TCP Audio Streamer and Player (Voice Chat over IP)

Banjoo
23 Apr 2014 CPOL

Stream TCP Audio data (Voice Chat over IP)

Download EXE - 96.4 KB

Download source - 182.7 KB

Introduction

This is a proprietary VoIP project to send and receive audio data by TCP. It's an extension of my first article **Play or Capture Audio Sound. Send and Receive as Multicast (RTP)**. This application streams the audio data not by multicast but by TCP. So you can be sure there is no data lost and you can transfer them over subnets and routers away. The audio codec is U-Law. The sample rate is selectable from 5000 to 44100.

The server can run on your local PC. You can get your current IP4-Address with help of running *cmd.exe* and typing "`ipconfig`". You should use a static IP-Address, so that possible clients do not have to change their settings after reconnecting some days later. The clients must connect to the IP4-Address and port configured on the running server. The server can be run in silent mode (no input, no output) just transferring audio data between all connected clients.

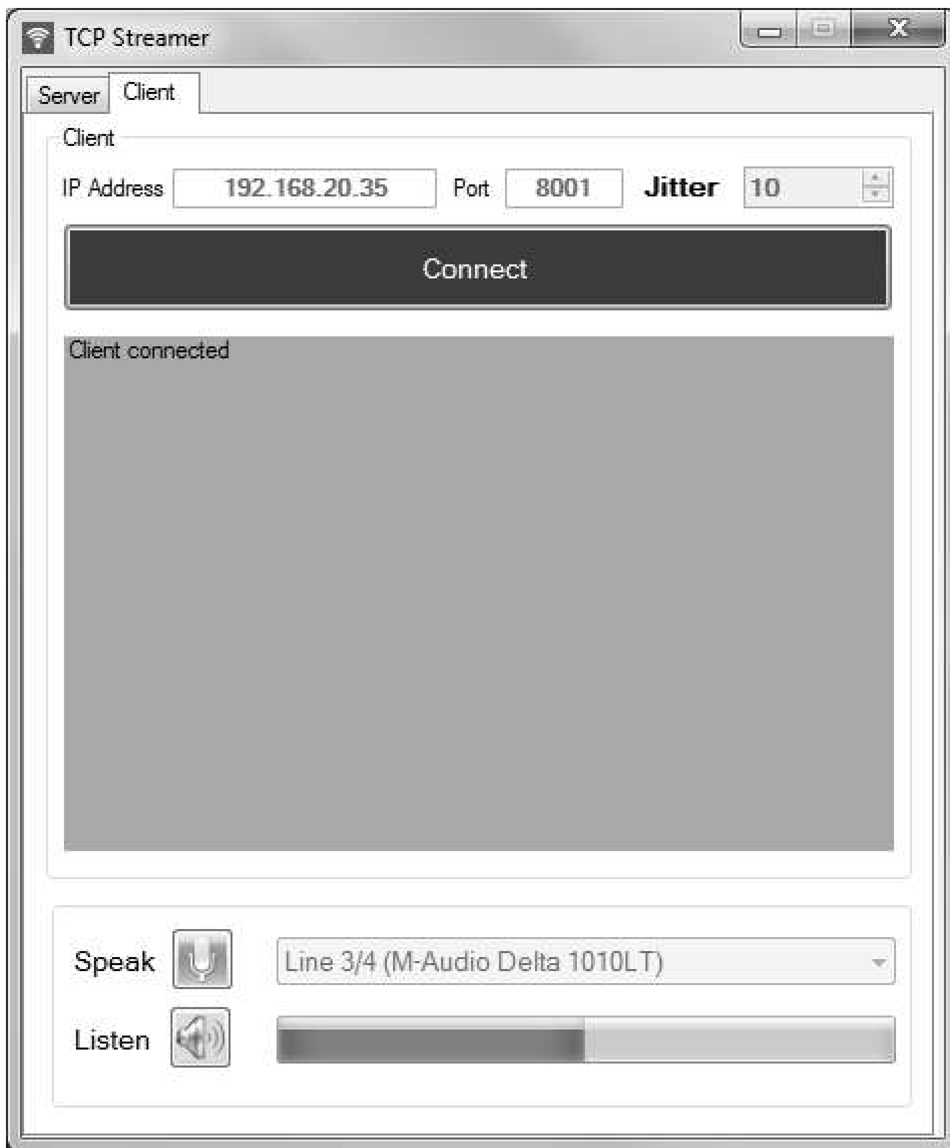
Choose a free port that is not used by another application (do not use 80 or other reserved ports). You can connect within LAN or Internet. For Internet chatting, you can configure a port forwarding on your router.

Note !!! This is a proprietary project. You can't use my servers or clients with any other standardized servers or clients. I don't use standards like RTCP or SDP.

Background

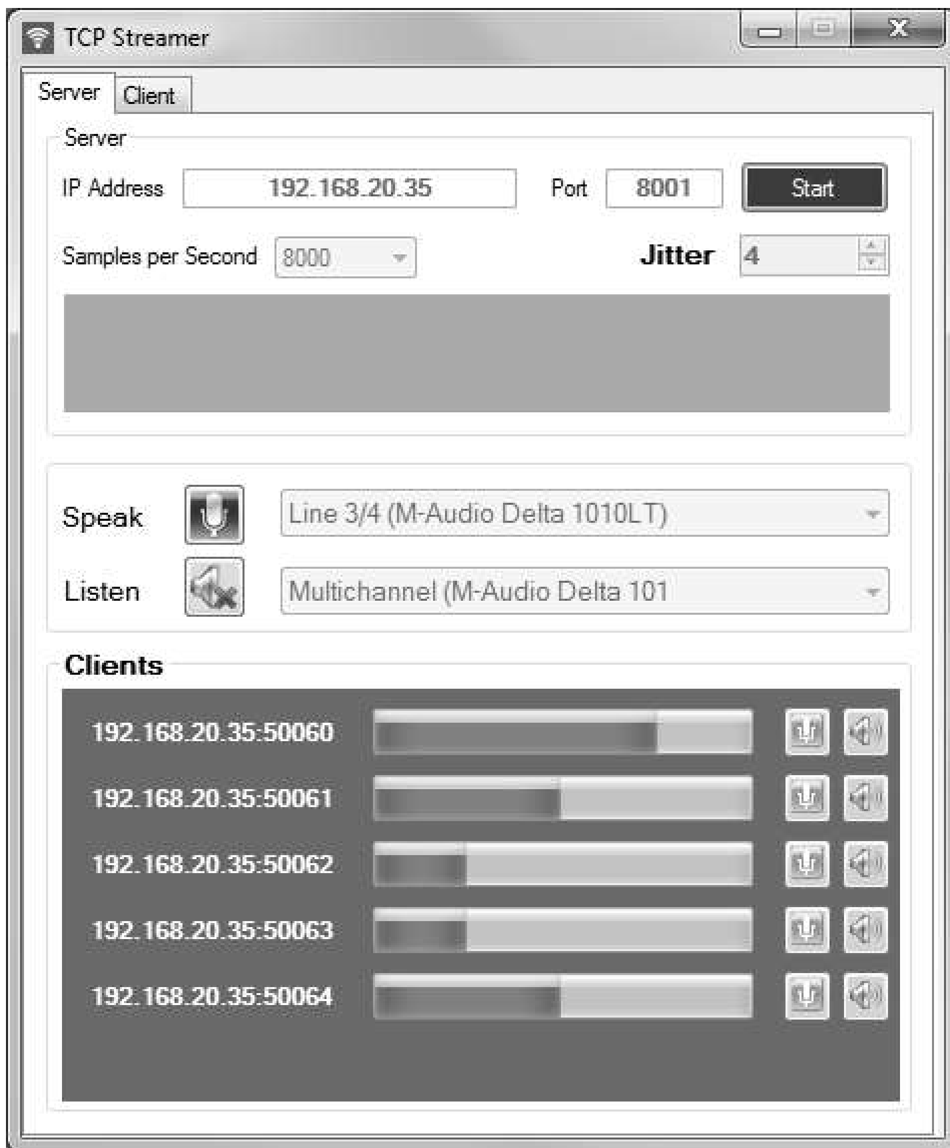
Because of network traffic and time clock differences, you have to use Jitter-Buffers to compensate data transfer. You can set the Jitter-Buffer for each server, so all clients will use the same amount. One Jitter-Buffer represents one data-packet, included in a TCP-Stream. The server starts playing, when the Jitter-Buffer reaches the half of maximum. You can watch this in the `progressbar` which is shown for each client. The more Jitter-Buffers you set, the more delay will occur. You can run the `TCPStreamer` as client or as server. One server can handle one or more clients.

The TCPStreamer as Client



Running as client, you can connect to a server instance. Choose your microphone and listen device. Click on the microphone or speaker buttons to mute. After the client is connected, the speaker combobox changes to a progress bar showing the value of incoming data. The used `SamplesPerSecond` (Quality) depends on Server-Configuration. The Jitter-Buffer client sides is only important for the delay of incoming data.

The TCPStreamer as Server



Running as a server, you can wait for one or more clients. Choose your microphone and listen device if wanted, but you can run the server without hearing or speaking server sides, so that only the clients speak to each other. Each client can be muted exclusive (Speaker and Micro). The IP-Address must be the address of your computer. The port number should not be used by other applications. The Jitter-Buffer value server sides is important for the delay of all connected clients. Use the lowest value as possible. The server has to mix all data from all clients, so you should choose a performant workstation on which to run the server. The quality of the speak depends on the `SamplesPerSecond` value.

Using the Code

There are the following assemblies:

- *TCPStreamer.exe* (main application)
- *TCPClient.dll* (TCP client wrapper helper)
- *TCPServer.dll* (TCP server wrapper helper)
- *WinSound.dll* (sound recording and playing)

I could send data direct from soundcard to network. But I decided to put them into a Jitter Buffer first, because some sound devices (especially on laptops) are not able to get the sound data in equal time periods. With a Jitter Buffer, I ensure sending data every 20 ms. But the disadvantage is a bigger delay. The amount for that not configurable Jitter Buffer is **8**, for a lower

delay, you can reduce that value in the source code (`RecordingJitterBufferCount`). But watch your soundcard for some minutes, if it can handle this.

C#

```
//----------------------------------------------------------------------------------
-
//
//Recording datas from Soundcard and put into Jitter Buffer
//
//----------------------------------------------------------------------------------
-
private void OnDataReceivedFromSoundcard_Server(Byte[] data)
{
    //Split datas in smaller equal pieces
    int bytesPerInterval = WinSound.Utils.GetBytesPerInterval((uint)
      m_Config.SamplesPerSecondServer,
            m_Config.BitsPerSampleServer, m_Config.ChannelsServer);
    int count = data.Length / bytesPerInterval;
    int currentPos = 0;
    for (int i = 0; i < count; i++)
    {
        //Cast to RTP packet (Linear to U-Law)
        Byte[] partBytes = new Byte[bytesPerInterval];
        Array.Copy(data, currentPos, partBytes, 0, bytesPerInterval);
        currentPos += bytesPerInterval;
        WinSound.RTPPacket rtp = ToRTPPacket(partBytes,
           m_Config.BitsPerSampleServer, m_Config.ChannelsServer);

        //Put RTP packet into Jitter Buffer
        m_JitterBufferServerRecording.AddData(rtp);
    }
}
```

When creating a RTP packet, most information like CSRC Count or Version are the same. After every sent RTP packet, I have only to increase the `SequenceNumber` and `Timestamp`. Before this, I translate the linear data to a compressed U-Law format to avoid network traffic.

C#

```
//----------------------------------------------------------------------------------
-
//
//Creating a RTP packet from linear data
//
//----------------------------------------------------------------------------------
-
private WinSound.RTPPacket ToRTPPacket(Byte[] linearData, int bitsPerSample, int channels)
{
    //Convert linear to Mulaw
    Byte[] mulaws = WinSound.Utils.LinearToMulaw(linearData, bitsPerSample, channels);

    //Create new RTP Packet
    WinSound.RTPPacket rtp = new WinSound.RTPPacket();

    //Init base values
    rtp.Data = mulaws;
```

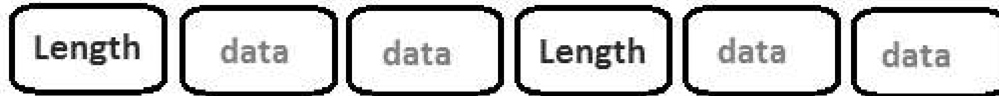
```
    rtp.CSRCCount = m_CSRCCount;
    rtp.Extension = m_Extension;
    rtp.HeaderLength = WinSound.RTPPacket.MinHeaderLength;
    rtp.Marker = m_Marker;
    rtp.Padding = m_Padding;
    rtp.PayloadType = m_PayloadType;
    rtp.Version = m_Version;
    rtp.SourceId = m_SourceId;

    //Update RTP header with SequenceNumber and Timestamp
    try
    {
        rtp.SequenceNumber = Convert.ToUInt16(m_SequenceNumber);
        m_SequenceNumber++;
    }
    catch (Exception)
    {
        m_SequenceNumber = 0;
    }
    try
    {
        rtp.Timestamp = Convert.ToUInt32(m_TimeStamp);
        m_TimeStamp += mulaws.Length;
    }
    catch (Exception)
    {
        m_TimeStamp = 0;
    }

    //Ready
    return rtp;
}
//-----------------------------------------------------------------------------------------
-
//
//Send sound datas (U-Law) over network
//
//-----------------------------------------------------------------------------------------
-
private void OnJitterBufferServerDataAvailable(Object sender, WinSound.RTPPacket rtp)
{
  //Convert RTP packt to bytes
  Byte[] rtpBytes = rtp.ToBytes();

  //For all clients connected
  List<NF.ServerThread> list = new List<NF.ServerThread>(m_Server.Clients);
  foreach (NF.ServerThread client in list)
  {
      //If not mute
      if (client.IsMute == false)
      {
        //Send
        client.Send(m_PrototolClient.ToBytes(rtpBytes));
      }
  }
}
```

In order to send and receive data by TCP, I use a simple proprietary protocol. Before each data block, I write a 32 bit data length information. So later, when I receive the data stream, I know how to interpret the data.



C#

```
//---------------------------------------------------------------------------------------
-
//
//Convert bytes to a proprietary protocol format
//
//---------------------------------------------------------------------------------------
-
public Byte[] ToBytes(Byte[] data)
{
   //Get length of the data block
   Byte[] bytesLength = BitConverter.GetBytes(data.Length);

   //Copy all together
   Byte[] allBytes = new Byte[bytesLength.Length + data.Length];
   Array.Copy(bytesLength, allBytes, bytesLength.Length);
   Array.Copy(data, 0, allBytes, bytesLength.Length, data.Length);

   //ready
   return allBytes;
}
```

The reverse path is to get the data by network in this case for every connected client. In the first step, I have to extract the packets from the whole stream with the help of my own protocol.

C#

```
//---------------------------------------------------------------------------------------
-
//
//Get datas over network
//
//---------------------------------------------------------------------------------------
-
private void OnServerDataReceived(NF.ServerThread st, Byte[] data)
{
    //If client existing
    if (m_DictionaryServerDatas.ContainsKey(st))
    {
        //Get protocol
        ServerThreadData stData = m_DictionaryServerDatas[st];
        if (stData.Protocol != null)
        {
           //Dispatch data over protocol
           stData.Protocol.Receive_LH(st, data);
        }
    }
}
```

With the help of the length information, I know when a packet starts and ends.

C#

```
//-----------------------------------------------------------------------------------------
-
//
//Get RTP datas with help of a proprietary protocol
//
//-----------------------------------------------------------------------------------------
-
public void Receive_LH(Object sender, Byte[] data)
{
   //Add datas to buffer
   m_DataBuffer.AddRange(data);

   //Check buffer overflow
   if (m_DataBuffer.Count > m_MaxBufferLength)
   {
       m_DataBuffer.Clear();
   }

   //Get the length of received datas (16 Bit value)
   Byte[] bytes = m_DataBuffer.Take(4).ToArray();
   int length = (int)BitConverter.ToInt32(bytes.ToArray(), 0);

   //Check maximum length
   if (length > m_MaxBufferLength)
   {
       m_DataBuffer.Clear();
   }

   //For each complete data packet (check by the length)
   while (m_DataBuffer.Count >= length + 4)
   {
       //Get data
       Byte[] message = m_DataBuffer.Skip(4).Take(length).ToArray();

       //Raise event
       if (DataComplete != null)
       {
           DataComplete(sender, message);
       }
       //Remove handled datas from buffer
       m_DataBuffer.RemoveRange(0, length + 4);

       //As long as complete datas are available
       if (m_DataBuffer.Count > 4)
       {
           //Get next length
           bytes = m_DataBuffer.Take(4).ToArray();
           length = (int)BitConverter.ToInt32(bytes.ToArray(), 0);
       }
   }
}
```

Before playing the data to soundcard, I put them into a further Jitter Buffer. This is necessary because of the irregular network traffic, especially over internet. The more the amount of Jitter Buffer, the more the delay.

C#

```
//----------------------------------------------------------------------------------------
-
//
//Put network datas into Jitter Buffer
//
//----------------------------------------------------------------------------------------
-
private void OnProtocolDataComplete(Object sender, Byte[] bytes)
{
   //Convert bytes to RTP packet
   WinSound.RTPPacket rtp = new WinSound.RTPPacket(bytes);

   //When RTP packet correct
   if (rtp.Data != null)
   {
     //Add RTP packet to Jitter Buffer
     JitterBuffer.AddData(rtp);
   }
}
```

Finally, the data are ready to be played to soundcard. Before that, I translate the U-Law data back to linear data, because a sounddevice can only play linear one.

C#

```
//----------------------------------------------------------------------------------------
-
//
//Play datas to soundcard
//
//----------------------------------------------------------------------------------------
-
private void OnJitterBufferDataAvailable(Object sender, WinSound.RTPPacket rtp)
{
   //If not muted
   if (IsMuteAll == false && IsMute == false)
   {
       //Convert U-Law to linear
       Byte[] linearBytes = WinSound.Utils.MuLawToLinear(rtp.Data, BitsPerSample,
Channels);
       //Play to soundcard
       Player.PlayData(linearBytes, false);
   }
}
```

I implemented my own Jitter Buffer as a queue of RTP packets. The data can be added and then is handled by a high frequently timer function (20 ms).

C#

```
//----------------------------------------------------------------------------------------
-
```

```
//
// Adding datas to Jitter Buffer
//
//-----------------------------------------------------------------------------------------
-
public void AddData(RTPPacket packet)
{
    //Check overflow
    if (m_Overflow == false)
    {
        //Check maximum size
        if (m_Buffer.Count <= m_MaxRTPPackets)
        {
        //Adding data
            m_Buffer.Enqueue(packet);
        }
        else
        {
            //Overflow
            m_Overflow = true;
        }
    }
}
```

The Jitter Buffer handles the data every 20 milliseconds. To get such an exact timer, you can't use the normal .NET Timers. So I used the timer functions from the Win32 kernel32 and Winmm library. Before starting a timer, I set the precision as best as the system can offer. This can differ from 1 to more milliseconds. Better than 1 millisecond is not possible with Windows.

C#

```
[DllImport("Kernel32.dll", EntryPoint = "QueryPerformanceCounter")]
public static extern bool QueryPerformanceCounter(out long lpPerformanceCount);

[DllImport("Kernel32.dll", EntryPoint = "QueryPerformanceFrequency")]
public static extern bool QueryPerformanceFrequency(out long lpFrequency);

[DllImport("winmm.dll", SetLastError = true, EntryPoint = "timeSetEvent")]
public static extern UInt32 TimeSetEvent(UInt32 msDelay, UInt32 msResolution,
       TimerEventHandler handler, ref UInt32 userCtx, UInt32 eventType);

[DllImport("winmm.dll", SetLastError = true, EntryPoint = "timeKillEvent")]
public static extern UInt32 TimeKillEvent(UInt32 timerId);

[DllImport("kernel32.dll", EntryPoint = "CreateTimerQueue")]
public static extern IntPtr CreateTimerQueue();

[DllImport("kernel32.dll", EntryPoint = "DeleteTimerQueue")]
public static extern bool DeleteTimerQueue(IntPtr TimerQueue);

[DllImport("kernel32.dll", EntryPoint = "CreateTimerQueueTimer")]
public static extern bool CreateTimerQueueTimer(out IntPtr phNewTimer, IntPtr TimerQueue,
  DelegateTimerProc Callback, IntPtr Parameter, uint DueTime, uint Period, uint Flags);

[DllImport("kernel32.dll")]
public static extern bool DeleteTimerQueueTimer(IntPtr TimerQueue,
              IntPtr Timer, IntPtr CompletionEvent);
```

```
[DllImport("winmm.dll", SetLastError = true, EntryPoint = "timeGetDevCaps")]
public static extern MMRESULT TimeGetDevCaps(ref TimeCaps timeCaps, UInt32 sizeTimeCaps);

[DllImport("winmm.dll", SetLastError = true, EntryPoint = "timeBeginPeriod")]
public static extern MMRESULT TimeBeginPeriod(UInt32 uPeriod);

[DllImport("winmm.dll", SetLastError = true, EntryPoint = "timeEndPeriod")]
public static extern MMRESULT TimeEndPeriod(UInt32 uPeriod);
```

The Jitter Buffer is designed to handle the data, when half of the maximum is reached. After an overflow or underflow, the buffer tries to get back to this value.

C#

```
//-------------------------------------------------------------------------------------------
-
//
// Jitter Buffer Timer main function
//
//-------------------------------------------------------------------------------------------
-
private void OnTimerTick()
{
  if (DataAvailable != null)
  {
      //When datas existing
      if (m_Buffer.Count > 0)
      {
          //When overflow
          if (m_Overflow)
          {
              //Wait until buffer is half of maximum
              if (m_Buffer.Count <= m_MaxRTPPackets / 2)
              {
                  m_Overflow = false;
              }
          }

          //When underflow
          if (m_Underflow)
          {
              //Wait until buffer is half of maximum
              if (m_Buffer.Count < m_MaxRTPPackets / 2)
              {
                  return;
              }
              else
              {
                  m_Underflow = false;
              }
          }

          //Get data and raise event
          m_LastRTPPacket = m_Buffer.Dequeue();
          DataAvailable(m_Sender, m_LastRTPPacket);
      }
      else
```

```
        {
            //No overflow
            m_Overflow = false;

            //Whenn buffer is empty
            if (m_LastRTPPacket != null && m_Underflow == false)
            {
                if (m_LastRTPPacket.Data != null)
                {
                    //Underflow
                    m_Underflow = true;
                }
            }
        }
    }
}
```

This project does not use overheaded libraries or extensions, so it can be used to learn the basics of manipulating sound data and network operations. Feel free to extend and improve it for your needs.

History

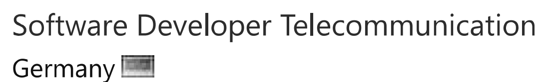
- 31.05.2012 - Added
- 03.05.2013 - Added duplex connections. Removed File-Player
- 09.05.2013 - Changed tip to article
- 12.12.2013 - Added communication between all clients
- 18.12.2013 - Fixed some bugs
- 22.04.2014 - Solved possible stability problems

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

About the Author

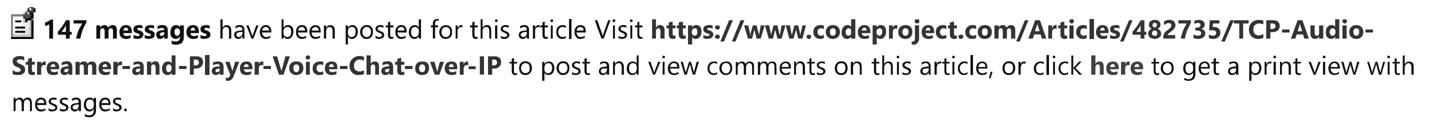
Banjoo



Software Developer Telecommunication
Germany

No Biography provided

Comments and Discussions


147 messages have been posted for this article Visit **https://www.codeproject.com/Articles/482735/TCP-Audio-Streamer-and-Player-Voice-Chat-over-IP** to post and view comments on this article, or click **here** to get a print view with messages.

Article Copyright 2012 by Banjoo
Everything else Copyright © CodeProject, 1999-2022