

## 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 阅读数：10058

=====

RTMPdump(libRTMP) 源代码分析系列文章：

[RTMPdump 源代码分析 1：main\(\)函数](#)

[RTMPDump \(libRTMP\) 源代码分析2：解析RTMP地址——RTMP\\_ParseURL\(\)](#)

[RTMPdump \(libRTMP\) 源代码分析3：AMF编码](#)

[RTMPdump \(libRTMP\) 源代码分析4：连接第一步——握手 \(HandShake\)](#)

[RTMPdump \(libRTMP\) 源代码分析5：建立一个流媒体连接 \(NetConnection部分\)](#)

[RTMPdump \(libRTMP\) 源代码分析6：建立一个流媒体连接 \(NetStream部分 1\)](#)

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

[RTMPdump \(libRTMP\) 源代码分析8：发送消息 \(Message\)](#)

[RTMPdump \(libRTMP\) 源代码分析9：接收消息 \(Message\) \(接收视音频数据\)](#)

[RTMPdump \(libRTMP\) 源代码分析10：处理各种消息 \(Message\)](#)

=====

### 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]
1. //处理接收到的数据
2. int
3. RTMP_ClientPacket(RTMP *r, RTMPPacket *packet)
4. {
5.     int bHasMediaPacket = 0;
6.     switch (packet->m_packetType)
7.     {
8.         //RTMP消息类型ID=1,设置块大小
9.         case 0x01:
10.            /* chunk size */
11.            //-----
12.            r->dlg->AppendCInfo("处理收到的数据。消息 Set Chunk Size (typeID=1)。");
13.            //-----
14.            RTMP_LogPrintf("处理消息 Set Chunk Size (typeID=1)\n");
15.            HandleChangeChunkSize(r, packet);
16.            break;
17.        //RTMP消息类型ID=3, 致谢
18.        case 0x03:
19.            /* bytes read report */
20.            RTMP_Log(RTMP_LOGDEBUG, "%s, received: bytes read report", __FUNCTION__);
21.            break;
22.        //RTMP消息类型ID=4, 用户控制
23.        case 0x04:
24.            /* ctrl */
25.            //-----
26.            r->dlg->AppendCInfo("处理收到的数据。消息 User Control (typeID=4)。");
27.            //-----
28.            RTMP_LogPrintf("处理消息 User Control (typeID=4)\n");
29.            HandleCtrl(r, packet);
```

```

30.         break;
31. //RTMP消息类型ID=5
32. case 0x05:
33.     /* server bw */
34.     //-----
35.     r->dlg->AppendCInfo("处理收到的数据。消息 Window Acknowledgement Size (typeID=5)。");
36.     //-----
37.     RTMP_LogPrintf("处理消息 Window Acknowledgement Size (typeID=5)\n");
38.     HandleServerBW(r, packet);
39.     break;
40. //RTMP消息类型ID=6
41. case 0x06:
42.     /* client bw */
43.     //-----
44.     r->dlg->AppendCInfo("处理收到的数据。消息 Set Peer Bandwidth (typeID=6)。");
45.     //-----
46.     RTMP_LogPrintf("处理消息 Set Peer Bandwidth (typeID=6)\n");
47.     HandleClientBW(r, packet);
48.     break;
49. //RTMP消息类型ID=8, 音频数据
50. case 0x08:
51.     /* audio data */
52.     /*RTMP_Log(RTMP_LOGDEBUG, "%s, received: audio %lu bytes", __FUNCTION__, packet.m_nBodySize); */
53.     HandleAudio(r, packet);
54.     bHasMediaPacket = 1;
55.     if (!r->m_mediaChannel)
56.     r->m_mediaChannel = packet->m_nChannel;
57.     if (!r->m_pausing)
58.     r->m_mediaStamp = packet->m_nTimeStamp;
59.     break;
60. //RTMP消息类型ID=9, 视频数据
61. case 0x09:
62.     /* video data */
63.     /*RTMP_Log(RTMP_LOGDEBUG, "%s, received: video %lu bytes", __FUNCTION__, packet.m_nBodySize); */
64.     HandleVideo(r, packet);
65.     bHasMediaPacket = 1;
66.     if (!r->m_mediaChannel)
67.     r->m_mediaChannel = packet->m_nChannel;
68.     if (!r->m_pausing)
69.     r->m_mediaStamp = packet->m_nTimeStamp;
70.     break;
71. //RTMP消息类型ID=15, AMF3编码, 忽略
72. case 0x0F: /* flex stream send */
73.     RTMP_Log(RTMP_LOGDEBUG,
74.         "%s, flex stream send, size %lu bytes, not supported, ignoring",
75.         __FUNCTION__, packet->m_nBodySize);
76.     break;
77. //RTMP消息类型ID=16, AMF3编码, 忽略
78. case 0x10: /* flex shared object */
79.     RTMP_Log(RTMP_LOGDEBUG,
80.         "%s, flex shared object, size %lu bytes, not supported, ignoring",
81.         __FUNCTION__, packet->m_nBodySize);
82.     break;
83. //RTMP消息类型ID=17, AMF3编码, 忽略
84. case 0x11: /* flex message */
85.     {
86.         RTMP_Log(RTMP_LOGDEBUG,
87.             "%s, flex message, size %lu bytes, not fully supported",
88.             __FUNCTION__, packet->m_nBodySize);
89.         /*RTMP_LogHex(packet.m_body, packet.m_nBodySize); */
90.
91.         /* some DEBUG code */
92.         #if 0
93.             RTMP_LIB_AMFObject obj;
94.             int nRes = obj.Decode(packet.m_body+1, packet.m_nBodySize-1);
95.             if(nRes < 0) {
96.                 RTMP_Log(RTMP_LOGERROR, "%s, error decoding AMF3 packet", __FUNCTION__);
97.                 /*return; */
98.             }
99.
100.             obj.Dump();
101.         #endif
102.
103.         if (HandleInvoke(r, packet->m_body + 1, packet->m_nBodySize - 1) == 1)
104.             bHasMediaPacket = 2;
105.         break;
106.     }
107. //RTMP消息类型ID=18, AMF0编码, 数据消息
108. case 0x12:
109.     /* metadata (notify) */
110.
111.     RTMP_Log(RTMP_LOGDEBUG, "%s, received: notify %lu bytes", __FUNCTION__,
112.         packet->m_nBodySize);
113.     //处理元数据, 暂时注释
114.     /*
115.     if (HandleMetadata(r, packet->m_body, packet->m_nBodySize))
116.     bHasMediaPacket = 1;
117.     break;
118.     */
119. //RTMP消息类型ID=19, AMF0编码, 忽略
120. case 0x13:
121.     RTMP_Log(RTMP_LOGDEBUG, "%s, flex shared object, not supported, ignoring",

```

```

121.         RTMP_Log(RTMP_LOGDEBUG, "%s, shared object, not supported, ignoring",
122.             __FUNCTION__);
123.         break;
124.         //RTMP消息类型ID=20, AMF0编码, 命令消息
125.         //处理命令消息!
126.         case 0x14:
127.             //-----
128.             r->dlg->AppendCInfo("处理收到的数据. 消息 命令 (AMF0编码) (typeID=20)。");
129.             //-----
130.             /* invoke */
131.             RTMP_Log(RTMP_LOGDEBUG, "%s, received: invoke %lu bytes", __FUNCTION__,
132.                 packet->m_nBodySize);
133.             RTMP_LogPrintf("处理命令消息 (typeID=20, AMF0编码)\n");
134.             /*RTMP_LogHex(packet.m_body, packet.m_nBodySize); */
135.
136.             if (HandleInvoke(r, packet->m_body, packet->m_nBodySize) == 1)
137.                 bHasMediaPacket = 2;
138.             break;
139.             //RTMP消息类型ID=22
140.             case 0x16:
141.                 {
142.                     /* go through FLV packets and handle metadata packets */
143.                     unsigned int pos = 0;
144.                     uint32_t nTimeStamp = packet->m_nTimeStamp;
145.
146.                     while (pos + 11 < packet->m_nBodySize)
147.                     {
148.                         uint32_t dataSize = AMF_DecodeInt24(packet->m_body + pos + 1); /* size without header (11) and prevTagSize (4) */
149.
150.                         if (pos + 11 + dataSize + 4 > packet->m_nBodySize)
151.                         {
152.                             RTMP_Log(RTMP_LOGWARNING, "Stream corrupt?!");
153.                             break;
154.                         }
155.                         if (packet->m_body[pos] == 0x12)
156.                         {
157.                             HandleMetadata(r, packet->m_body + pos + 11, dataSize);
158.                         }
159.                         else if (packet->m_body[pos] == 8 || packet->m_body[pos] == 9)
160.                         {
161.                             nTimeStamp = AMF_DecodeInt24(packet->m_body + pos + 4);
162.                             nTimeStamp |= (packet->m_body[pos + 7] << 24);
163.                         }
164.                         pos += (11 + dataSize + 4);
165.                     }
166.                     if (!r->m_pausing)
167.                         r->m_mediaStamp = nTimeStamp;
168.
169.                     /* FLV tag(s) */
170.                     /*RTMP_Log(RTMP_LOGDEBUG, "%s, received: FLV tag(s) %lu bytes", __FUNCTION__, packet.m_nBodySize); */
171.                     bHasMediaPacket = 1;
172.                     break;
173.                 }
174.             default:
175.                 RTMP_Log(RTMP_LOGDEBUG, "%s, unknown packet type received: 0x%02x", __FUNCTION__,
176.                     packet->m_packetType);
177.                 #ifdef _DEBUG
178.                 RTMP_LogHex(RTMP_LOGDEBUG, (const uint8_t *)packet->m_body, packet->m_nBodySize);
179.                 #endif
180.             }
181.
182.             return bHasMediaPacket;
183.         }

```

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。

```

1. static void
2. HandleChangeChunkSize(RTMP *r, const RTMPPacket *packet)
3. {
4.     if (packet->m_nBodySize >= 4)
5.     {
6.         r->m_inChunkSize = AMF_DecodeInt32(packet->m_body);
7.         RTMP_Log(RTMP_LOGDEBUG, "%s, received: chunk size change to %d", __FUNCTION__,
8.             r->m_inChunkSize);
9.     }
10. }

```

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

```

1. //处理用户控制(UserControl)消息。用户控制消息是服务器端发出的。
2. static void
3. HandleCtrl(RTMP *r, const RTMPPacket *packet)
4. {
5.     short nType = -1;
6.     unsigned int tmp;
7.     if (packet->m_body && packet->m_nBodySize >= 2)
8.         //事件类型(2B)
9.         nType = AMF_DecodeInt16(packet->m_body);
10.    RTMP_Log(RTMP_LOGDEBUG, "%s, received ctrl. type: %d, len: %d", __FUNCTION__, nType,
11.        packet->m_nBodySize);
12.    /*RTMP_LogHex(packet.m_body, packet.m_nBodySize); */
13.
14.    if (packet->m_nBodySize >= 6)
15.    {
16.        //不同事件类型做不同处理
17.        switch (nType)
18.        {
19.            //流开始
20.            case 0:
21.                //流ID
22.                tmp = AMF_DecodeInt32(packet->m_body + 2);
23.                RTMP_Log(RTMP_LOGDEBUG, "%s, Stream Begin %d", __FUNCTION__, tmp);
24.                break;
25.            //流结束
26.            case 1:
27.                //流ID
28.                tmp = AMF_DecodeInt32(packet->m_body + 2);
29.                RTMP_Log(RTMP_LOGDEBUG, "%s, Stream EOF %d", __FUNCTION__, tmp);
30.                if (r->m_pausing == 1)
31.                    r->m_pausing = 2;
32.                break;
33.            //流枯竭
34.            case 2:
35.                //流ID
36.                tmp = AMF_DecodeInt32(packet->m_body + 2);
37.                RTMP_Log(RTMP_LOGDEBUG, "%s, Stream Dry %d", __FUNCTION__, tmp);
38.                break;
39.            //是录制流
40.            case 4:
41.                tmp = AMF_DecodeInt32(packet->m_body + 2);
42.                RTMP_Log(RTMP_LOGDEBUG, "%s, Stream IsRecorded %d", __FUNCTION__, tmp);
43.                break;
44.            //Ping客户端
45.            case 6: /* server ping. reply with pong. */
46.                tmp = AMF_DecodeInt32(packet->m_body + 2);
47.                RTMP_Log(RTMP_LOGDEBUG, "%s, Ping %d", __FUNCTION__, tmp);
48.                RTMP_SendCtrl(r, 0x07, tmp, 0);
49.                break;
50.
51.            /* FMS 3.5 servers send the following two controls to let the client
52.             * know when the server has sent a complete buffer. I.e., when the
53.             * server has sent an amount of data equal to m_nBufferMS in duration.
54.             * The server meters its output so that data arrives at the client
55.             * in realtime and no faster.
56.             *
57.             * The rtmpdump program tries to set m_nBufferMS as large as
58.             * possible, to force the server to send data as fast as possible.
59.             * In practice, the server appears to cap this at about 1 hour's
60.             * worth of data. After the server has sent a complete buffer, and
61.             * sends this BufferEmpty message, it will wait until the play
62.             * duration of that buffer has passed before sending a new buffer.
63.             * The BufferReady message will be sent when the new buffer starts.
64.             * (There is no BufferReady message for the very first buffer;
65.             * presumably the Stream Begin message is sufficient for that
66.             * purpose.)
67.             *

```

```

68.  * If the network speed is much faster than the data bitrate, then
69.  * there may be long delays between the end of one buffer and the
70.  * start of the next.
71.  *
72.  * Since usually the network allows data to be sent at
73.  * faster than realtime, and rtmpdump wants to download the data
74.  * as fast as possible, we use this RTMP_LF_BUFHACK hack: when we
75.  * get the BufferEmpty message, we send a Pause followed by an
76.  * Unpause. This causes the server to send the next buffer immediately
77.  * instead of waiting for the full duration to elapse. (That's
78.  * also the purpose of the ToggleStream function, which rtmpdump
79.  * calls if we get a read timeout.)
80.  *
81.  * Media player apps don't need this hack since they are just
82.  * going to play the data in realtime anyway. It also doesn't work
83.  * for live streams since they obviously can only be sent in
84.  * realtime. And it's all moot if the network speed is actually
85.  * slower than the media bitrate.
86.  */
87.  case 31:
88.      tmp = AMF_DecodeInt32(packet->m_body + 2);
89.      RTMP_Log(RTMP_LOGDEBUG, "%s, Stream BufferEmpty %d", __FUNCTION__, tmp);
90.      if (!(r->Link.LFlags & RTMP_LF_BUFHACK))
91.          break;
92.      if (!r->m_pausing)
93.      {
94.          r->m_pauseStamp = r->m_channelTimestamp[r->m_mediaChannel];
95.          RTMP_SendPause(r, TRUE, r->m_pauseStamp);
96.          r->m_pausing = 1;
97.      }
98.      else if (r->m_pausing == 2)
99.      {
100.          RTMP_SendPause(r, FALSE, r->m_pauseStamp);
101.          r->m_pausing = 3;
102.      }
103.      break;
104.
105.  case 32:
106.      tmp = AMF_DecodeInt32(packet->m_body + 2);
107.      RTMP_Log(RTMP_LOGDEBUG, "%s, Stream BufferReady %d", __FUNCTION__, tmp);
108.      break;
109.
110.  default:
111.      tmp = AMF_DecodeInt32(packet->m_body + 2);
112.      RTMP_Log(RTMP_LOGDEBUG, "%s, Stream xx %d", __FUNCTION__, tmp);
113.      break;
114.  }
115.
116.  }
117.
118.  if (nType == 0x1A)
119.  {
120.      RTMP_Log(RTMP_LOGDEBUG, "%s, SWFVerification ping received: ", __FUNCTION__);
121.      if (packet->m_nBodySize > 2 && packet->m_body[2] > 0x01)
122.      {
123.          RTMP_Log(RTMP_LOGERROR,
124.                  "%s: SWFVerification Type %d request not supported! Patches welcome...",
125.                  __FUNCTION__, packet->m_body[2]);
126.      }
127.  #ifdef CRYPTO
128.      /*RTMP_LogHex(packet.m_body, packet.m_nBodySize); */
129.
130.      /* respond with HMAC SHA256 of decompressed SWF, key is the 30byte player key, also the last 30 bytes of the server handshake a
131.      applied */
132.      else if (r->Link.SWFSize)
133.      {
134.          RTMP_SendCtrl(r, 0x1B, 0, 0);
135.      }
136.      else
137.      {
138.          RTMP_Log(RTMP_LOGERROR,
139.                  "%s: Ignoring SWFVerification request, use --swfVfy!",
140.                  __FUNCTION__);
141.      }
142.  #else
143.      RTMP_Log(RTMP_LOGERROR,
144.              "%s: Ignoring SWFVerification request, no CRYPTO support!",
145.              __FUNCTION__);
146.  #endif
147.  }

```

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]
1. static void
2. HandleServerBW(RTMP *r, const RTMPPacket *packet)
3. {
4.     r->m_nServerBW = AMF_DecodeInt32(packet->m_body);
5.     RTMP_Log(RTMP_LOGDEBUG, "%s: server BW = %d", __FUNCTION__, r->m_nServerBW);
6. }
```

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]
1. static void
2. HandleClientBW(RTMP *r, const RTMPPacket *packet)
3. {
4.     r->m_nClientBW = AMF_DecodeInt32(packet->m_body);
5.     if (packet->m_nBodySize > 4)
6.         r->m_nClientBW2 = packet->m_body[4];
7.     else
8.         r->m_nClientBW2 = -1;
9.     RTMP_Log(RTMP_LOGDEBUG, "%s: client BW = %d %d", __FUNCTION__, r->m_nClientBW,
10.             r->m_nClientBW2);
11. }
```

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>

版权声明：本文为博主原创文章，未经博主允许不得转载。<https://blog.csdn.net/leixiaohua1020/article/details/12972399>

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](#)

此PDF由spygg生成, 请尊重原作者版权!!!

我的邮箱:liushidc@163.com