原 RTMPdump（libRTMP） 源代码分析 8： 发送消息（Message）

2013年10月23日 15:03:32　阅读数：16481

==========

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)

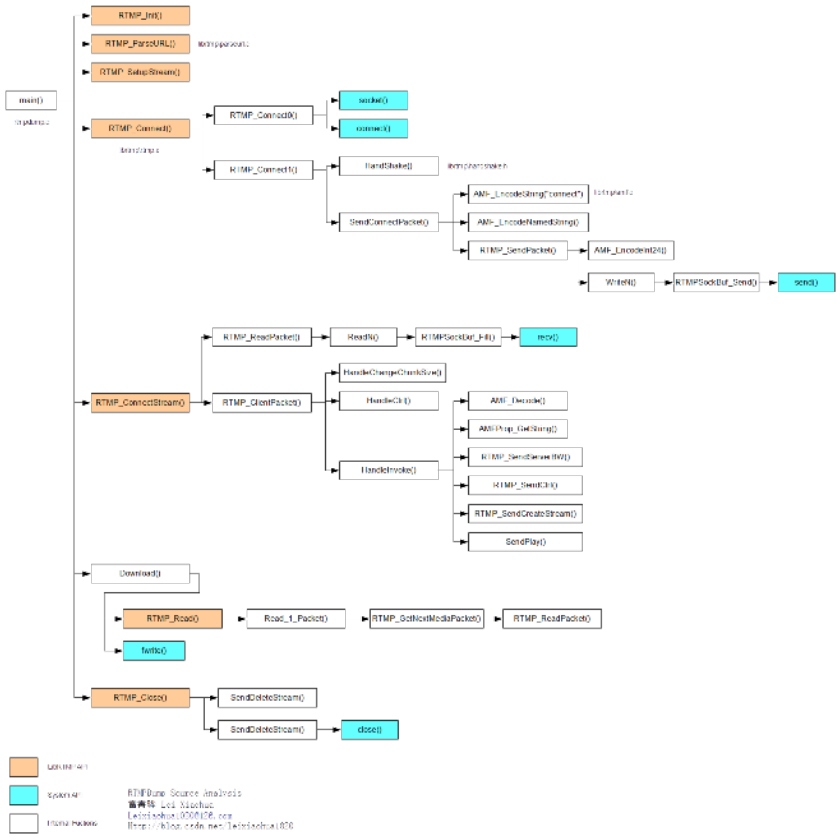
RTMPdump (libRTMP) 源代码分析9： 接收消息 (Message) (接收视音频数据)

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

之前写了一系列的文章介绍RTMPDump各种函数。比如怎么建立网络连接（NetConnection），怎么建立网络流（NetStream）之类的，唯独没有介绍这些发送或接收

的数据，在底层到底是怎么实现的。本文就是要剖析一下其内部的实现。即这些消息（Message）到底是怎么发送和接收的。
先来看看发送消息吧。

- 发送connect命令使用函数SendConnectPacket()
- 发送createstream命令使用RTMP_SendCreateStream()
- 发送realeaseStream命令使用SendReleaseStream()
- 发送publish命令使用SendPublish()
- 发送deleteStream的命令使用SendDeleteStream()
- 发送pause命令使用RTMP_SendPause()

不再一一例举，发现函数命名有两种规律：RTMP_Send***()或者Send***()，其中*号代表命令的名称。

SendConnectPacket()这个命令是每次程序开始运行的时候发送的第一个命令消息，内容比较多，包含了很多AMF编码的内容，在此不多做分析，贴上代码：

```cpp
//发送“connect”命令
static int
SendConnectPacket(RTMP *r, RTMPPacket *cp)
{
  RTMPPacket packet;
  char pbuf[4096], *pend = pbuf + sizeof(pbuf);
  char *enc;

  if (cp)
    return RTMP_SendPacket(r, cp, TRUE);

  packet.m_nChannel = 0x03; /* control channel (invoke) */
  packet.m_headerType = RTMP_PACKET_SIZE_LARGE;
  packet.m_packetType = 0x14;   /* INVOKE */
  packet.m_nTimeStamp = 0;
  packet.m_nInfoField2 = 0;
  packet.m_hasAbsTimestamp = 0;
  packet.m_body = pbuf + RTMP_MAX_HEADER_SIZE;

  enc = packet.m_body;
  enc = AMF_EncodeString(enc, pend, &av_connect);
  enc = AMF_EncodeNumber(enc, pend, ++r->m_numInvokes);
  *enc++ = AMF_OBJECT;

  enc = AMF_EncodeNamedString(enc, pend, &av_app, &r->Link.app);
  if (!enc)
    return FALSE;
  if (r->Link.protocol & RTMP_FEATURE_WRITE)
    {
      enc = AMF_EncodeNamedString(enc, pend, &av_type, &av_nonprivate);
      if (!enc)
    return FALSE;
    }
  if (r->Link.flashVer.av_len)
    {
      enc = AMF_EncodeNamedString(enc, pend, &av_flashVer, &r->Link.flashVer);
      if (!enc)
    return FALSE;
    }
  if (r->Link.swfUrl.av_len)
    {
      enc = AMF_EncodeNamedString(enc, pend, &av_swfUrl, &r->Link.swfUrl);
      if (!enc)
    return FALSE;
    }
  if (r->Link.tcUrl.av_len)
    {
      enc = AMF_EncodeNamedString(enc, pend, &av_tcUrl, &r->Link.tcUrl);
      if (!enc)
    return FALSE;
    }
  if (!(r->Link.protocol & RTMP_FEATURE_WRITE))
    {
      enc = AMF_EncodeNamedBoolean(enc, pend, &av_fpad, FALSE);
      if (!enc)
    return FALSE;
      enc = AMF_EncodeNamedNumber(enc, pend, &av_capabilities, 15.0);
      if (!enc)
    return FALSE;
      enc = AMF_EncodeNamedNumber(enc, pend, &av_audioCodecs, r->m_fAudioCodecs);
      if (!enc)
    return FALSE;
      enc = AMF_EncodeNamedNumber(enc, pend, &av_videoCodecs, r->m_fVideoCodecs);
      if (!enc)
    return FALSE;
      enc = AMF_EncodeNamedNumber(enc, pend, &av_videoFunction, 1.0);
      if (!enc)
    return FALSE;
      if (r->Link.pageUrl.av_len)
```

```
    {
      enc = AMF_EncodeNamedString(enc, pend, &av_pageUrl, &r->Link.pageUrl);
      if (!enc)
        return FALSE;
    }
    }
  if (r->m_fEncoding != 0.0 || r->m_bSendEncoding)
    {   /* AMF0, AMF3 not fully supported yet */
      enc = AMF_EncodeNamedNumber(enc, pend, &av_objectEncoding, r->m_fEncoding);
      if (!enc)
    return FALSE;
    }
  if (enc + 3 >= pend)
    return FALSE;
  *enc++ = 0;
  *enc++ = 0;           /* end of object - 0x00 0x00 0x09 */
  *enc++ = AMF_OBJECT_END;

  /* add auth string */
  if (r->Link.auth.av_len)
    {
      enc = AMF_EncodeBoolean(enc, pend, r->Link.lFlags & RTMP_LF_AUTH);
      if (!enc)
    return FALSE;
      enc = AMF_EncodeString(enc, pend, &r->Link.auth);
      if (!enc)
    return FALSE;
    }
  if (r->Link.extras.o_num)
    {
      int i;
      for (i = 0; i < r->Link.extras.o_num; i++)
    {
      enc = AMFProp_Encode(&r->Link.extras.o_props[i], enc, pend);
      if (!enc)
        return FALSE;
    }
    }
  packet.m_nBodySize = enc - packet.m_body;
  //----------------
  r->dlg->AppendMLInfo(20,1,"命令消息","Connect");
  //----------------------------
  return RTMP_SendPacket(r, &packet, TRUE);
}
```

RTMP_SendCreateStream()命令相对而言比较简单，代码如下：

```cpp
//发送“createstream”命令
int
RTMP_SendCreateStream(RTMP *r)
{
  RTMPPacket packet;
  char pbuf[256], *pend = pbuf + sizeof(pbuf);
  char *enc;

  packet.m_nChannel = 0x03; /* control channel (invoke) */
  packet.m_headerType = RTMP_PACKET_SIZE_MEDIUM;
  packet.m_packetType = 0x14;   /* INVOKE */
  packet.m_nTimeStamp = 0;
  packet.m_nInfoField2 = 0;
  packet.m_hasAbsTimestamp = 0;
  packet.m_body = pbuf + RTMP_MAX_HEADER_SIZE;

  enc = packet.m_body;
  enc = AMF_EncodeString(enc, pend, &av_createStream);
  enc = AMF_EncodeNumber(enc, pend, ++r->m_numInvokes);
  *enc++ = AMF_NULL;        /* NULL */

  packet.m_nBodySize = enc - packet.m_body;
  //----------------
  r->dlg->AppendMLInfo(20,1,"命令消息","CreateStream");
  //----------------------------
  return RTMP_SendPacket(r, &packet, TRUE);
}
```

同样，SendReleaseStream()内容也比较简单，我对其中部分内容作了注释：

```cpp
//发送RealeaseStream命令
static int
SendReleaseStream(RTMP *r)
{
  RTMPPacket packet;
  char pbuf[1024], *pend = pbuf + sizeof(pbuf);
  char *enc;

  packet.m_nChannel = 0x03; /* control channel (invoke) */
  packet.m_headerType = RTMP_PACKET_SIZE_MEDIUM;
  packet.m_packetType = 0x14;   /* INVOKE */
  packet.m_nTimeStamp = 0;
  packet.m_nInfoField2 = 0;
  packet.m_hasAbsTimestamp = 0;
  packet.m_body = pbuf + RTMP_MAX_HEADER_SIZE;

 enc = packet.m_body;
  //对“releaseStream”字符串进行AMF编码
  enc = AMF_EncodeString(enc, pend, &av_releaseStream);
  //对传输ID（0）进行AMF编码？
  enc = AMF_EncodeNumber(enc, pend, ++r->m_numInvokes);
  //命令对象
  *enc++ = AMF_NULL;
  //对播放路径字符串进行AMF编码
  enc = AMF_EncodeString(enc, pend, &r->Link.playpath);
  if (!enc)
    return FALSE;

  packet.m_nBodySize = enc - packet.m_body;
  //----------------
  r->dlg->AppendMLInfo(20,1,"命令消息","ReleaseStream");
  //----------------------------
  return RTMP_SendPacket(r, &packet, FALSE);
}
```

再来看一个SendPublish()函数，用于发送“publish”命令

```cpp
//发送Publish命令
static int
SendPublish(RTMP *r)
{
  RTMPPacket packet;
  char pbuf[1024], *pend = pbuf + sizeof(pbuf);
  char *enc;
  //块流ID为4
  packet.m_nChannel = 0x04; /* source channel (invoke) */
  packet.m_headerType = RTMP_PACKET_SIZE_LARGE;
  //命令消息,类型20
  packet.m_packetType = 0x14;   /* INVOKE */
  packet.m_nTimeStamp = 0;
  //流ID
  packet.m_nInfoField2 = r->m_stream_id;
  packet.m_hasAbsTimestamp = 0;
  packet.m_body = pbuf + RTMP_MAX_HEADER_SIZE;
  //指向Chunk的负载
  enc = packet.m_body;
   //对“publish”字符串进行AMF编码
  enc = AMF_EncodeString(enc, pend, &av_publish);
  enc = AMF_EncodeNumber(enc, pend, ++r->m_numInvokes);
  //命令对象为空
  *enc++ = AMF_NULL;
  enc = AMF_EncodeString(enc, pend, &r->Link.playpath);
  if (!enc)
    return FALSE;

  /* FIXME: should we choose live based on Link.lFlags & RTMP_LF_LIVE? */
  enc = AMF_EncodeString(enc, pend, &av_live);
  if (!enc)
    return FALSE;

  packet.m_nBodySize = enc - packet.m_body;
  //----------------
  r->dlg->AppendMLInfo(20,1,"命令消息","Pulish");
  //----------------------------
  return RTMP_SendPacket(r, &packet, TRUE);
}
```

其他的命令不再一一例举，总体的思路是声明一个RTMPPacket类型的结构体，然后设置各种属性值，最后交给RTMP_SendPacket()进行发送。

RTMPPacket类型的结构体定义如下，一个RTMPPacket对应RTMP协议规范里面的一个块（Chunk）。

```cpp
//Chunk信息
  typedef struct RTMPPacket
  {
    uint8_t m_headerType;//ChunkMsgHeader的类型（4种）
    uint8_t m_packetType;//Message type ID（1-7协议控制；8，9音视频；10以后为AMF编码消息）
    uint8_t m_hasAbsTimestamp;  /* Timestamp 是绝对值还是相对值? */
    int m_nChannel;          //块流ID
    uint32_t m_nTimeStamp;  // Timestamp
    int32_t m_nInfoField2;  /* last 4 bytes in a long header,消息流ID */
    uint32_t m_nBodySize;   //消息长度
    uint32_t m_nBytesRead;
    RTMPChunk *m_chunk;
    char *m_body;
  } RTMPPacket;
```

下面我们来看看RTMP_SendPacket()吧，各种的RTMPPacket（即各种Chunk）都需要用这个函数进行发送。

```cpp
//自己编一个数据报发送出去！
//非常常用
int
RTMP_SendPacket(RTMP *r, RTMPPacket *packet, int queue)
{
  const RTMPPacket *prevPacket = r->m_vecChannelsOut[packet->m_nChannel];
  uint32_t last = 0;
  int nSize;
  int hSize, cSize;
  char *header, *hptr, *hend, hbuf[RTMP_MAX_HEADER_SIZE], c;
  uint32_t t;
  char *buffer, *tbuf = NULL, *toff = NULL;
  int nChunkSize;
  int tlen;
  //不是完整ChunkMsgHeader
  if (prevPacket && packet->m_headerType != RTMP_PACKET_SIZE_LARGE)
    {
      /* compress a bit by using the prev packet's attributes */
    //获取ChunkMsgHeader的类型
    //前一个Chunk和这个Chunk对比
      if (prevPacket->m_nBodySize == packet->m_nBodySize
      && prevPacket->m_packetType == packet->m_packetType
      && packet->m_headerType == RTMP_PACKET_SIZE_MEDIUM)
    packet->m_headerType = RTMP_PACKET_SIZE_SMALL;


      if (prevPacket->m_nTimeStamp == packet->m_nTimeStamp
      && packet->m_headerType == RTMP_PACKET_SIZE_SMALL)
    packet->m_headerType = RTMP_PACKET_SIZE_MINIMUM;
      //上一个packet的TimeStamp
      last = prevPacket->m_nTimeStamp;
    }

  if (packet->m_headerType > 3)   /* sanity */
    {
      RTMP_Log(RTMP_LOGERROR, "sanity failed!! trying to send header of type: 0x%02x.",
      (unsigned char)packet->m_headerType);
      return FALSE;
    }
  //chunk包头大小；packetSize[] = { 12, 8, 4, 1 }
  nSize = packetSize[packet->m_headerType];
  hSize = nSize; cSize = 0;
  //相对的TimeStamp
  t = packet->m_nTimeStamp - last;

  if (packet->m_body)
    {
    //Header的Start
    //m_body是指向负载数据首地址的指针；“-”号用于指针前移
      header = packet->m_body - nSize;
    //Header的End
      hend = packet->m_body;
    }
  else
    {
      header = hbuf + 6;
      hend = hbuf + sizeof(hbuf);
    }
  //当ChunkStreamID大于319时
  if (packet->m_nChannel > 319)
    //ChunkBasicHeader是3个字节
    cSize = 2;
  //当ChunkStreamID大于63时
  else if (packet->m_nChannel > 63)
    //ChunkBasicHeader是2个字节
    cSize = 1;
  if (cSize)
    {
    //header指针指向ChunkMsgHeader
```

```
      header -= cSize;
    //hsize加上ChunkBasicHeader的长度
      hSize += cSize;
    }
  //相对TimeStamp大于0xffffff，此时需要使用ExtendTimeStamp
  if (nSize > 1 && t >= 0xffffff)
    {
      header -= 4;
      hSize += 4;
    }

  hptr = header;
  //把ChunkBasicHeader的Fmt类型左移6位
  c = packet->m_headerType << 6;
  switch (cSize)
    {
    //把ChunkBasicHeader的低6位设置成ChunkStreamID
    case 0:
      c |= packet->m_nChannel;
      break;
    //同理，但低6位设置成000000
    case 1:
      break;
    //同理，但低6位设置成000001
    case 2:
      c |= 1;
      break;
    }
  //可以拆分成两句*hptr=c;hptr++，此时hptr指向第2个字节
  *hptr++ = c;
  //CSize>0，即ChunkBasicHeader大于1字节
  if (cSize)
    {
    //将要放到第2字节的内容tmp
      int tmp = packet->m_nChannel - 64;
    //获取低位存储与第2字节
      *hptr++ = tmp & 0xff;
    //ChunkBasicHeader是最大的3字节时
      if (cSize == 2)
    //获取高位存储于最后1个字节（注意：排序使用大端序列，和主机相反）
    *hptr++ = tmp >> 8;
    }
  //ChunkMsgHeader。注意一共有4种，包含的字段数不同。
  //TimeStamp(3B)
  if (nSize > 1)
    {
    //相对TimeStamp和绝对TimeStamp?
      hptr = AMF_EncodeInt24(hptr, hend, t > 0xffffff ? 0xffffff : t);
    }
  //MessageLength+MessageTypeID(4B)
  if (nSize > 4)
    {
    //MessageLength
      hptr = AMF_EncodeInt24(hptr, hend, packet->m_nBodySize);
    //MessageTypeID
      *hptr++ = packet->m_packetType;
    }
  //MessageStreamID(4B)
  if (nSize > 8)
    hptr += EncodeInt32LE(hptr, packet->m_nInfoField2);

  //ExtendedTimeStamp
  if (nSize > 1 && t >= 0xffffff)
    hptr = AMF_EncodeInt32(hptr, hend, t);
  //负载长度，指向负载的指针
  nSize = packet->m_nBodySize;
  buffer = packet->m_body;
  //Chunk大小，默认128字节
  nChunkSize = r->m_outChunkSize;

  RTMP_Log(RTMP_LOGDEBUG2, "%s: fd=%d, size=%d", __FUNCTION__, r->m_sb.sb_socket,
      nSize);
  /* send all chunks in one HTTP request */
  //使用HTTP
  if (r->Link.protocol & RTMP_FEATURE_HTTP)
    {
    //nSize:Message负载长度；nChunkSize：Chunk长度；
    //例nSize：307，nChunkSize:128；
    //可分为（307+128-1）/128=3个
    //为什么+nChunkSize-1？因为除法会只取整数部分！
      int chunks = (nSize+nChunkSize-1) / nChunkSize;
    //Chunk个数超过一个
      if (chunks > 1)
        {
    //注意：CSize=1表示ChunkBasicHeader是2字节
    //消息分n块后总的开销：
    //n个ChunkBasicHeader，1个ChunkMsgHeader，1个Message负载
    //实际中只有第一个Chunk是完整的，剩下的只有ChunkBasicHeader
      tlen = chunks * (cSize + 1) + nSize + hSize;
    //分配内存
      tbuf = (char *) malloc(tlen);
      if (!tbuf)
```

```
      if (!tbuf)
        return FALSE;
      toff = tbuf;
    }
    //消息的负载+头
    }
  while (nSize + hSize)
    {
      int wrote;
      //消息负载<Chunk大小（不用分块）
      if (nSize < nChunkSize)
    //Chunk可能小于设定值
    nChunkSize = nSize;

      RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)header, hSize);
      RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)buffer, nChunkSize);
      if (tbuf)
        {
    //void *memcpy(void *dest, const void *src, int n);
    //由src指向地址为起始地址的连续n个字节的数据复制到以dest指向地址为起始地址的空间内
      memcpy(toff, header, nChunkSize + hSize);
      toff += nChunkSize + hSize;
    }
      else
        {
      wrote = WriteN(r, header, nChunkSize + hSize);
      if (!wrote)
        return FALSE;
    }
      //消息负载长度-Chunk负载长度
      nSize -= nChunkSize;
      //Buffer指针前移1个Chunk负载长度
      buffer += nChunkSize;
      hSize = 0;

      //如果消息没有发完
      if (nSize > 0)
    {
    //ChunkBasicHeader
      header = buffer - 1;
      hSize = 1;
      if (cSize)
        {
          header -= cSize;
          hSize += cSize;
        }
      //ChunkBasicHeader第1个字节
      *header = (0xc0 | c);
      //ChunkBasicHeader大于1字节
      if (cSize)
        {
          int tmp = packet->m_nChannel - 64;
          header[1] = tmp & 0xff;
          if (cSize == 2)
        header[2] = tmp >> 8;
        }
    }
    }
  if (tbuf)
    {
    //
      int wrote = WriteN(r, tbuf, toff-tbuf);
      free(tbuf);
      tbuf = NULL;
      if (!wrote)
        return FALSE;
    }

  /* we invoked a remote method */
  if (packet->m_packetType == 0x14)
    {
      AVal method;
      char *ptr;
      ptr = packet->m_body + 1;
      AMF_DecodeString(ptr, &method);
      RTMP_Log(RTMP_LOGDEBUG, "Invoking %s", method.av_val);
      /* keep it in call queue till result arrives */
      if (queue) {
        int txn;
        ptr += 3 + method.av_len;
        txn = (int)AMF_DecodeNumber(ptr);
    AV_queue(&r->m_methodCalls, &r->m_numCalls, &method, txn);
      }
    }

  if (!r->m_vecChannelsOut[packet->m_nChannel])
    r->m_vecChannelsOut[packet->m_nChannel] = (RTMPPacket *) malloc(sizeof(RTMPPacket));
  memcpy(r->m_vecChannelsOut[packet->m_nChannel], packet, sizeof(RTMPPacket));
  return TRUE;
}
```

这个函数乍一看好像非常复杂，其实不然，他只是按照RTMP规范将数据编码成符合规范的块（Chunk），规范可以参考相关的文档。

具体怎么编码成块（Chunk）就不多分析了，在这里需要注意一个函数：WriteN()。该函数完成了将数据发送出去的功能。

来看一下WriteN()函数：

```cpp
//发送数据报的时候调用（连接，buffer，长度）
static int
WriteN(RTMP *r, const char *buffer, int n)
{
  const char *ptr = buffer;
#ifdef CRYPTO
  char *encrypted = 0;
  char buf[RTMP_BUFFER_CACHE_SIZE];

  if (r->Link.rc4keyOut)
    {
      if (n > sizeof(buf))
    encrypted = (char *)malloc(n);
      else
    encrypted = (char *)buf;
      ptr = encrypted;
      RC4_encrypt2((RC4_KEY *)r->Link.rc4keyOut, n, buffer, ptr);
    }
#endif

  while (n > 0)
    {
      int nBytes;
      //因方式的不同而调用不同函数
      //如果使用的是HTTP协议进行连接
      if (r->Link.protocol & RTMP_FEATURE_HTTP)
        nBytes = HTTP_Post(r, RTMPT_SEND, ptr, n);
      else
        nBytes = RTMPSockBuf_Send(&r->m_sb, ptr, n);
      /*RTMP_Log(RTMP_LOGDEBUG, "%s: %d\n", __FUNCTION__, nBytes); */
      //成功发送字节数<0
      if (nBytes < 0)
    {
      int sockerr = GetSockError();
      RTMP_Log(RTMP_LOGERROR, "%s, RTMP send error %d (%d bytes)", __FUNCTION__,
          sockerr, n);

      if (sockerr == EINTR && !RTMP_ctrlC)
        continue;

      RTMP_Close(r);
      n = 1;
      break;
    }

      if (nBytes == 0)
    break;

      n -= nBytes;
      ptr += nBytes;
    }

#ifdef CRYPTO
  if (encrypted && encrypted != buf)
    free(encrypted);
#endif

  return n == 0;
}
```

该函数中，RTMPSockBuf_Send()完成了数据发送的功能，再来看看这个函数（函数调用真是好多啊。。。。）

```cpp
//Socket发送（指明套接字，buffer缓冲区，数据长度）
//返回所发数据量
int
RTMPSockBuf_Send(RTMPSockBuf *sb, const char *buf, int len)
{
  int rc;

#ifdef _DEBUG
  fwrite(buf, 1, len, netstackdump);
#endif

#if defined(CRYPTO) && !defined(NO_SSL)
  if (sb->sb_ssl)
    {
      rc = TLS_write((SSL *)sb->sb_ssl, buf, len);
    }
  else
#endif
    {
    //向一个已连接的套接口发送数据。
    //int send( SOCKET s, const char * buf, int len, int flags);
    //s：一个用于标识已连接套接口的描述字。
    //buf：包含待发送数据的缓冲区。
    //len：缓冲区中数据的长度。
    //flags：调用执行方式。
    //rc:所发数据量。
      rc = send(sb->sb_socket, buf, len, 0);
    }
  return rc;
}

int
RTMPSockBuf_Close(RTMPSockBuf *sb)
{
#if defined(CRYPTO) && !defined(NO_SSL)
  if (sb->sb_ssl)
    {
      TLS_shutdown((SSL *)sb->sb_ssl);
      TLS_close((SSL *)sb->sb_ssl);
      sb->sb_ssl = NULL;
    }
#endif
  return closesocket(sb->sb_socket);
}
```

到这个函数的时候，发现一层层的调用终于完成了，最后调用了系统Socket的send()函数完成了数据的发送功能。

之前贴过一张图总结这个过程，可能理解起来要方便一些： RTMPDump源代码分析 0： 主要函数调用分析

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 消息 源代码 send

个人分类： libRTMP

所属专栏： 开源多媒体项目源代码分析