Bach Khoa University

Computer Network Assignment 1

Video Stream Programming Report

Group Member:
Alexandre Rousseau

Video Stream Programming Report

The Program includes 5 files:

- Client.py
- ClientLauncher.py
- RtpPacket.py
- Server.py
- Server.Worker.py

A. Run Server.py on Server Terminal to start server:

python Server.py server_port

server_port is the port your server listens to for incoming RTSP connections

We can give it the value 1025

Standard RTSP port is 554

In this project we shall make the value > 1024

B. Run ClientLauncher.py on Client Terminal to start a client:

python ClientLauncher.py server_host server_port PRT_port video_file

server_host is the IP address of local machine (we can use “127.0.0.1”)

server_port is the port the server is listening on (here “1025”)

RTP_port is the port where RTP packets are received (here “5008”)

video_file is the name of video file that we want to play (here “movie.mjpeg”)

RTSP (Real Time Streaming Protocol):

For entertainment and communications systems to control streaming media servers. Establishing and controlling media sessions between end points. It uses TCP.

RTP (Real-time Transport Protocol):

Network protocol for delivering audio and video over IP Networks. It uses UDP.

How RTSP and RTP work together?

```
class ServerWorker:
    SETUP = 'SETUP'
    PLAY = 'PLAY'
    PAUSE = 'PAUSE'
    TEARDOWN = 'TEARDOWN'

    INIT = 0
    READY = 1
    PLAYING = 2
    state = INIT

    OK_200 = 0
    FILE_NOT_FOUND_404 = 1
    CON_ERR_500 = 2
```

```
class Client:
    INIT = 0
    READY = 1
    PLAYING = 2
    state = INIT

    SETUP = 0
    PLAY = 1
    PAUSE = 2
    TEARDOWN = 3
```

What will be sent from client to server via RTSP Protocol are the commands like:

- SETUP
- PLAY
- PAUSE
- TEARDOWN

These commands will let server side know what next action; it should complete.

What will be replied from server to client via RTSP Protocol are the parameters like:

OK_200
FILE_NOT_FOUND_404
CON_ERR_500

To tell the client if the server receives its commands correctly.

After client receives server`s reply, it will change its state accordingly to:

- READY
- PLAYING

If SETUP command was sent from client to server:

```
def sendRtspRequest(self, requestCode):
    """Send RTSP request to the server."""
    #-------------
    # TO COMPLETE
    #-------------

    # Setup request
    if requestCode == self.SETUP and self.state == self.INIT:
        threading.Thread(target=self.recvRtspReply).start()
        # Update RTSP sequence number.
        # ...
        self.rtspSeq = 1

        # Write the RTSP request to be sent.
        # request = ...
        request = "SETUP " + str(self.fileName) + "\n" + str(self.rtspSeq) + "\n" + " RTSP/1.0 RTP/UDP " + str(self.rtpPort)

        self.rtspSocket.send(request.encode('utf-8'))
        # Keep track of the sent request.
        # self.requestSent = ...
        self.requestSent = self.SETUP
```

The “SETUP” RTSP Packet will include:

1. Setup command.
2. Video file name to be play.
3. RTSP Packet Sequence Number starts from 1.
4. Protocol type: RTSP/1.0 RTP.
5. Transmission Protocol: UDP.
6. RTP Port for video stream transmission.

```
    # Process SETUP request
    if requestType == self.SETUP:
        if self.state == self.INIT:
            # Update state
            print ("SETUP Request received\n")

            try:

                self.clientInfo['videoStream'] = VideoStream(filename)
                self.state = self.READY

            except IOError:
                self.replyRtsp(self.FILE_NOT_FOUND_404, seq[1])

            # Generate a randomized RTSP session ID
            self.clientInfo['session'] = randint(100000, 999999)

            # Send RTSP reply
            self.replyRtsp(self.OK_200, seq[0])  #seq[0] the sequenceNum received from Client.py
            print ("sequenceNum is " + seq[0])
            # Get the RTP/UDP port from the last line
            self.clientInfo['rtpPort'] = request[2].split(' ')[3]
            print ('-'*60 + "\nrtpPort is :" + self.clientInfo['rtpPort'] + "\n" + '-'*60)
            print ("filename is " + filename)
```

When Server side receives “SETUP” command, it will:

1. Assign the clint a Specific Session Number randomly.
2. If something wrong with this command or server’s state, it will reply ERROR packet back to client.
3. If command correct.

```
class VideoStream:
    def __init__(self, filename):
        self.filename = filename
        try:
            self.file = open(filename, 'rb')
            print('-'*60 +  "\nVideo file : |" + filename +  "| read\n" + '-'*60)
        except:
            print ("read " + filename + " error")
            raise IOError
        self.frameNum = 0
```

The server will open the video file specified in the SETUP Packet and Initialize its video frame number to 0.

C. If command processes correctly, it will reply OK_200 back to client and set its STATE to READY

The Client side will loop to receive Server`s RTSP Reply:

```
def recvRtspReply(self):
    """Receive RTSP reply from the server."""
    while True:
        reply = self.rtspSocket.recv(1024)

        if reply:
            self.parseRtspReply(reply)

        # Close the RTSP socket upon requesting Teardown
        if self.requestSent == self.TEARDOWN:
            self.rtspSocket.shutdown(socket.SHUT_RDWR)
            self.rtspSocket.close()
            break
```

Then Parse the RTSP Reply Packet:

```
def parseRtspReply(self, dt):
    print ("Parsing Received Rtsp data...")

    """Parse the RTSP reply from the server."""
    data = dt.decode('utf-8')
    lines = data.split('\n')
    seqNum = int(lines[1].split(' ')[1])

    # Process only if the server reply's sequence number is the same as the request's
    if seqNum == self.rtspSeq:
        session = int(lines[2].split(' ')[1])
        # New RTSP session ID
        if self.sessionId == 0:
            self.sessionId = session
```

And if the Reply Packet is responded for the SETUP command.

The client will set its STATE as READY:

```
if int(lines[0].split(' ')[1]) == 200:
    if self.requestSent == self.SETUP:
        #-------------
        # TO COMPLETE
        #-------------
        # Update RTSP state.
        print ("Updating RTSP state...")
        # self.state = ...
        self.state = self.READY
        # Open RTP port.
        #self.openRtpPort()
        print ("Setting Up RtpPort for Video Stream")
        self.openRtpPort()
```

Then open a RTP Port to receive incoming video stream:

```
try:
    #self.rtpSocket.connect(self.serverAddr,self.rtpPort)
    self.rtpSocket.bind((self.serverAddr,self.rtpPort))
    #self.rtpSocket.listen(5)
    print ("Bind RtpPort Success")

except:
    tkMessageBox.showwarning('Connection Failed', 'Connection to rtpServer failed...')
```

Afterward, if PLAY RTSP command was sent from client to server:

```
elif requestCode == self.PLAY and self.state == self.READY:
    # Update RTSP sequence number.
    # ...
    self.rtspSeq = self.rtspSeq + 1
    # Write the RTSP request to be sent.
    # request = ...
    request = "PLAY " + "\n" + str(self.rtspSeq)

    self.rtspSocket.send(request.encode('utf-8'))
    print ('-'*60 + "\nPLAY request sent to Server...\n" + '-'*60)
    # Keep track of the sent request.
    # self.requestSent = ...
    self.requestSent = self.PLAY
```

The Server will create a Socket for RTP transmission via UDP, and start a tread to send video stream packet:

```
elif requestType == self.PLAY:
    if self.state == self.READY:
        print( '-'*60 + "\nPLAY Request Received\n" + '-'*60)
        self.state = self.PLAYING

        # Create a new socket for RTP/UDP
        self.clientInfo["rtpSocket"] = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

        self.replyRtsp(self.OK_200, seq[0])
        print ('-'*60 + "\nSequence Number ("+ seq[0] + ")\nReplied to client\n" + '-'*60)

        # Create a new thread and start sending RTP packets
        self.clientInfo['event'] = threading.Event()
        self.clientInfo['worker']= threading.Thread(target=self.sendRtp)
        self.clientInfo['worker'].start()
```

VideoStream.py will help chop the video file to separate frame, and put each frame into RTP data packet:

```
def nextFrame(self):
    """Get next frame."""

    data = self.file.read(5) # Get the framelength from the first 5 bytes
    #data_ints = struct.unpack('<' + 'B'*len(data),data)
    data = bytearray(data)

    data_int = (data[0] - 48) * 10000 + (data[1] - 48) * 1000 + (data[2] - 48) * 100 + (data[3] - 48) * 10 + (data[4] - 48)# = #int(data.encode('hex'),16)

    final_data_int = data_int

    if data:

        framelength = final_data_int#int(data)#final_data_int/8  # xx bytes
        # Read the current frame
        frame = self.file.read(framelength)
        if len(frame) != framelength:
            raise ValueError('incomplete frame data')
        #print "frame length"
        #print len(frame)
        #if not (data.startswith(b'\xff\xd8') and data.endswith(b'\xff\xd9')):
        #    raise ValueError('invalid jpeg')

        self.frameNum += 1
        print ('-'*10 + "\nNext Frame (#" + str(self.frameNum) + ") length:" + str(framelength) + "\n" + '-'*10)

        return frame
```

Each data packet will also be encoded with a header, the header will include:

- RTP-version filed
- Padding
- Extension
- Contributing source
- Marker
- Type Field
- Sequence Number
- Timestamp
- SSRC

RTP packet header

Bit offset[b]	0–1	2	3	4–7	8	9–15	16–31
0	Version	P	X	CC	M	PT	Sequence number
32	Timestamp						
64	SSRC identifier						
96	CSRC identifiers ...						
96+32×CC	Profile-specific extension header ID						Extension header length
128+32×CC	Extension header ...						

They have been inserted in the RTP Packet via bitwise operations:

```
self.header[0] = version << 6
self.header[0] = self.header[0] | padding << 5
self.header[0] = self.header[0] | extension << 4
self.header[0] = self.header[0] | cc
self.header[1] = marker << 7
self.header[1] = self.header[1] | pt

self.header[2] = seqnum >> 8
self.header[3] = seqnum

self.header[4] = (timestamp >> 24) & 0xFF
self.header[5] = (timestamp >> 16) & 0xFF
self.header[6] = (timestamp >> 8) & 0xFF
self.header[7] = timestamp & 0xFF

self.header[8] = ssrc >> 24
self.header[9] = ssrc >> 16
self.header[10] = ssrc >> 8
self.header[11] = ssrc
```

Finally, the RTP Packet will include a header and a video frame be sent to the RTP Port on the client side:

```
self.clientInfo['rtpSocket'].sendto(self.makeRtp(data, frameNumber),(self.clientInfo['rtspSocket'][1][0],port))
```

Then Client decode the RTP Packet to get the header and the video frame, reorganize the frames and display on the UI:

```
def decode(self, byteStream):
    """Decode the RTP packet."""

    #print byteStream[:HEADER_SIZE]
    self.header = bytearray(byteStream[:HEADER_SIZE])   #temporary solved

    self.payload = byteStream[HEADER_SIZE:]
```

If a PAUSE command was sent from client to server, it will stop the server from sending video frames to client.

```
# Process PAUSE request
elif requestType == self.PAUSE:
    if self.state == self.PLAYING:
        print ('-'*60 + "\nPAUSE Request Received\n" + '-'*60)
        self.state = self.READY

        self.clientInfo['event'].set()

        self.replyRtsp(self.OK_200, seq[0])
```

If a TEARDOWN command was sent from client to server, it will also stop the server from sending video frames to client and close the client terminal as well.

```
# Process TEARDOWN request
elif requestType == self.TEARDOWN:
    print ('-'*60 + "\nTEARDOWN Request Received\n" + '-'*60)

    self.clientInfo['event'].set()

    self.replyRtsp(self.OK_200, seq[0])

    # Close the RTP socket
    self.clientInfo['rtpSocket'].close()
```