tmp = AMF_DecodeInt32(packet->m_body + 2);
      RTMP_Log(RTMP_LOGDEBUG, "%s, Stream Dry %d", __FUNCTION__, tmp);
      break;
    //是录制流
    case 4:
      tmp = AMF_DecodeInt32(packet->m_body + 2);
      RTMP_Log(RTMP_LOGDEBUG, "%s, Stream IsRecorded %d", __FUNCTION__, tmp);
      break;
    //Ping客户端
    case 6:      /* server ping. reply with pong. */
      tmp = AMF_DecodeInt32(packet->m_body + 2);
      RTMP_Log(RTMP_LOGDEBUG, "%s, Ping %d", __FUNCTION__, tmp);
      RTMP_SendCtrl(r, 0x07, tmp, 0);
      break;

    /* FMS 3.5 servers send the following two controls to let the client
     * know when the server has sent a complete buffer. I.e., when the
     * server has sent an amount of data equal to m_nBufferMS in duration.
     * The server meters its output so that data arrives at the client
     * in realtime and no faster.
     *
     * The rtmpdump program tries to set m_nBufferMS as large as
     * possible, to force the server to send data as fast as possible.
     * In practice, the server appears to cap this at about 1 hour's
     * worth of data. After the server has sent a complete buffer, and
     * sends this BufferEmpty message, it will wait until the play
     * duration of that buffer has passed before sending a new buffer.
     * The BufferReady message will be sent when the new buffer starts.
     * (There is no BufferReady message for the very first buffer;
     * presumably the Stream Begin message is sufficient for that
     * purpose.)
     *
```

```
   * If the network speed is much faster than the data bitrate, then
   * there may be long delays between the end of one buffer and the
   * start of the next.
   *
   * Since usually the network allows data to be sent at
   * faster than realtime, and rtmpdump wants to download the data
   * as fast as possible, we use this RTMP_LF_BUFX hack: when we
   * get the BufferEmpty message, we send a Pause followed by an
   * Unpause. This causes the server to send the next buffer immediately
   * instead of waiting for the full duration to elapse. (That's
   * also the purpose of the ToggleStream function, which rtmpdump
   * calls if we get a read timeout.)
   *
   * Media player apps don't need this hack since they are just
   * going to play the data in realtime anyway. It also doesn't work
   * for live streams since they obviously can only be sent in
   * realtime. And it's all moot if the network speed is actually
   * slower than the media bitrate.
   */
  case 31:
    tmp = AMF_DecodeInt32(packet->m_body + 2);
    RTMP_Log(RTMP_LOGDEBUG, "%s, Stream BufferEmpty %d", __FUNCTION__, tmp);
    if (!(r->Link.lFlags & RTMP_LF_BUFX))
      break;
    if (!r->m_pausing)
      {
        r->m_pauseStamp = r->m_channelTimestamp[r->m_mediaChannel];
        RTMP_SendPause(r, TRUE, r->m_pauseStamp);
        r->m_pausing = 1;
      }
    else if (r->m_pausing == 2)
      {
        RTMP_SendPause(r, FALSE, r->m_pauseStamp);
        r->m_pausing = 3;
      }
    break;

  case 32:
    tmp = AMF_DecodeInt32(packet->m_body + 2);
    RTMP_Log(RTMP_LOGDEBUG, "%s, Stream BufferReady %d", __FUNCTION__, tmp);
    break;

  default:
    tmp = AMF_DecodeInt32(packet->m_body + 2);
    RTMP_Log(RTMP_LOGDEBUG, "%s, Stream xx %d", __FUNCTION__, tmp);
    break;
  }

  }

if (nType == 0x1A)
  {
    RTMP_Log(RTMP_LOGDEBUG, "%s, SWFVerification ping received: ", __FUNCTION__);
    if (packet->m_nBodySize > 2 && packet->m_body[2] > 0x01)
  {
    RTMP_Log(RTMP_LOGERROR,
          "%s: SWFVerification Type %d request not supported! Patches welcome...",
      __FUNCTION__, packet->m_body[2]);
  }
#ifdef CRYPTO
    /*RTMP_LogHex(packet.m_body, packet.m_nBodySize); */

    /* respond with HMAC SHA256 of decompressed SWF, key is the 30byte player key, also the last 30 bytes of the server handshake a
applied */
    else if (r->Link.SWFSize)
  {
    RTMP_SendCtrl(r, 0x1B, 0, 0);
  }
    else
  {
    RTMP_Log(RTMP_LOGERROR,
        "%s: Ignoring SWFVerification request, use --swfVfy!",
        __FUNCTION__);
  }
#else
    RTMP_Log(RTMP_LOGERROR,
    "%s: Ignoring SWFVerification request, no CRYPTO support!",
    __FUNCTION__);
#endif
  }
}
```

消息类型ID是0x05的消息功能是“窗口致谢大小（Window Acknowledgement Size，翻译的真是挺别扭）”，处理函数是HandleServerBW()。在这里注意一下，该消息在Adobe官方公开的文档中叫“Window Acknowledgement Size”，但是在Adobe公开协议规范之前，破解RTMP协议的组织一直管该协议叫“ServerBW”，只是个称呼，倒是也无所谓~处理代码很简单：

[cpp]

```cpp
static void
HandleServerBW(RTMP *r, const RTMPPacket *packet)
{
  r->m_nServerBW = AMF_DecodeInt32(packet->m_body);
  RTMP_Log(RTMP_LOGDEBUG, "%s: server BW = %d", __FUNCTION__, r->m_nServerBW);
}
```

消息类型ID是0x06的消息功能是“设置对等端带宽（Set Peer Bandwidth）”，处理函数是HandleClientBW()。与上一种消息一样，该消息在Adobe官方公开的文档中叫“Set Peer Bandwidth”，但是在Adobe公开协议规范之前，破解RTMP协议的组织一直管该协议叫“ClientBW”。处理函数也不复杂：

[cpp]

```cpp
static void
HandleClientBW(RTMP *r, const RTMPPacket *packet)
{
  r->m_nClientBW = AMF_DecodeInt32(packet->m_body);
  if (packet->m_nBodySize > 4)
    r->m_nClientBW2 = packet->m_body[4];
  else
    r->m_nClientBW2 = -1;
  RTMP_Log(RTMP_LOGDEBUG, "%s: client BW = %d %d", __FUNCTION__, r->m_nClientBW,
      r->m_nClientBW2);
}
```

消息类型ID是0x08的消息用于传输音频数据，在这里不处理。

消息类型ID是0x09的消息用于传输音频数据，在这里不处理。

消息类型ID是0x0F-11的消息用于传输AMF3编码的命令。

消息类型ID是0x12-14的消息用于传输AMF0编码的命令。

注：消息类型ID是0x14的消息很重要，用于传输AMF0编码的命令，已经做过分析。

rtmpdump源代码（Linux）：http://download.csdn.net/detail/leixiaohua1020/6376561

rtmpdump源代码（VC 2005 工程）：http://download.csdn.net/detail/leixiaohua1020/6563163

文章标签： RTMPdump rtmp 源代码 Message 流媒体

个人分类： libRTMP

所属专栏： 开源多媒体项目源代码分析