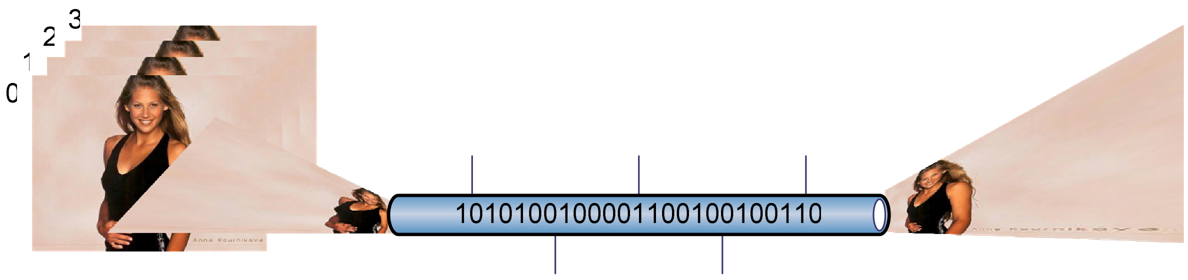
IP Protocols



RTP
(Real Time Protocol)

Jordi Cenzano Ferret
2008-01-05

Index

1 Introduction

This work is divided in four distinct sections. In the first one we explain the Real Time Protocol (RTP)[1] described in RFC3550[3]. In the second one we analyze the part of RTP that is used to send MPEG2 transport streams; this part is described in RFC2250[5].

In section number three we develop a C application in order to demonstrate the two previous sections. And in the last section we extract the conclusions.

2 Real Time Protocol (RFC3550)

The RTP provides end to end network transport functions; it can be used for applications that transmits real time data, such as video, audio or simulation data.

RTP can be used over multicast or unicast networks and does not guarantee quality of service (QoS).

RTP is designed to be independent of underlying network and transport layers.

Usually RTP uses Real Time Control Protocol (RTCP), described in RFC3550 as well, to get information about the quality of the data distribution. The use of RTCP is not mandatory.

The applications typically run RTP over UDP to make use of its multiplexing and checksum services. In the Figure 1 there is a protocol layering diagram [2].

The RTP adds to underling protocol (usually UDP) the following mechanism:

1- Sequence numbers: It can be used to detect packet loses and to reorder packets.
2- Timestamp: It allows the receiver to do jitter calculations.

The RTP is designed to be tailored through modifications and / or additions to the headers as needed, for example the RFC3551[4] defines a lot of different audio video extensions/profiles of RTP.

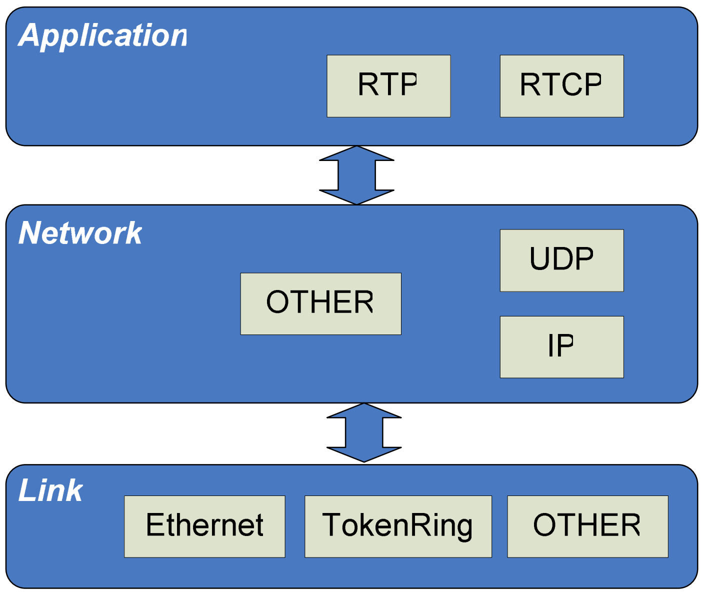


***Figure 1** (Protocol layering diagram)*

2.1 RTP Definitions

2.1.1 RTP Payload

It is the data transported by RTP packet.

2.1.2 RTP packet

A data RTP packet consists in a fixed header (see 2.3), a possible empty list of contributing sources, and the payload data. Typically a packet of underlying protocol contains only a single RTP packet, but several packets may be contained.

2.1.3 RTCP packet

It consists in a fixed header (see 2.4) followed by structured elements that vary depending upon the RTCP packet type.

2.1.4 RTP session

It is an association among a set of participants communicating with RTP.

2.1.5 Multimedia session

It is a set of concurrent RTP sessions among a common group of participants. For example, a videoconference, it may contain an audio RTP session and a video RTP session.

2.2 RTP Elements

The RFC3550 defines different RTP elements, in the Figure 2 we can see how are connected between them. In this section we describe some of them:

2.2.1 Source

Is the element that sends the RTP data; it can be live, like a video conference or stored, like an audio file.

2.2.1.1 Layered encoder

The layered encoder is a kind of source that is able to encode its data in a progressive way. The main idea is to transmit this progressive data in different RTP streams, this will allow the receivers to decode only a part of the stream depending of the bandwidth or depending on network congestion. The most common example is in video transmission, the layered encoder can send different RTP streams: basic bitrate, extension1, extension2, etc… And the receiver that has enough capacity to decode all the streams will have the best video quality, and the receiver that only can decode the basic bitrate stream will have the basic quality.

2.2.2 Destination

It shows or stores the data that sources are sending. The destination can be a source at the same time, full duplex transmission.

2.2.3 Mixer

It compounds one single stream from different source streams. It can be useful to communicate different multicast streams to an element in a network that doesn’t allow multicast. A mixer could mix all those multicast streams into a single unicast connection.

2.2.4 Monitor

An application that receives de RTCP packets sent by participants in a RTP session, it estimates the current quality of service. It can be implemented into the application that participates in the RTP Session, but also can be a separate application.

2.2.5 *Translator*

Translate / encapsulate RTP packets to / inside other protocol. For example if we want to connect to an element that is behind a firewall we can use two translators for tunneling this connection.

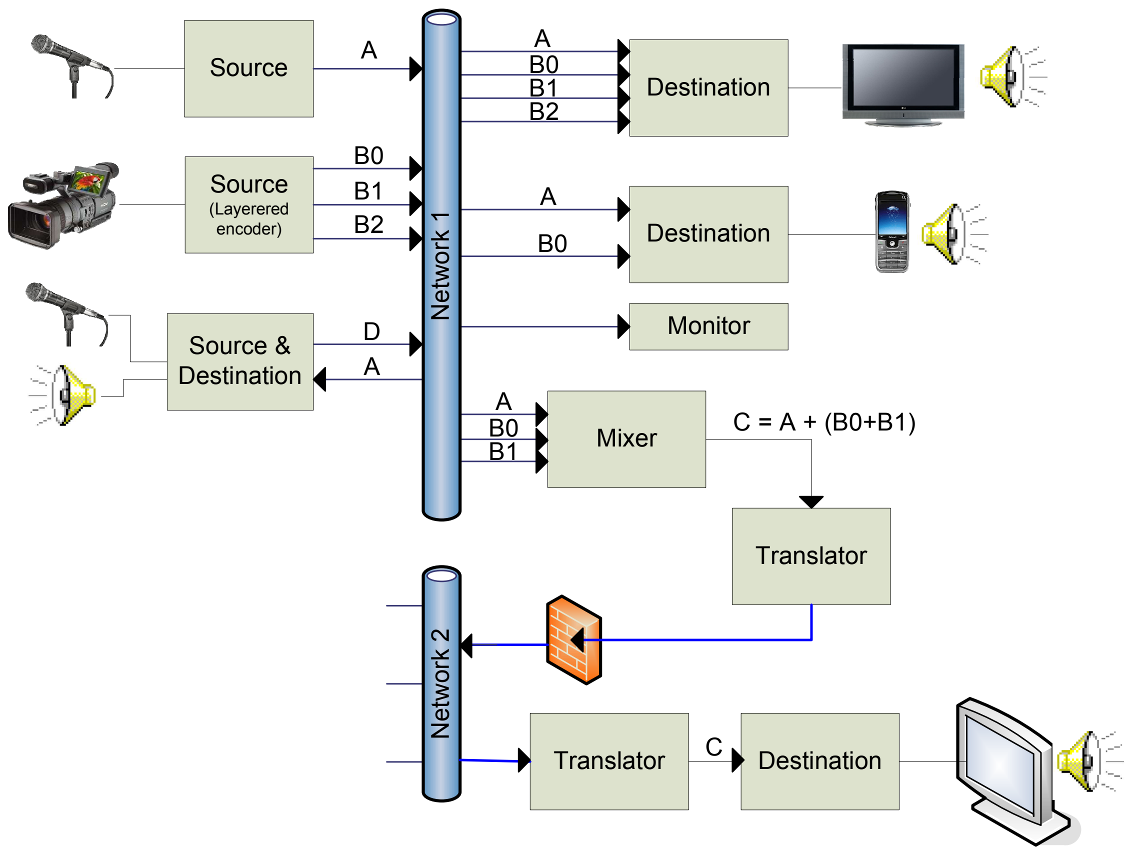


Figure 2 *(Element connections)*

2.3 RTP Packet syntax

In Figure 3 we can see the structure of RTP packet. We have to keep in mind that this RTP packet will be usually the payload of an UDP packet and it will use the multiplexing and checksum services given by UDP.

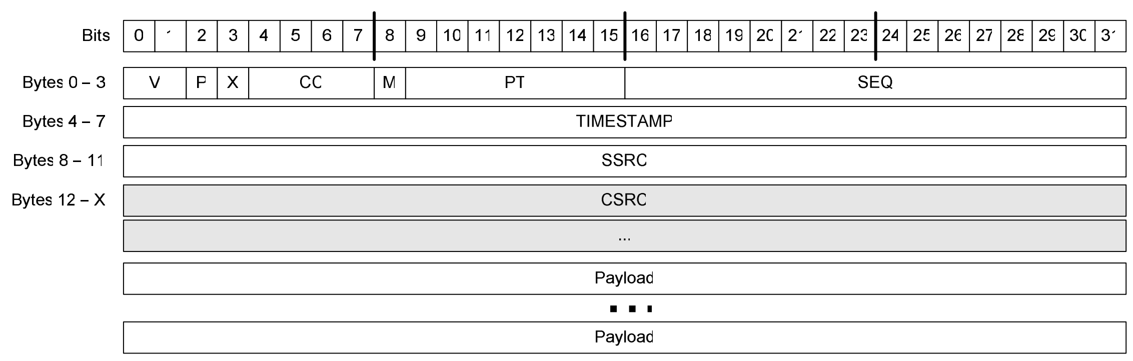


Figure 3 *(RTP Packet header syntax)*

- **V: Version** of the protocol, usually equal to 2.

- **P: Padding** indication. If is set means that the packet contains one or more padding packets at the end which are not part of the payload. The last byte of the padding contains the count of how many bytes have to be ignored.

- **X: Extension**. If is set indicates that the header of the Figure 3 is followed by the extension header (see Figure 4).

- **CC: Contains the number of CSRC** identifiers that follow the fixed header (it can be 0).

- **M: Marker.** The meaning of this bit is defined in the profile. It is only a flag.

- **PT: Payload type**. Indicate the format of the RTP payload. In RFC3551 there is a default mapping from PT to audio/video payloads formats. A receiver must ignore the PT doesn’t understand.

- **SEQ: Sequence number.** It increments by one for each RTP packet sent. This allows to the receivers to detect packet loses, and restores packet sequence.

- **Timestamp:** Is the sampling instant of the first byte in the payload. It allows synchronization and jitter calculation to the receiver. The format can change depending on profile.

- **SSRC: Synchronization source**. A random number that identifies the packets that belong to the same timing and sequence number space.

- **CSRC: Contribution source**. This field is filled for the mixers (See 2.2.3) and means all SSRC that are mixed to compose that packet.

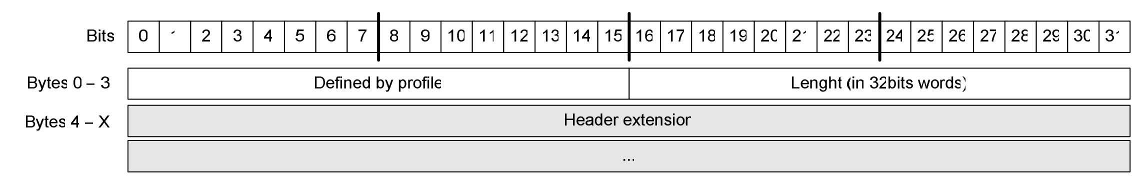


Figure 4 *(RTP Packet extension header syntax)*

2.4 *Real Time Control Protocol (RTCP)*

RTCP is based on the periodic transmission of control packets to all participants in the RTP session, using the same distribution mechanism as the data packets.

The RTCP performs four functions:

1- The primary function is to provide feedback on the quality of the data distribution. This feedback may be useful for control the adaptive encoders.

2- To carry a persistent transport level identifier for an RTP source called "canonical name" (CNAME). Is useful to keep track of each participant in front of changes of SRRC (it can change if the application restarts).
Receivers may also require the CNAME to associate to multiple RTP sessions from a given participant, for example to get audio and video from a videoconference (multimedia) session.

3- The first two functions require that all participants send~~s~~ RTCP packets, this function tries to control this rate.

4- The last function is to send some session control information, for example a user friendly name of each participant. This function is optional.

In the Figure 5 we can see that the syntax of RTCP packets is very similar to the RTP packets.

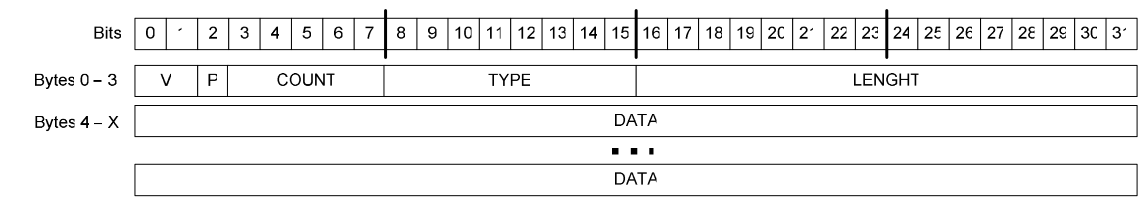


Figure 5 *(RTCP Packet syntax)*

- **V: Version** of the protocol, usually equal to 2.
- **P: Padding** indication. If is set means that the packet contains one or more padding packets at the end, which are not part of the payload. The last byte of the padding contains the count of how many bytes have to be ignored.
- **COUNT:** Contains the **number of report blocks** contained in this packet.
- **TYPE: Packet type.**
  - **SR: Sender report.** Transmits statistics from participants that are active senders.
  - **RR: Receiver report.** For reception statistics from participants that are not active senders.
  - **SDES: Source description**. Used to send source description items.
  - **BYE:** Indicates the end of participation.
  - **APP:** Application defined functions.

2.5 RTP Scenarios

In this section we show some examples in order to illustrate the basic operation of applications using RTP, remember that RTP protocol may be large adapted to a lot of a variety of circumstances.

In these examples we are assuming that RTP is used on top of UDP, and that it follows the conventions established in the audio and video profile specified in RFC3551.

2.5.1 Simple multicast audio conference

In the Figure 6 we can see the blocks diagram of a multicast audio conference. We can see that every client opens 2 multicast UDP address-port pairs, the first one is used to transmit RTP packets and the second one is used to send the control packets (RTCP). The distribution of address-port pair is not in the scope of RTP protocol.

The RTP packet header has to indicate the audio format, it could be PCMA. This format is described in RFC3551.

The encoders can change the encoding bitrate or format during the conference in order to adapt the rate to network bandwidth or to accommodate a new participant connected through a low bandwidth connection.

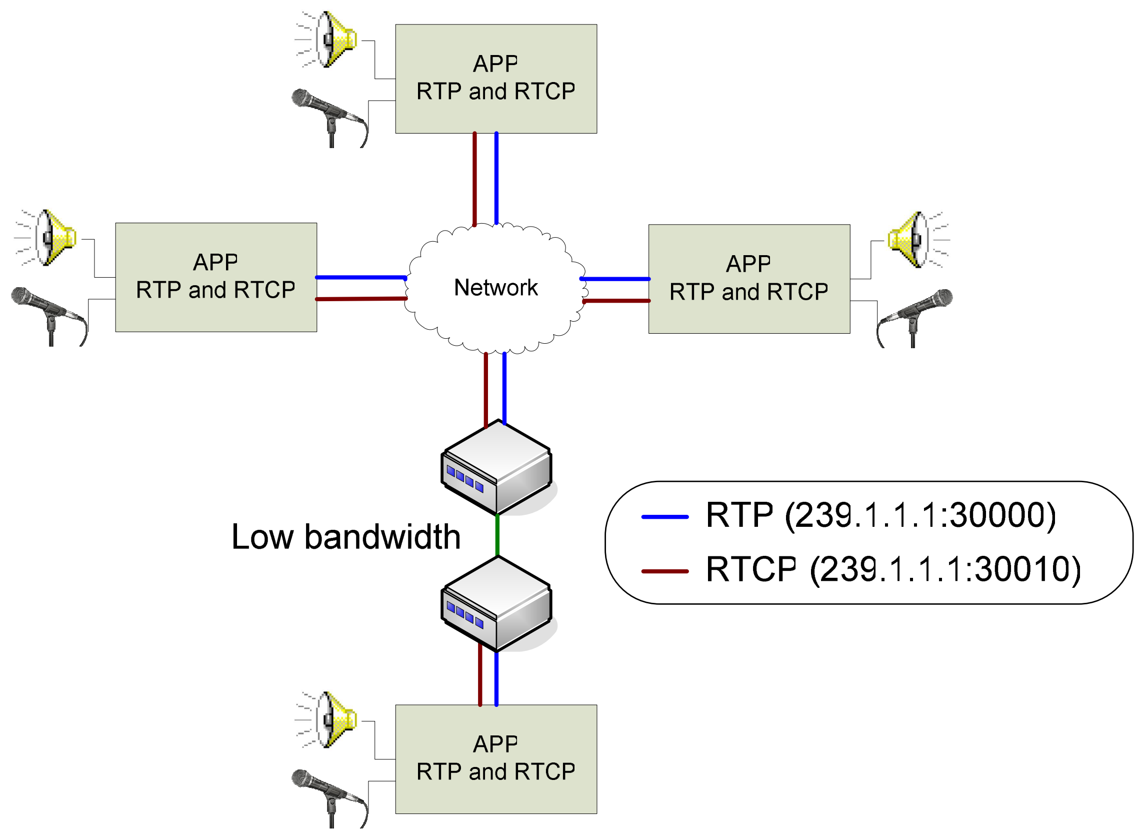


Figure 6 *(Multicast audio conference)*

2.5.2 Audio and video conference

In the Figure 7 there is a block diagram of a videoconference using RTP and RTCP. We can see that audio and video are transmitted as separate RTP sessions. There is no direct coupling at RTP level between the audio and video sessions. A user that is participating in both RTP sessions (audio and video) can use the CNAME of RTCP to associate both RTP sessions. The audio / video synchronization can be archived using timing information carried in RTP packets.

A participant that only wants to use RTP audio session, due to low bandwidth for example, can do that joining only to RTP audio session.

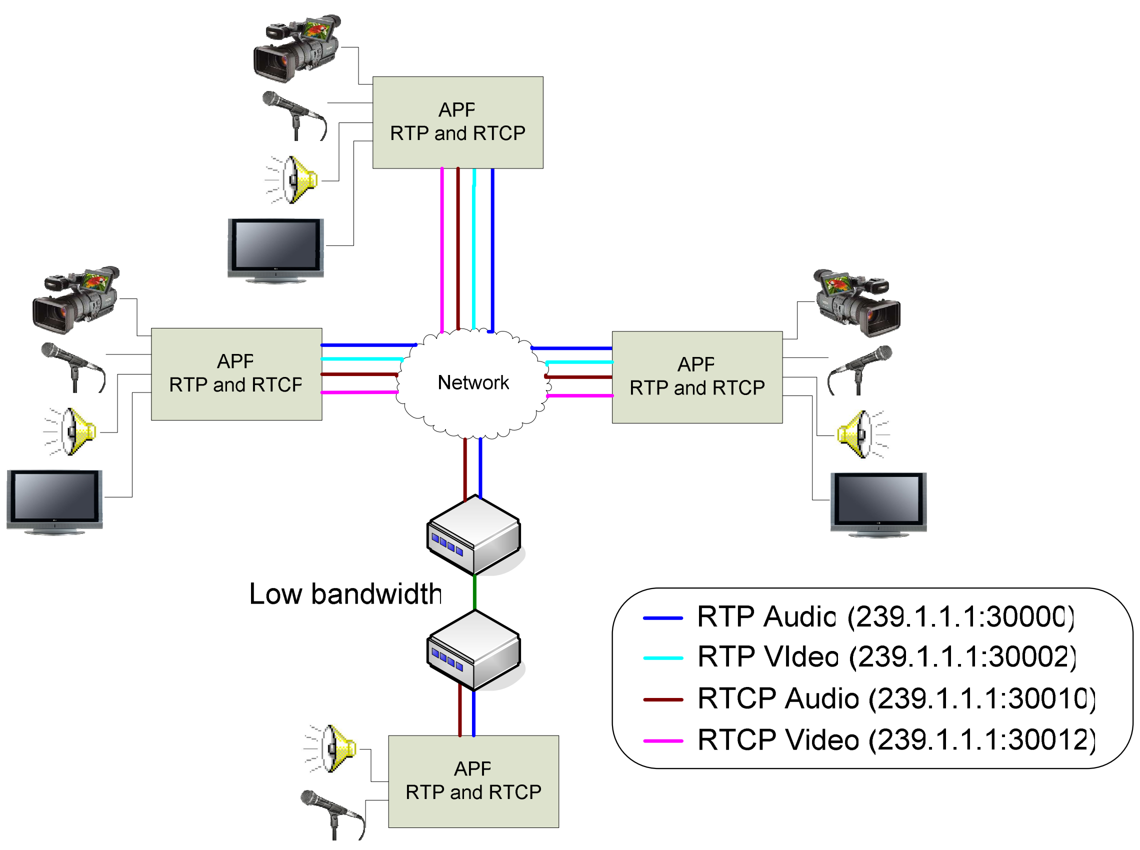

APF
RTP and RTCP
APF
RTP and RTCF
Network
APF
RTP and RTCP
Low bandwidth
APF
RTP and RTCP
RTP Audio (239.1.1.1:30000)
RTP VIdeo (239.1.1.1:30002)
RTCP Audio (239.1.1.1:30010)
RTCP Video (239.1.1.1:30012)

Figure 7 *(Audio and video conference)*

2.6 RTP Payload for MPEG1 / MPEG2

The payload format for mpeg video and audio streams is defined in RFC2250. It describes two approaches to encapsulate mpeg streams.

To understand properly the two packetization approaches we need to introduce the MPEG encapsulation.

The MPEG1 specification is defined in three parts: system, video and audio. The video and audio portions of the specification describe the basic format of the video or audio stream. Video and audio are encoded into elementary streams (ES) and these streams are packetized and encapsulated into a MPEG program stream (MPS).

The MPEG2 specification is structured in a similar way. The MPEG2 System specification defines two system stream formats: The MPEG program stream (MPS), the same as MPEG1, and the MPEG2 Transport Stream (MTS). The MTS is tailored for communicating or storing one or more programs of MPEG compressed data and also other data in relatively error-prone environments. The MPS is tailored for relatively error-free environments.

The RFC2250 defines these two encapsulation schemes:

1- Encapsulation of MPS and MTS streams.

2- Encapsulation of MPEG ES.

In this work we focus into the first encapsulation scheme (system and transport streams).

2.6.1 Encapsulation of MPEG streams

In this section we describe how we have to parse the MPEG system or transport stream to encapsulate it into RTP packets.

2.6.1.1 Encapsulation of MPEG system streams

We have to extract the Program Clock Reference (PCR) or System Clock reference (STC) and put it inside the timestamp field of the RTP header. There are no packetization restrictions, these streams are treated as a stream of bytes. See Figure 8.

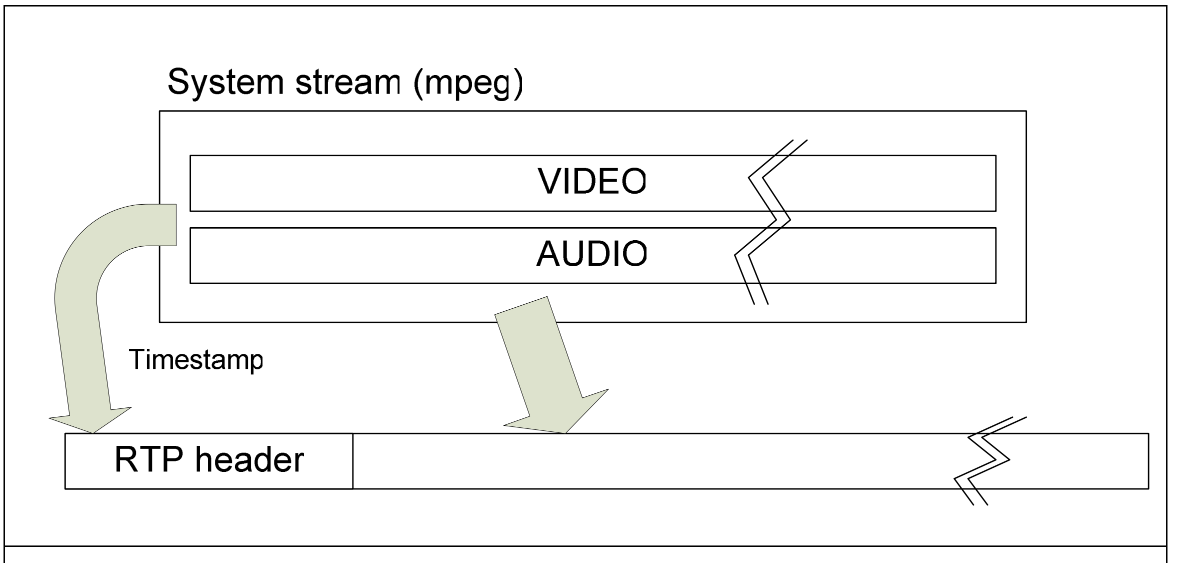


Figure 8 *(RTP encapsulation of mpeg system streams)*

The M (marker) bit must be 0, unless you switch from one video/audio sequence to another. It is used as discontinuity indication.

2.6.1.2 Encapsulation of MPEG transport streams

It has to implement the same timestamping method as in system streams. We have to extract the PCR or STC and put it inside the timestamp field of the RTP header.

In this case the RTP payload must contain an integral number of MTS packets (usually 188 bytes each). See *Figure 9*.

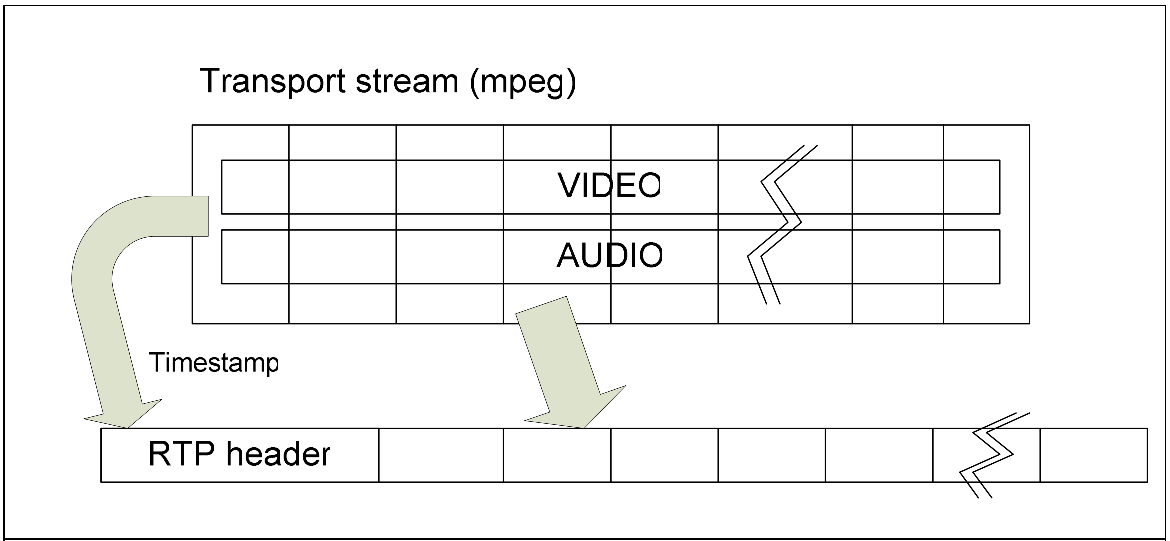


Figure 9 *(RTP encapsulation of mpeg transport streams)*

In this case the payload type of RTP header must be set to 33. And the M (marker) bit must be 0, unless you switch from one video/audio sequence to another.

3 RTP source test application

In order to practice to manage RTP protocol we have implemented in C a RTP source of MTS. The key idea is to learn the details of RTP protocol, and how to manage the application programming interfaces (APIs) of RTP underlying protocols such UDP (sockets).

The developed application reads an MTS file, extracts its timing information, opens a multicast socket, encapsulates MPEG file into RTP packets and ~~it~~ sends the RTP packets through the UDP socket according to the extracted timing information. In the Figure 10 there is an application flow chart.

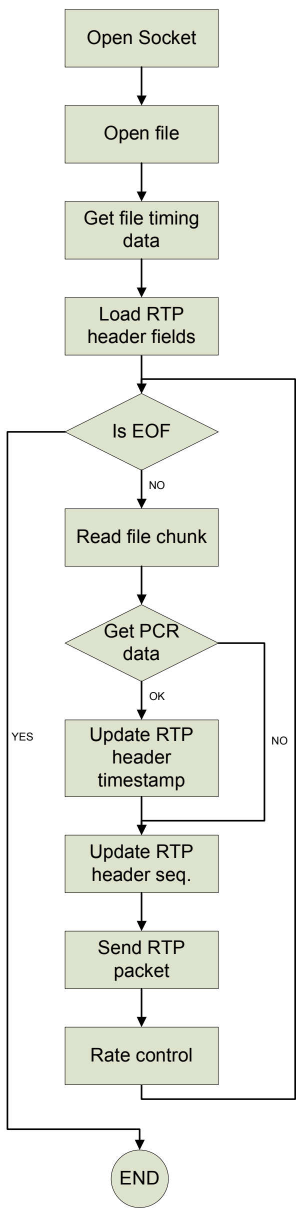


Figure 10 *(RTP source test flow chart)*

3.1 Application code

In this section we analyze the C code of RTP source test application. Doing that we can realize ~~of~~ the real scope of RTP protocol, like syntax, timing, behavior, etc…

We follow the flow chart (Figure 10) to analyze the code.

3.1.1 Open socket

We open a socket in UDP mode, and we load the IPaddr-port destination (usually multicast). See Code 1.

```
m_nSck = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
if (m_nSck == INVALID_SOCKET)
{
      printf ("Error opening socket\n");
      return FALSE;
}

//Set socket addr
ZeroMemory((void*)&m_groupSock,sizeof(sockaddr_in));
m_groupSock.sin_family = AF_INET;
m_groupSock.sin_addr.s_addr = inet_addr(m_szIP); //239.1.1.1
m_groupSock.sin_port = htons(m_nPort); //1234
```

Code 1 *(Open UDP socket)*

3.1.2 Open file

We open a MPEG2 transport stream file for read. See Code 2.

```
//Open File
m_hFile = CreateFile(m_szFile,GENERIC_READ,FILE_SHARE_READ,NULL,
OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);

if (m_hFile == INVALID_HANDLE_VALUE)
{
      printf ("Error CreateFile: %s\n",m_szFile);
      return FALSE;
}
```

Code 2 *(Open file for read)*

3.1.3 Load RTP header fields

In this part of the application we load all RTP header parameters following the rules of RFC3550. In this application all fields of the header will remain invariable except the sequence number and the timestamp. See *Code 3*.

```
int  CRTPSrc::LoadRtpHeader  (BYTE  *pData,int  nDataLen,int  nVer2,int
nPadding, int nExtension,int nCSRCLen,int nMarker,int nPayloadType,int
nSeqNumber,unsigned  long  lTstamp,unsigned  long  lSSRC,unsigned  long
*plCSRC)
{
      if (pData == NULL){return -1;}
      if (nDataLen < 12){return -1;}

      BYTE bTmp = 0;
      int nFisrtPayloadByte = 0;

      bTmp = (nVer2&0x3)<<6;
      bTmp = bTmp | ((nPadding<<5)&(0x1));
      bTmp = bTmp | ((nExtension<<4)&(0x1));
      bTmp = bTmp | ((nCSRCLen<<4)&0xF);
      pData[0] = bTmp;

      bTmp = 0;
      bTmp = (nMarker&0x1)<<7;
      bTmp = bTmp | (nPayloadType&(0x7F));
      pData[1] = bTmp;

      pData[2] = (BYTE) ((nSeqNumber&0xFF00)>>8);
      pData[3] = (BYTE) (nSeqNumber&0x00FF);

      pData[4] = (BYTE) ((lTstamp&0xFF000000)>>24);
      pData[5] = (BYTE) ((lTstamp&0x00FF0000)>>16);
      pData[6] = (BYTE) ((lTstamp&0x0000FF00)>>8);
      pData[7] = (BYTE) (lTstamp&0x000000FF);

      pData[8] = (BYTE) ((lSSRC&0xFF000000)>>24);
      pData[9] = (BYTE) ((lSSRC&0x00FF0000)>>16);
      pData[10] = (BYTE) ((lSSRC&0x0000FF00)>>8);
      pData[11] = (BYTE) (lSSRC&0x000000FF);

      nFisrtPayloadByte = 12;

      if ((plCSRC != NULL)&&(nCSRCLen > 0))
      {
            for (int n = 0; n < nCSRCLen; n++)
            {
                  if (nDataLen < (15 + n*4)){return -1;}

                  unsigned long lTmp = plCSRC[n];
                  pData[12 + n*4] = (BYTE) ((lTmp&0xFF000000)>>24);
                  pData[13 + n*4] = (BYTE) ((lTmp&0x00FF0000)>>16);
                  pData[14 + n*4] = (BYTE) ((lTmp&0x0000FF00)>>8);
                  pData[15 + n*4] = (BYTE) (lTmp&0x000000FF);
            }

            nFisrtPayloadByte = (15 + n*4) + 1;
      }

      return nFisrtPayloadByte;
}
```

***Code 3** (Load RTP header fields)*

3.1.4 Get file timing data

We start to read MTS packets (188 bytes each) until we found a packet with timing information. In order to do that we have implemented the *GetPCR* function. We save the first found PCR into variable *nTS1*. After that we continue searching the next MTS packet with timing information and we save that information into variable *nTS2*.

At this moment we know the first timestamp (*nTS1*), the second timestamp (*nTS2*) and the packets between them, and it is easy to calculate the output bitrate (in packets per second).

Finally we "rewind" the file.

```
BOOL        CRTPSrc::GetPacketIntervalByPCR        (unsigned        int
*pnPacktesPerSecond,__int64 *pnFirstPCR,__int64 *pnSecondPCR)
{
      BYTE bData[188];
      unsigned int nBytesToReadFromFile = 188;
      DWORD dwReaded = nBytesToReadFromFile;
      BOOL bRc = FALSE;
      __int64 nTS1 = -1;
      __int64 nTS2 = -1;
      __int64 nCounter = 0;

      *pnPacktesPerSecond = 0;

      if (pnPacktesPerSecond == NULL){return FALSE;}
      if ((m_hFile == NULL)||(m_hFile == INVALID_HANDLE_VALUE))
      {return FALSE;}

      while  ((dwReaded  >=  nBytesToReadFromFile)  &&  ((nTS1  <  0)  ||
      (nTS2 < 0)))
      {
            bRc      =      ReadFile(m_hFile,bData,nBytesToReadFromFile,
            &dwReaded,NULL);

            if (bRc == FALSE){return FALSE;}

            if (nTS1 < 0)
            {
                  nTS1 = GetPCR (bData,dwReaded);
                  if (nTS1 >= 0){nCounter++;}

                  if (pnFirstPCR != NULL){*pnFirstPCR = nTS1;}
            }
            else if (nTS2 < 0)
            {
                  nTS2 = GetPCR (bData,dwReaded);
                  if (nTS2 < 0){nCounter++;}
```

```
                        if (pnSecondPCR != NULL){*pnSecondPCR = nTS2;}
                }
        }

        if ((nTS1 < 0)||(nTS2 < 0)){return FALSE;}

        double dGAP = ((double)(FromPCRToTimeus (nTS2) - FromPCRToTimeus
        (nTS1)) / 1000000.0);
        *pnPacktesPerSecond = (__int64)((double)nCounter / dGAP);

        if (SetFilePointer (m_hFile,0,NULL,FILE_BEGIN) != 0)
        {return FALSE;}

        return TRUE;
}

__int64 CRTPSrc::GetPCR (BYTE *pData,int nDataLen)
{
        if (pData == NULL){return -1;}

        __int64 lTstampRet = -1;
        __int64 lTstamp = 0;
        __int64 lTstampTmp = 0;
        BOOL bFound = FALSE;
        int n = 0;

        while ((n < nDataLen)&&(bFound == FALSE))
        {
                if (pData[n] = 0x47)
                {
                        BYTE bTmp = pData[n + 3];
                        if (((bTmp>>4)&0x3) == 0x3) //Adaptation pessent
                        {
                                //Adaptation
                                if (pData[n + 4] > 1) ////Adaptation length >1
                                {
                                        BYTE bTmp = pData[n + 5];
                                        if (((bTmp>>4)&0x1)>0) //PCR FLAG
                                        {
                                                lTstamp    =    lTstamp|(pData[n    +
                                                6]&0xFF);
                                                lTstamp = lTstamp<<8;
                                                lTstamp    =    lTstamp|(pData[n    +
                                                7]&0xFF);
                                                lTstamp = lTstamp<<8;
                                                lTstamp    =    lTstamp|(pData[n    +
                                                8]&0xFF);
                                                lTstamp = lTstamp<<8;
                                                lTstamp    =    lTstamp|(pData[n    +
                                                9]&0xFF);
                                                lTstamp = lTstamp<<8;
                                                lTstamp    =    lTstamp|(pData[n    +
                                                10]&0xFF);
                                                lTstamp = lTstamp<<8;
                                                lTstamp    =    lTstamp|(pData[n    +
                                                11]&0xFF);

                                                int   nRem     =   (int)(lTstamp   &
                                                0x1FF);
                                                lTstamp = (lTstamp>>15);
```

```
                                    lTstampRet  =  (lTstamp  *  300)  +
                                    nRem;

                                    bFound = TRUE;
                              }
                        }
                  }
            }

            n = n + 188;
      }

      return lTstampRet;
}

__int64 CRTPSrc::FromPCRToPSTus (__int64 nPCR)
{
      unsigned        long        lTstampRTP        =        (unsigned
long)(nPCR>>16)&0xFFFFFFFFFFFFFFFF;
      double dTmp = (double)lTstampRTP;
      dTmp = dTmp * 11.11111111111;

      return (__int64)dTmp;
}
```

Code 4 *(Get MPEG2 transport stream timing data)*

3.1.5 Is EOF

Check if we reach the end of file. If is true we close all resources and exit. This is a trivial part of the code.

3.1.6 Read file chunk

We Read 7 MTS packets because we know that the Maximum Transfer Unit (MTU) in ethernet is 1500 bytes and:

- 188 bytes * 7 packets = 1316 bytes
- RTP header length in this application = 12 bytes
- RTP packet length 1316 + 12 = **1328 bytes (1328 < 1500)**

If we follow the RFC2250 rules, we have to put an integer number of MPEG transport stream packets into RTP payload, and RFC3550 says that we have to try to put all RTP packets as possible into underneath layer packet.

```
DWORD dwBytesToReadFromFile = (7 * 188);


bRc = ReadFile(m_hFile,&m_pData[m_nFirstPayloadByte],
dwBytesToReadFromFile,&dwReaded,NULL);

if (bRc == FALSE)
{
      printf ("Error Reading a File\n");
      return FALSE;
}
```

Code 5 *(Read file chunk)*

3.1.7 Get PCR data

We try to get the first timestamp of all read packets; we use that information to calculate the drift of our source versus internal file timestamps. See Code 6 and Code 4.

```
nTstampTmp = GetPCR (&m_pData[m_nFirstPayloadByte],(int)dwReaded);
```

Code 6 *(Get PCR data)*

3.1.8 Update RTP header timestamp

If we can read the timestamping data of the file, we use that timing information to update the RTP timestamp field. In Code 7 we translate the MPEG timestamp (PCR) into RTP timestamp format.

In the Code 8 we update the sequence field and the timestamp field of RTP header.

```
//GET presentation timestamp & translate it as RTP timestamp
lTstampRTP = (unsigned long)(nTstampTmp>>16)&0xFFFFFFFFFFFFFFFF;
```

Code 7 *(Update RTP header timestamp)*

```
BOOL    CRTPSrc::UpdateRtpHeader    (BYTE    *pData,int    nDataLen,int
nSeqNumber,long lTstamp)
{
      if (pData == NULL){return FALSE;}
      if (nDataLen < 8){return FALSE;}

      pData[2] = (BYTE) ((nSeqNumber&0xFF00)>>8);
      pData[3] = (BYTE) (nSeqNumber&0x00FF);

      pData[4] = (BYTE) ((lTstamp&0xFF000000)>>24);
      pData[5] = (BYTE) ((lTstamp&0x00FF0000)>>16);
      pData[6] = (BYTE) ((lTstamp&0x0000FF00)>>8);
      pData[7] = (BYTE) (lTstamp&0x000000FF);

      return TRUE;
}
```

Code 8 *(Update RTP header timestamp)*

3.1.9 Update RTP header sequence

To update the sequence field we use the same function as we used to update timestamp field. See *Code 8*.

3.1.10 Send RTP packet and bitrate control

This block sends the RTP header followed by 7 MTS packets. After that, it applies a rate control.

To perform rate control:

1- It gets the time gap from application starts to send packets until now.

2- It calculates the number of packets that the application should have sent based in the gap time and the number of RTP packets per second calculated in 3.1.4.

3- It compares the number of RTP packets that it has really sent against the theoretical number of them.

4- If the number of sent RTP packets is higher that the number that it should have been sent, then it calculates the delay that has to apply (the number of RTP packet difference * packet transmission time), and it calls sleep function.

```
if      (sendto(m_nSck,(const      char*)m_pData,dwReaded      +
m_nFirstPayloadByte,0,(struct      sockaddr*)&m_groupSock,sizeof
(sockaddr_in))== SOCKET_ERROR)
{
      printf ("Error sendto\n");
      return FALSE;
}

nSeq++;

//Calc time
long lGap = timeGetTime () - dwCounterStartms;
double       dNumPacketsToSend       =       (double)lGap       *
(double)m_nNumPacketsPerSecond;

long lSentPackets = nSeq * 7;

long lNumPacketsToSend =  dNumPacketsToSend / 1000.0;

if (lSentPackets > lNumPacketsToSend)
{
      long  lNumOverflowPackets  =  (nSeq  *  TS_PACKETS_TO_READ)  -
      lNumPacketsToSend;
      DWORD    dwWaitMs    =    (DWORD)(lNumOverflowPackets    *
      ((double)m_nNumPacketsPerSecond/1000.0));

      if (bVerbose == TRUE ){printf ("Delay: %d ms, Adv: %d ms,
      lNumPacketsToSend:            %d,            lSentPackets:
      %d\n",dwWaitMs,lAdvms,lNumPacketsToSend,lSentPackets);}

      Sleep(dwWaitMs);
}
```

***Code 9** (Send RTP packet and rate control)*

3.2 Application test environment

To demonstrate that our application works good we implement the test environment shown in Figure 11.

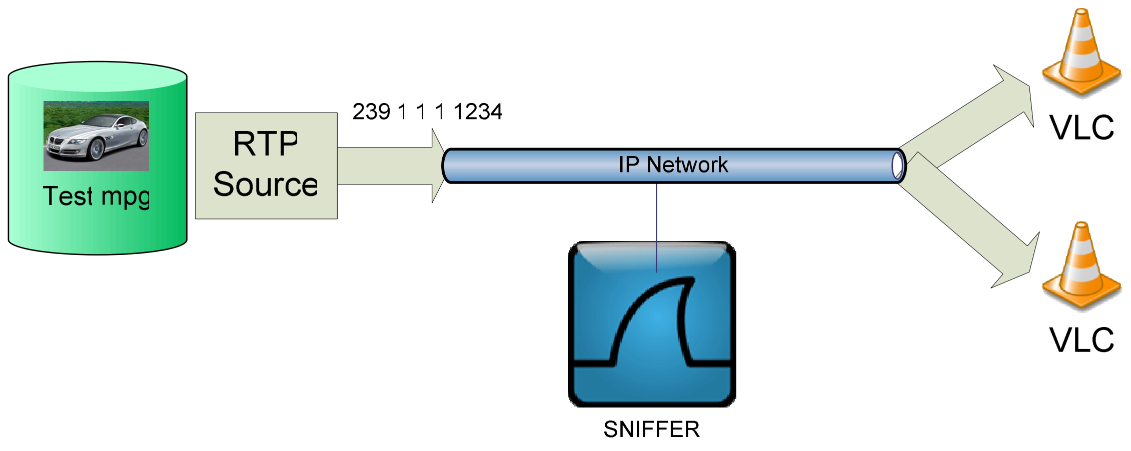
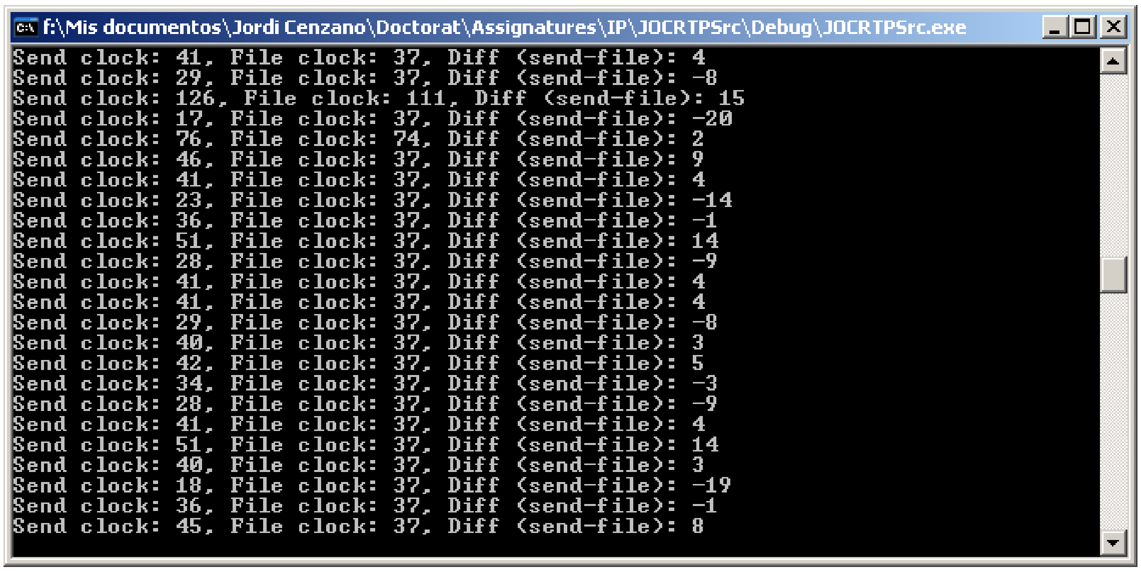


Figure 11 *(Test environment)*

We used our application to send a MTS file through the network, we have used wireshark (free sniffer) like a network monitor, and we have run two VLCs (free video player) to display the streamed video.

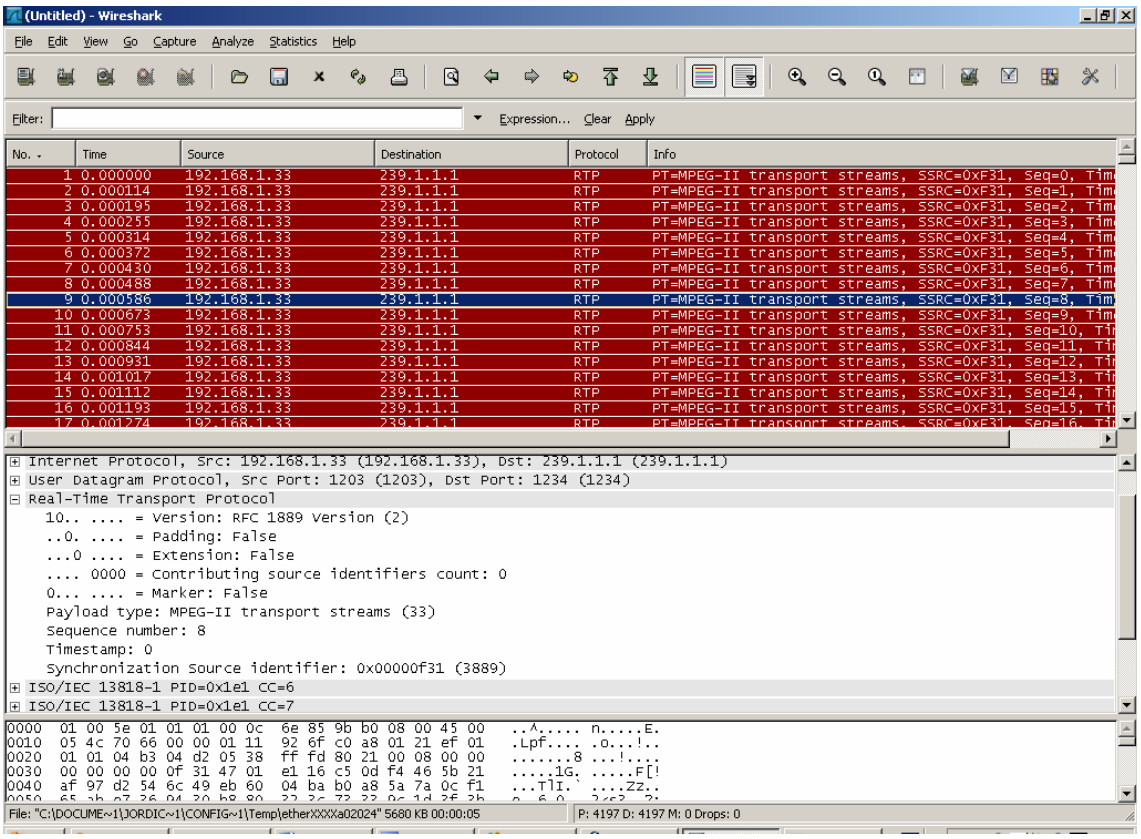
In the Figure 12 we can see a screenshot of our application (RTP Source). We see that the application clock oscillates around the file clock.


f:\Mis documentos\Jordi Cenzano\Doctorat\Assignatures\IP\JOCRTPSrc\Debug\JOCRTPSrc.exe

```
Send clock: 41, File clock: 37, Diff (send-file): 4
Send clock: 29, File clock: 37, Diff (send-file): -8
Send clock: 126, File clock: 111, Diff (send-file): 15
Send clock: 17, File clock: 37, Diff (send-file): -20
Send clock: 76, File clock: 74, Diff (send-file): 2
Send clock: 46, File clock: 37, Diff (send-file): 9
Send clock: 41, File clock: 37, Diff (send-file): 4
Send clock: 23, File clock: 37, Diff (send-file): -14
Send clock: 36, File clock: 37, Diff (send-file): -1
Send clock: 51, File clock: 37, Diff (send-file): 14
Send clock: 28, File clock: 37, Diff (send-file): -9
Send clock: 41, File clock: 37, Diff (send-file): 4
Send clock: 41, File clock: 37, Diff (send-file): 4
Send clock: 29, File clock: 37, Diff (send-file): -8
Send clock: 40, File clock: 37, Diff (send-file): 3
Send clock: 42, File clock: 37, Diff (send-file): 5
Send clock: 34, File clock: 37, Diff (send-file): -3
Send clock: 28, File clock: 37, Diff (send-file): -9
Send clock: 41, File clock: 37, Diff (send-file): 4
Send clock: 51, File clock: 37, Diff (send-file): 14
Send clock: 40, File clock: 37, Diff (send-file): 3
Send clock: 18, File clock: 37, Diff (send-file): -19
Send clock: 36, File clock: 37, Diff (send-file): -1
Send clock: 45, File clock: 37, Diff (send-file): 8
```

Figure 12 *(RTP source screenshot)*

We have used a sniffer with RTP packet decoding to check if all parameters in all RTP header are OK. In the next figure we can see that.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No. .  Time       Source         Destination  Protocol  Info
     1 0.000000   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=0, Tim
     2 0.000114   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=1, Tim
     3 0.000195   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=2, Tim
     4 0.000255   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=3, Tim
     5 0.000314   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=4, Tim
     6 0.000372   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=5, Tim
     7 0.000430   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=6, Tim
     8 0.000488   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=7, Tim
     9 0.000586   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=8, Tim
    10 0.000673   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=9, Tim
    11 0.000753   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=10, Ti
    12 0.000844   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=11, Ti
    13 0.000931   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=12, Ti
    14 0.001017   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=13, Ti
    15 0.001112   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=14, Ti
    16 0.001193   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=15, Ti
    17 0.001274   192.168.1.33   239.1.1.1    RTP       PT=MPEG-II transport streams, SSRC=0xF31, Seq=16, Ti
```

```
⊞ Internet Protocol, Src: 192.168.1.33 (192.168.1.33), Dst: 239.1.1.1 (239.1.1.1)
⊞ User Datagram Protocol, Src Port: 1203 (1203), Dst Port: 1234 (1234)
⊟ Real-Time Transport Protocol
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    Payload type: MPEG-II transport streams (33)
    Sequence number: 8
    Timestamp: 0
    Synchronization Source identifier: 0x00000f31 (3889)
⊞ ISO/IEC 13818-1 PID=0x1e1 CC=6
⊞ ISO/IEC 13818-1 PID=0x1e1 CC=7
```

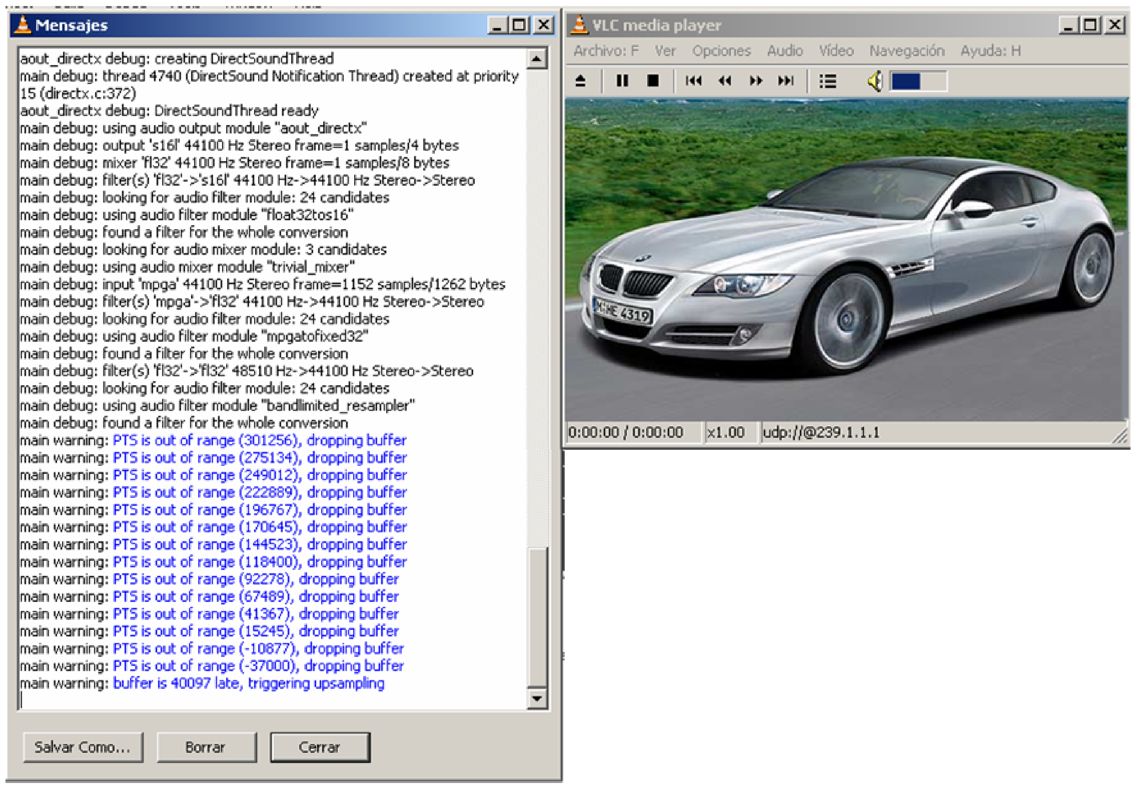
```
0000  01 00 5e 01 01 01 00 0c  6e 85 9b b0 08 00 45 00   ..^..... n.....E.
0010  05 4c 70 66 00 00 01 11  92 6f c0 a8 01 21 ef 01   .Lpf.... .o...!..
0020  01 01 04 b3 04 d2 05 38  ff fd 80 21 00 08 00 00   .......8 ...!....
0030  00 00 00 00 0f 31 47 01  e1 16 c5 0d f4 46 5b 21   .....1G. .....F[!
0040  af 97 d2 54 6c 49 eb 60  04 ba b0 a8 5a 7a 0c f1   ...TlI.` ....Zz..
```

```
File: "C:\DOCUME~1\JORDIC~1\CONFIG~1\Temp\etherXXXXa02024" 5680 KB 00:00:05    P: 4197 D: 4197 M: 0 Drops: 0
```

Figure 13 *(Sniffer screenshot)*

Finally we check to decode the stream in real time with a media player named VLC. VLC has integrated a RTP multicast decoder and a MPEG2 video decoder, we only have to put the multicast group and port and view the RTP streamed video.

We also check the warnings in VLC message window, and all RTP reception was OK. See Figure 14.



```
aout_directx debug: creating DirectSoundThread
main debug: thread 4740 (DirectSound Notification Thread) created at priority
15 (directx.c:372)
aout_directx debug: DirectSoundThread ready
main debug: using audio output module "aout_directx"
main debug: output 's16l' 44100 Hz Stereo frame=1 samples/4 bytes
main debug: mixer 'fl32' 44100 Hz Stereo frame=1 samples/8 bytes
main debug: filter(s) 'fl32'->'s16l' 44100 Hz->44100 Hz Stereo->Stereo
main debug: looking for audio filter module: 24 candidates
main debug: using audio filter module "float32tos16"
main debug: found a filter for the whole conversion
main debug: looking for audio mixer module: 3 candidates
main debug: using audio mixer module "trivial_mixer"
main debug: input 'mpga' 44100 Hz Stereo frame=1152 samples/1262 bytes
main debug: filter(s) 'mpga'->'fl32' 44100 Hz->44100 Hz Stereo->Stereo
main debug: looking for audio filter module: 24 candidates
main debug: using audio filter module "mpgatofixed32"
main debug: found a filter for the whole conversion
main debug: filter(s) 'fl32'->'fl32' 48510 Hz->44100 Hz Stereo->Stereo
main debug: looking for audio filter module: 24 candidates
main debug: using audio filter module "bandlimited_resampler"
main debug: found a filter for the whole conversion
main warning: PTS is out of range (301256), dropping buffer
main warning: PTS is out of range (275134), dropping buffer
main warning: PTS is out of range (249012), dropping buffer
main warning: PTS is out of range (222889), dropping buffer
main warning: PTS is out of range (196767), dropping buffer
main warning: PTS is out of range (170645), dropping buffer
main warning: PTS is out of range (144523), dropping buffer
main warning: PTS is out of range (118400), dropping buffer
main warning: PTS is out of range (92278), dropping buffer
main warning: PTS is out of range (67489), dropping buffer
main warning: PTS is out of range (41367), dropping buffer
main warning: PTS is out of range (15245), dropping buffer
main warning: PTS is out of range (-10877), dropping buffer
main warning: PTS is out of range (-37000), dropping buffer
main warning: buffer is 40097 late, triggering upsampling
```

Figure 14 *(VLC Decoder window)*

4 Conclusions

1- We realize that RTP is a protocol that can be large tailored and it is open to carry all kind of data: video, audio, simulation data, etc. The RFC3550 is an open standard that is refined by other standards like RFC2250 (the audio / video profile).

2- RTP is an end to end protocol at application level and it uses the resources of underneath protocol layers. RTP usually runs over UDP but is not mandatory by the standard; it can work over different network technologies and protocols.

3- RTP gives information about its payload to data decoders and adds to underling protocols two mechanisms:
   a. Sequencing: It allows ~~to~~ the receivers to detect packet loses, and restores packet sequence.
   b. Timestamping: It allows synchronization and jitter calculation to the receiver.

4- Thanks to its mechanism RTP is a very useful protocol to transport video and audio in real time, due to its low latency algorithm (no acknowledgment, no error detecting/correction).

5- Using RTP / RTCP you can detect network congestions or network problems and you can change parameters of your sources to adapt your traffic to the network dynamic conditions.

6- Implementing the RTP source application we can confirm that the packet timestamps and the receiver buffer are two delicate parameters. The timestamps have to be very exact to avoid dropping frames, and reception buffer is a trade-off between latency and network jitter. To have low latency we have to limit our network jitter.

5 References

1. Wikipedia (RTP protocol)
<http://en.wikipedia.org/wiki/Real-time_Transport_Protocol> [Visited: 8 January 2008]

2. Wikipedia (OSI Model)
<http://es.wikipedia.org/wiki/Modelo_OSI> [Visited: 8 January 2008]

3. RFC3350 Standard (RTP)
<http://tools.ietf.org/html/rfc3550> [Visited: 8 January 2008]

4. RFC3351 Standard (RTP profile for audio video conferences with minimal control)
<http://tools.ietf.org/html/rfc3551> [Visited: 8 January 2008]

5. RFC2250 Standard (RTP payload format for MPEG1/MPEG2 video)
<http://tools.ietf.org/html/rfc3551> [Visited: 8 January 2008]

6. Wireshark (Network sniffer with integrated RTP packet decoder)
<http://www.wireshark.org> [Visited: 8 January 2008]

7. VLC (Media player with streaming RTP decoder)
<http://www.videolan.org/vlc> [Visited: 8 January 2008]