Ho Chi Minh City University Of Techonology

Computer NetWork

Assignment 1 - Video Streaming Application

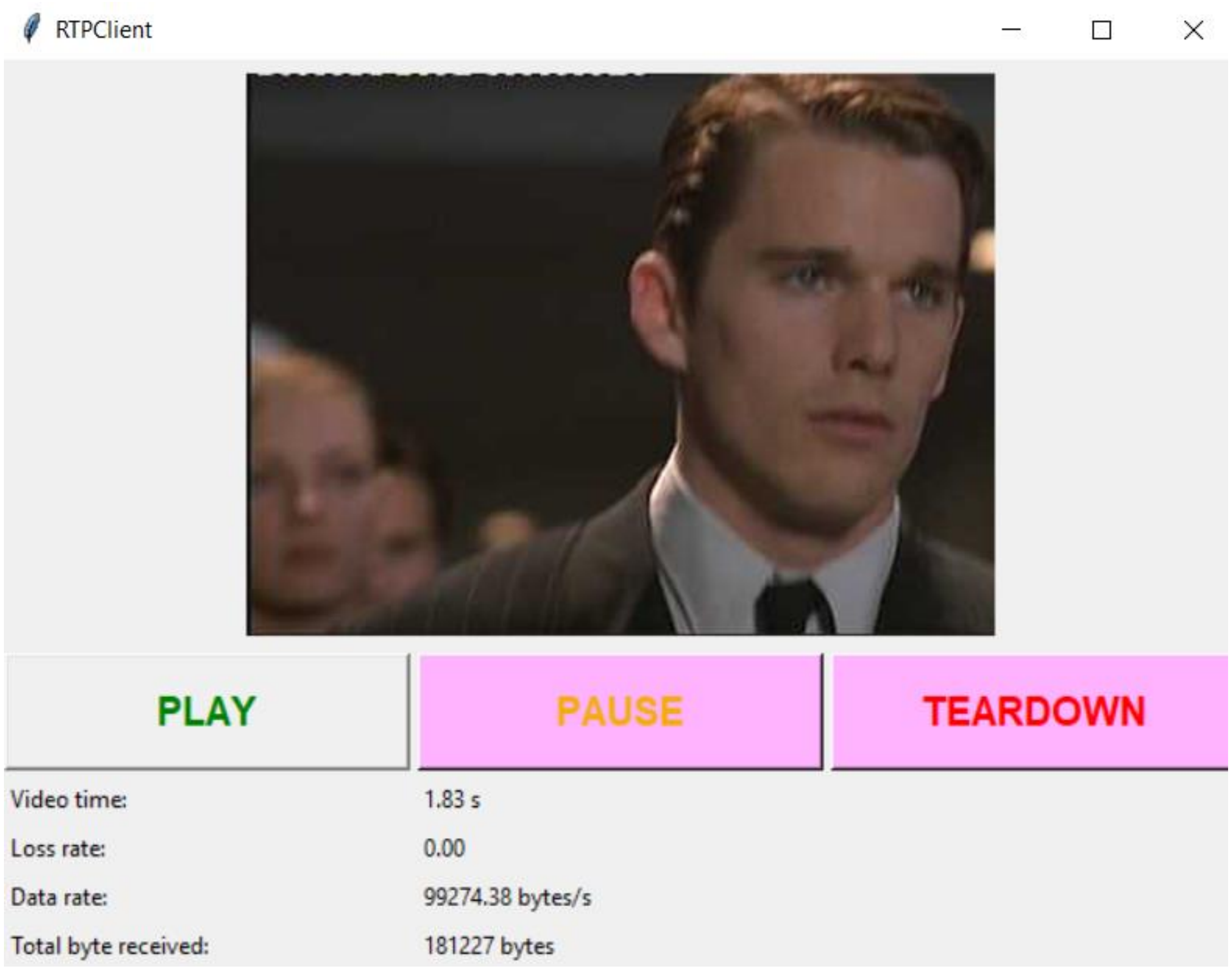
Instructor: Nguyen Le Duy Lai

Student : Mai Truong Khang - 1852152

Dang Gia Le - 1812791

Duong Dinh Trung - 1814498

Le Minh Duc - 1852328



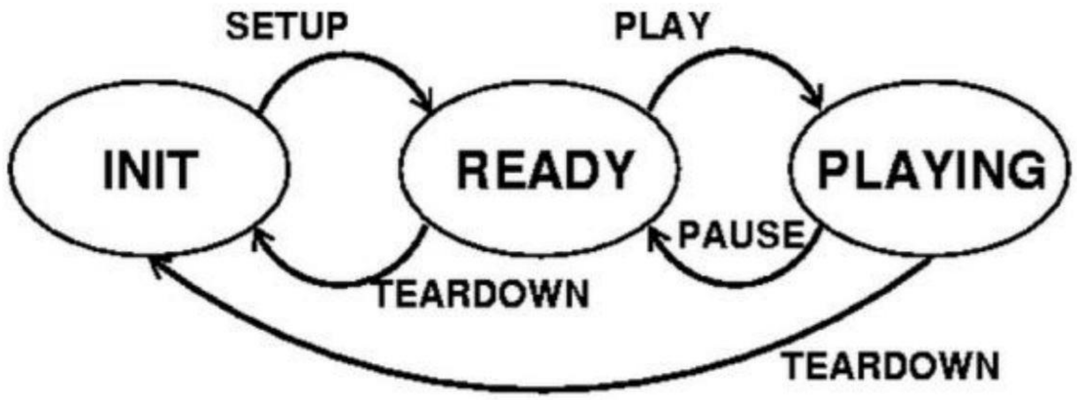
Video time: 1.83 s
Loss rate: 0.00
Data rate: 99274.38 bytes/s
Total byte received: 181227 bytes

A. Requirement Analysis

Software requirement is a functional or non-functional need to be implemented in the system. Functional means providing particular service to the user

1. System requirements
_The code shoud run on Python3 and support both MACOS and Windows computer
_The input video shoud be in the format of Motion JPEG. This is a video compression format in which each video frame or interlaced field of a digital video sequence is compressed separately as a JPEG image
_Grahphical user interface shoud be implemented by using the module Tkinter. The tkinter package ("Tk interface") is the standard Python interface to the Tk GUI toolkit
_Python Imaging Library (abbreviated as PIL) should be used to handle the image file.
2. User requirements
_The uploader can choose a mjpeg file for broadcasting service.
_Viewers can set up and then play the videos, they are also can pause and tear down the service.

3. Non-functional requirements.
   _Its is compulsory to run the app on MACOS or Windows.
   _User can upload videos and watch the video.
   _There must be a GUI for users.
4. Functional requirements.
   _In order to choose the video for broadcasting, the users start the server with the command prompt and type some commands.
   _ If the command is vaid, the severe port,the rpt port and the video are now determined and GUI is displayed.
   _Users can switch between states via a GUI. There are states below.



B. Function description

The Program includes 6 files:

Client.py

ClientLauncher.py

RtpPacket.py

Server.py

ServerWorker.py

VideoStream.py

What we will do is

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

E.g.

```
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

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

E.g

```
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 "video.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

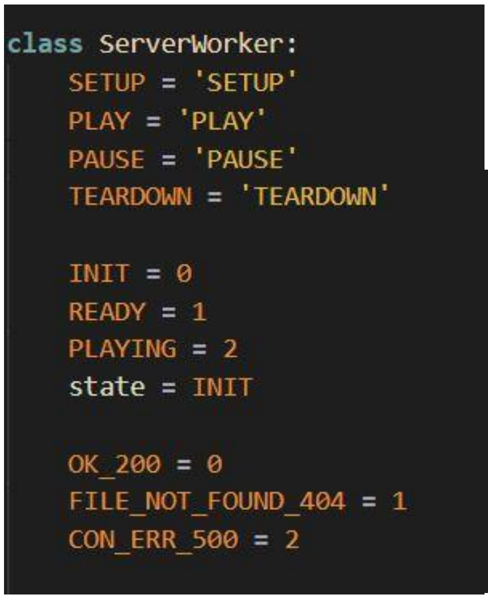
RTP

Real-time Transport Protocol

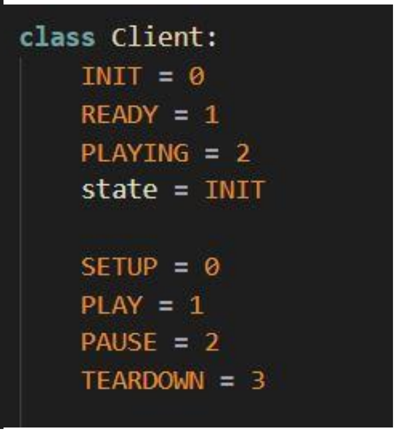
Network protocol for delivering audio and video over IP Networks

It uses UDP

Code explanation:



```
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

But in our assignment, we have abandon the SETUP button in our GUI, which means that whenever the program is started, it is also that state already.

These commands will let server side know what is 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

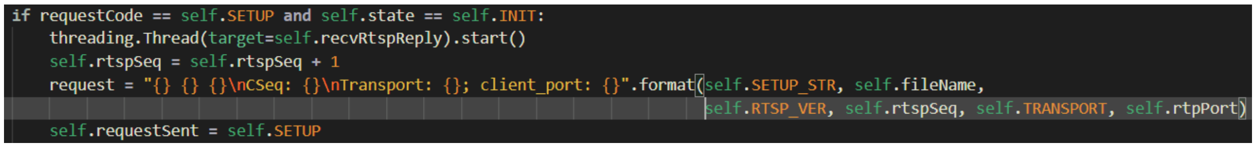
To tell the client if the server receive its commands correctly

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

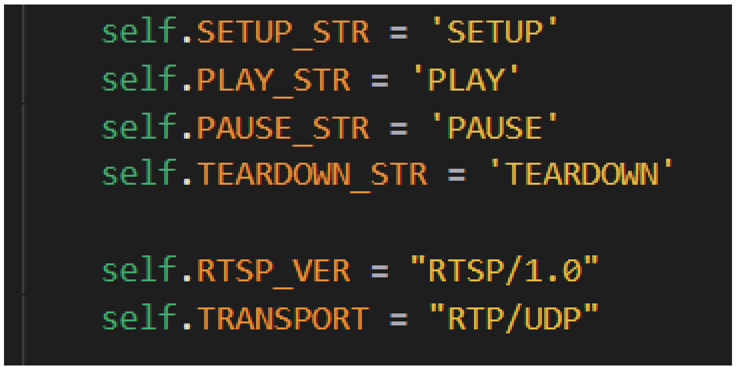
READY

PLAYING

If SETUP command was sent from client to server and State is initial



```
if requestCode == self.SETUP and self.state == self.INIT:
    threading.Thread(target=self.recvRtspReply).start()
    self.rtspSeq = self.rtspSeq + 1
    request = "{} {} {}\nCSeq: {}\nTransport: {}; client_port: {}".format(self.SETUP_STR, self.fileName,
                                                                           self.RTSP_VER, self.rtspSeq, self.TRANSPORT, self.rtpPort)
    self.requestSent = self.SETUP
```

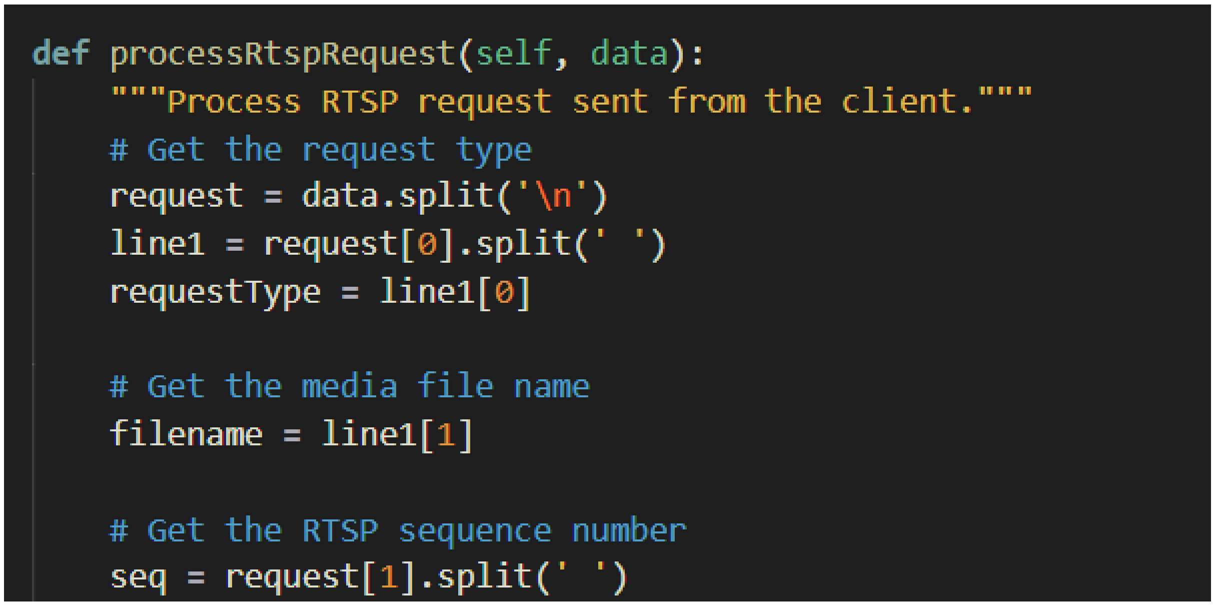


```
    self.SETUP_STR = 'SETUP'
    self.PLAY_STR = 'PLAY'
    self.PAUSE_STR = 'PAUSE'
    self.TEARDOWN_STR = 'TEARDOWN'

    self.RTSP_VER = "RTSP/1.0"
    self.TRANSPORT = "RTP/UDP"
```

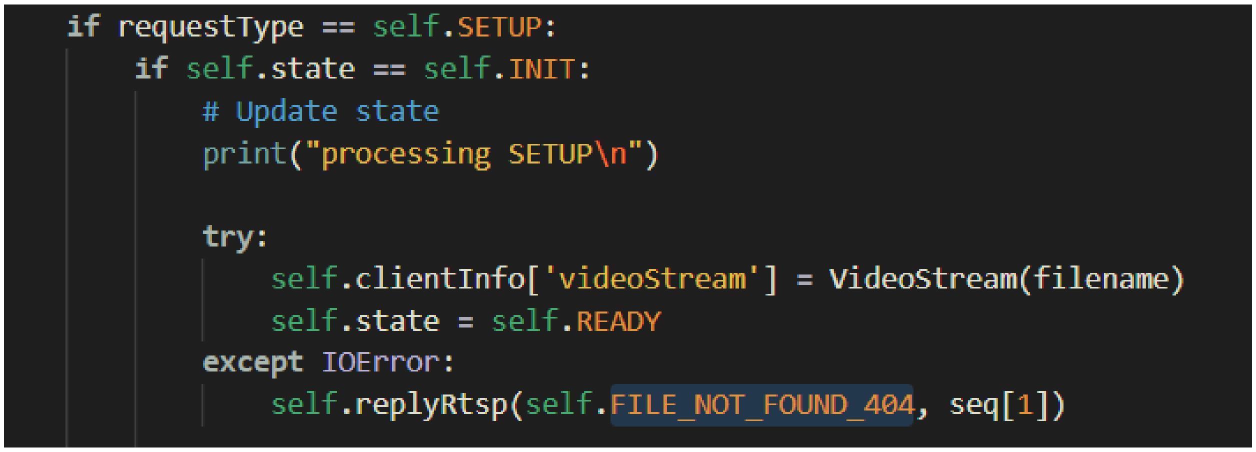
The “SETUP” request will include

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

Now Server Worker will process RTSP request sent from the client

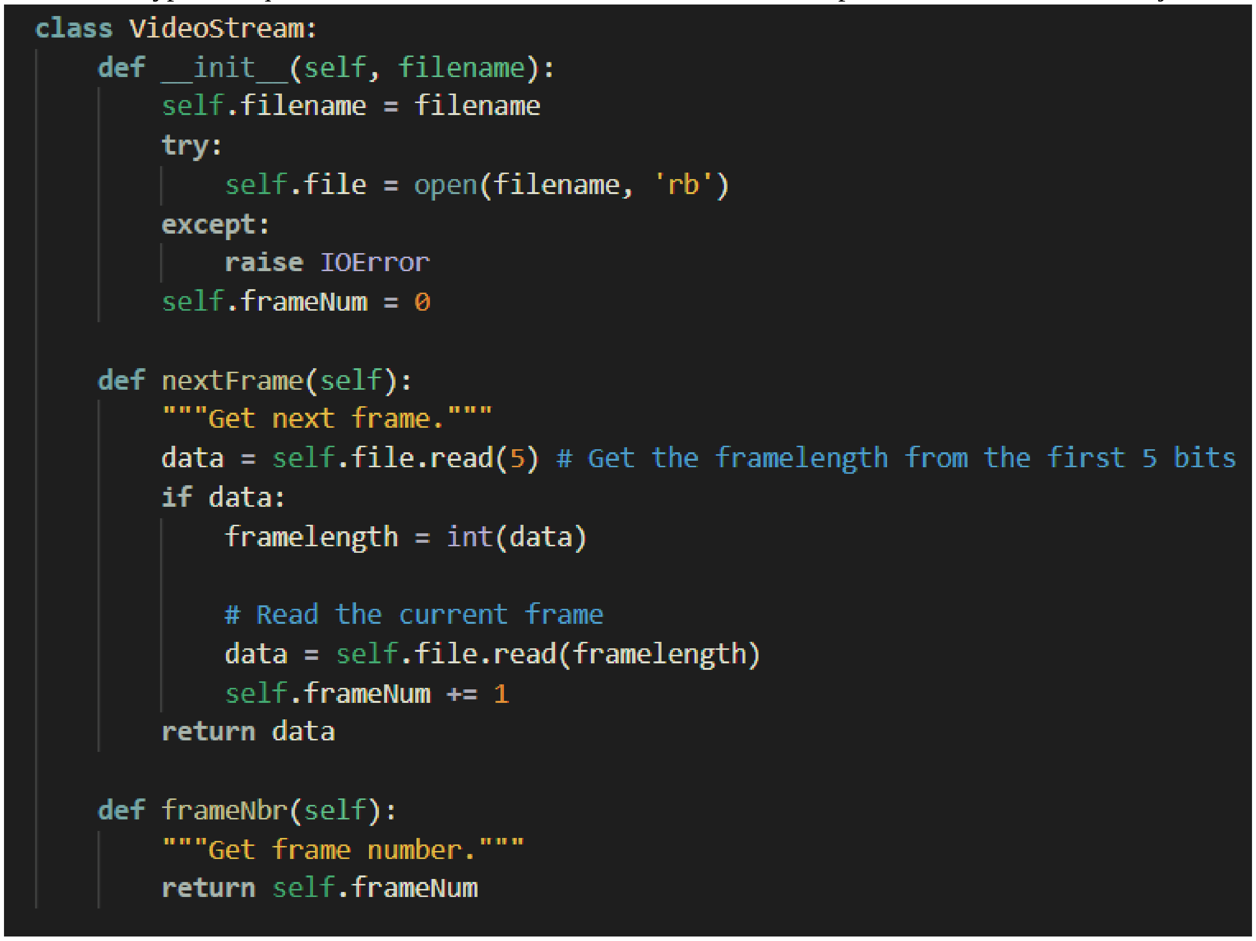


```
def processRtspRequest(self, data):
    """Process RTSP request sent from the client."""
    # Get the request type
    request = data.split('\n')
    line1 = request[0].split(' ')
    requestType = line1[0]

    # Get the media file name
    filename = line1[1]

    # Get the RTSP sequence number
    seq = request[1].split(' ')
```



```
if requestType == self.SETUP:
    if self.state == self.INIT:
        # Update state
        print("processing SETUP\n")

        try:
            self.clientInfo['videoStream'] = VideoStream(filename)
            self.state = self.READY
        except IOError:
            self.replyRtsp(self.FILE_NOT_FOUND_404, seq[1])
```

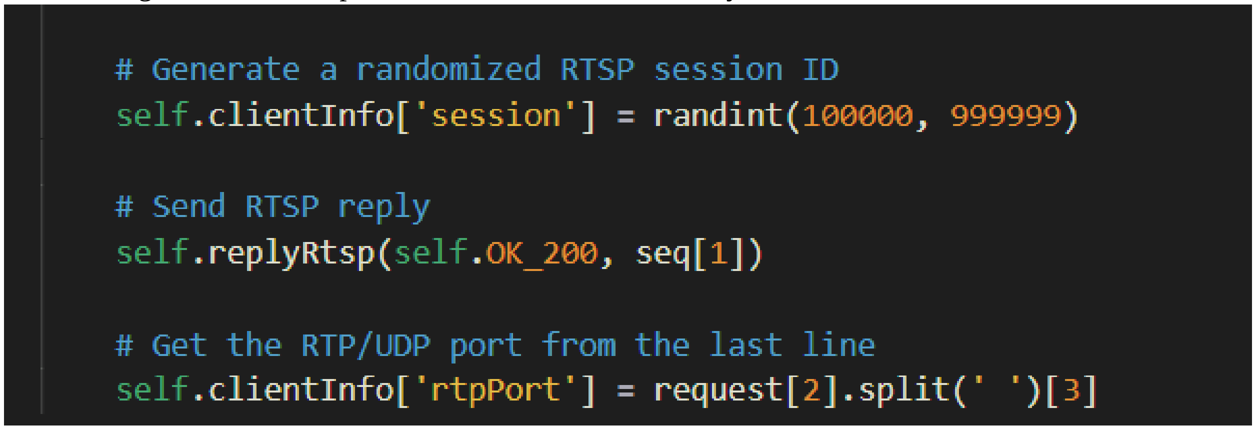
Now, if the type of request is SETUP, it will call the VideoStream and pass the file name to the object,



```
class VideoStream:
    def __init__(self, filename):
        self.filename = filename
        try:
            self.file = open(filename, 'rb')
        except:
            raise IOError
        self.frameNum = 0

    def nextFrame(self):
        """Get next frame."""
        data = self.file.read(5) # Get the framelength from the first 5 bits
        if data:
            framelength = int(data)

            # Read the current frame
            data = self.file.read(framelength)
            self.frameNum += 1
        return data

    def frameNbr(self):
        """Get frame number."""
        return self.frameNum
```

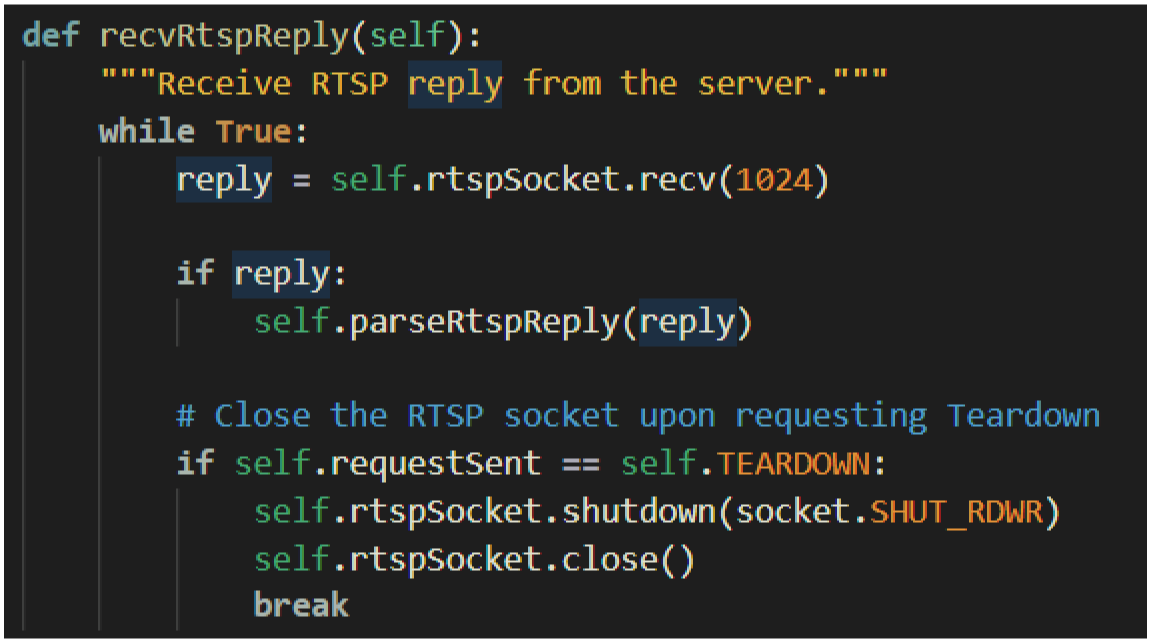
if IOError(It is an error raised when an input/output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. It is also raised for operating system-related errors). The sever will reply FILE_NOT_FOUND_404.

Else, it assigns the client a Specific Session Number randomly.



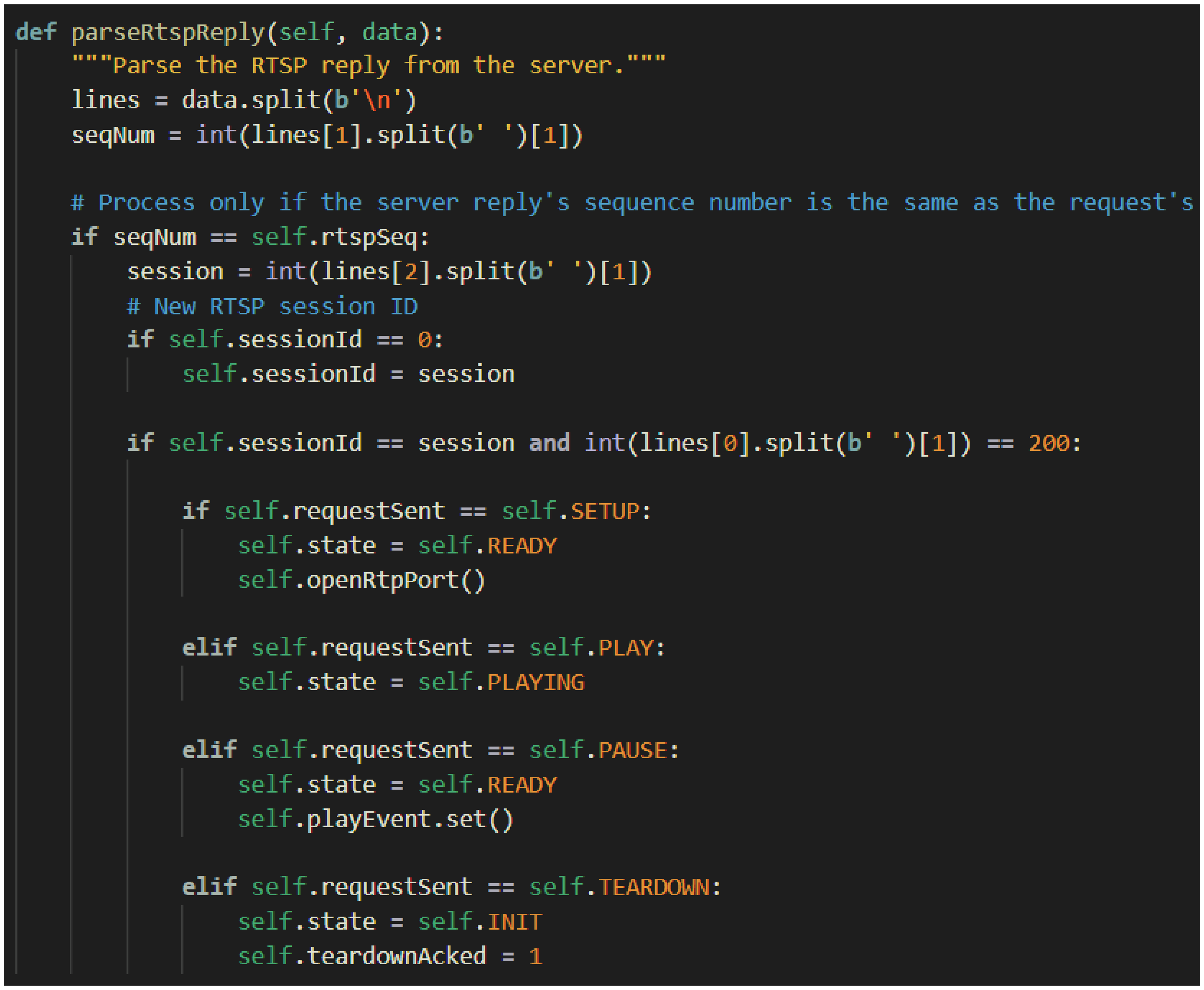
```
    # Generate a randomized RTSP session ID
    self.clientInfo['session'] = randint(100000, 999999)

    # Send RTSP reply
    self.replyRtsp(self.OK_200, seq[1])

    # Get the RTP/UDP port from the last line
    self.clientInfo['rtpPort'] = request[2].split(' ')[3]
```

The server will open the video file specified in the SETUP Packet and Initialize its video frame number to 0. If command processes correctly, it will reply OK_200 back to client and set its STATE to READY. Now, the Client side will loop to receive Server's RTSP Reply until it receive the TEARDOWN command



```
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 Relpy Packet:

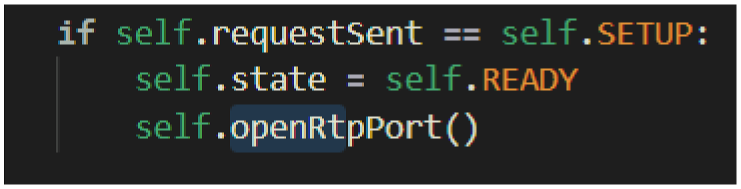


```python
def parseRtspReply(self, data):
    """Parse the RTSP reply from the server."""
    lines = data.split(b'\n')
    seqNum = int(lines[1].split(b' ')[1])

    # 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(b' ')[1])
        # New RTSP session ID
        if self.sessionId == 0:
            self.sessionId = session

        if self.sessionId == session and int(lines[0].split(b' ')[1]) == 200:

            if self.requestSent == self.SETUP:
                self.state = self.READY
                self.openRtpPort()

            elif self.requestSent == self.PLAY:
                self.state = self.PLAYING

            elif self.requestSent == self.PAUSE:
                self.state = self.READY
                self.playEvent.set()

            elif self.requestSent == self.TEARDOWN:
                self.state = self.INIT
                self.teardownAcked = 1
```

And if the Reply Packet is respond for the SETUP command

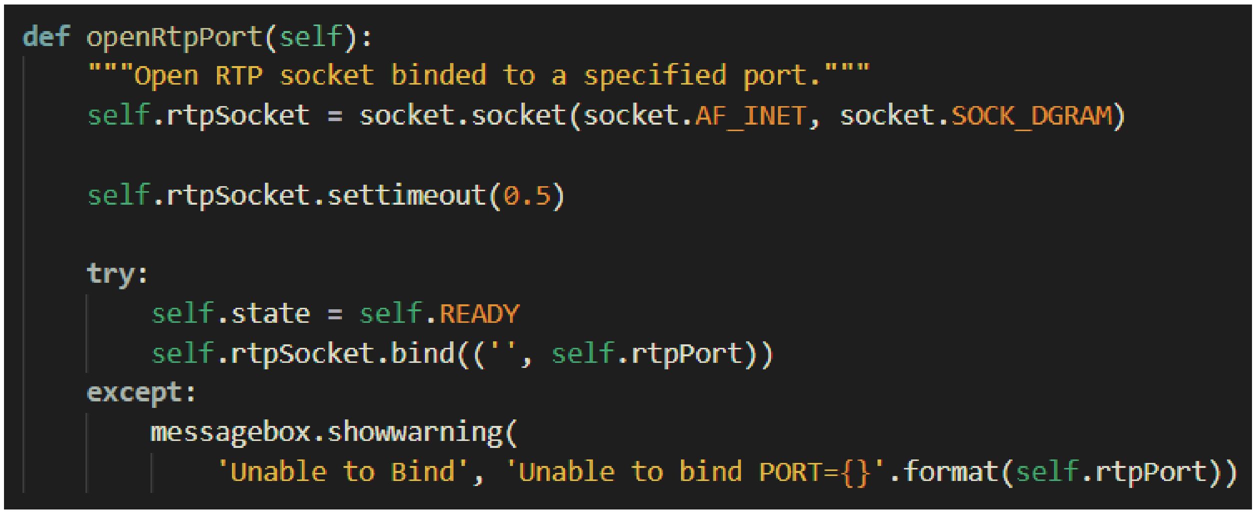
The client will set its STATE as READY

Then open a Rtp Port to receive incoming video stream



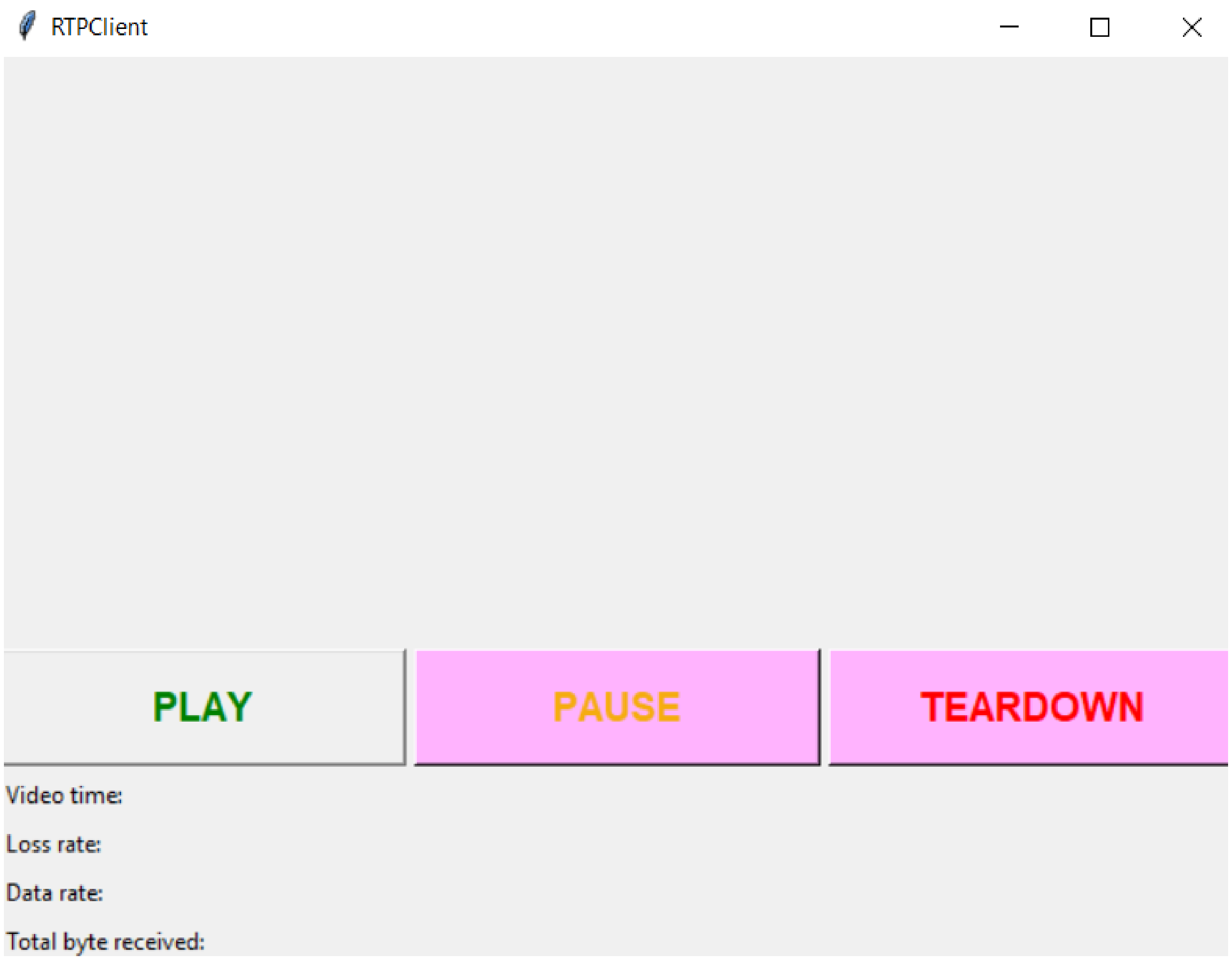
```python
if self.requestSent == self.SETUP:
    self.state = self.READY
    self.openRtpPort()
```

The RtpSocket implements a RTP socket for receiving and sending RTP packets. It will create a rtp socket, set time out and bind it to our rpt port.



```python
def openRtpPort(self):
    """Open RTP socket binded to a specified port."""
    self.rtpSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    self.rtpSocket.settimeout(0.5)

    try:
        self.state = self.READY
        self.rtpSocket.bind(('', self.rtpPort))
    except:
        messagebox.showwarning(
            'Unable to Bind', 'Unable to bind PORT={}'.format(self.rtpPort))
```

Afterward (actually our application will be at this state since you opened the app)



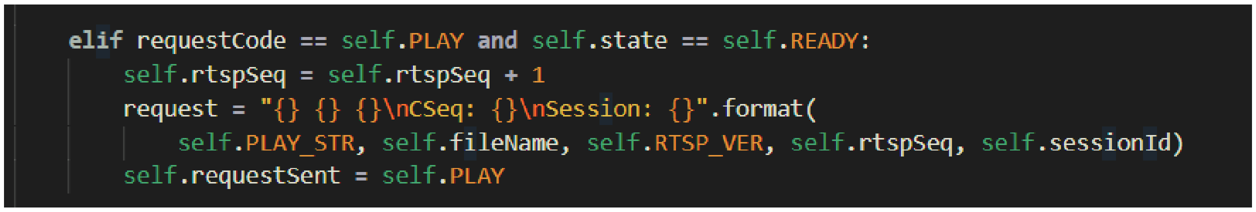
Video time:

Loss rate:

Data rate:

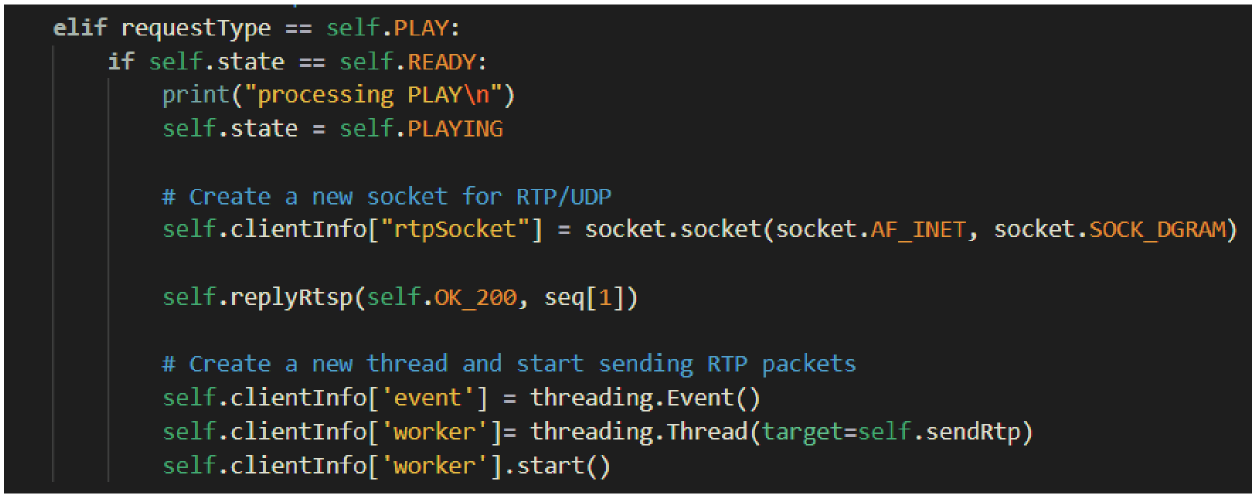
Total byte received:

If PLAY RTSP command was sent from client to server:



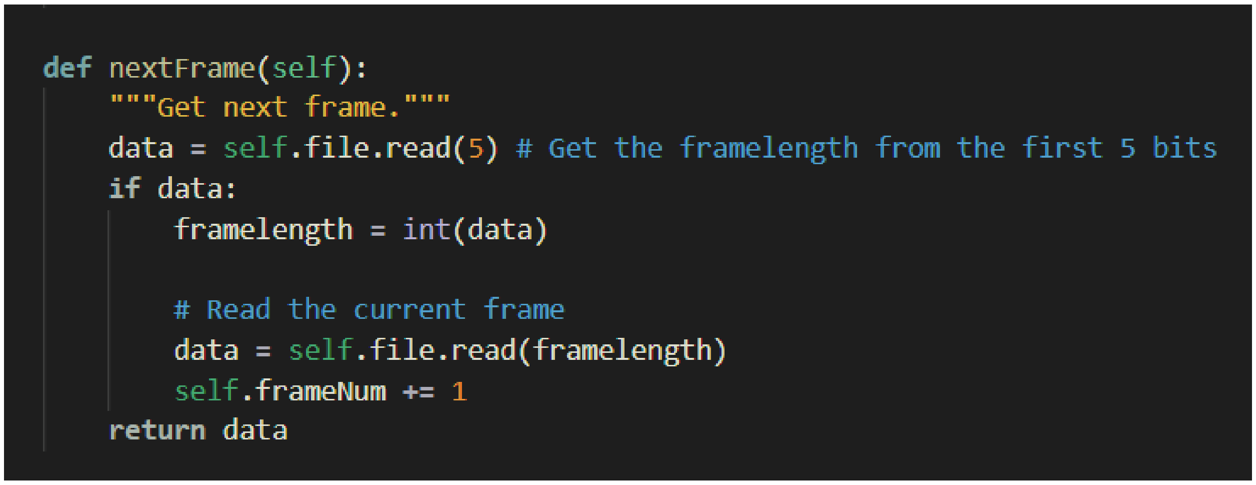
```
elif requestCode == self.PLAY and self.state == self.READY:
    self.rtspSeq = self.rtspSeq + 1
    request = "{} {} {}\nCSeq: {}\nSession: {}".format(
        self.PLAY_STR, self.fileName, self.RTSP_VER, self.rtspSeq, self.sessionId)
    self.requestSent = self.PLAY
```

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("processing PLAY\n")
        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[1])

        # 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 bits
    if data:
        framelength = int(data)

        # Read the current frame
        data = self.file.read(framelength)
        self.frameNum += 1
    return data
```

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

RTP-version filed

Padding

extension

Contributing source

Marker

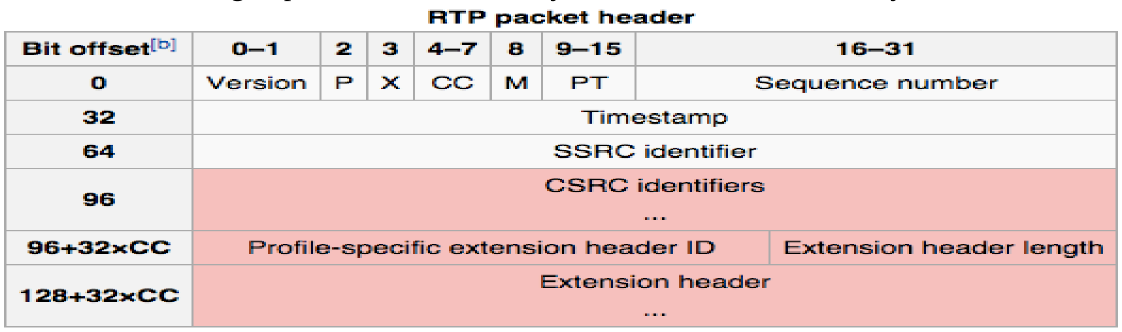
Type Field

Sequence Number

Timestamp

SSRC

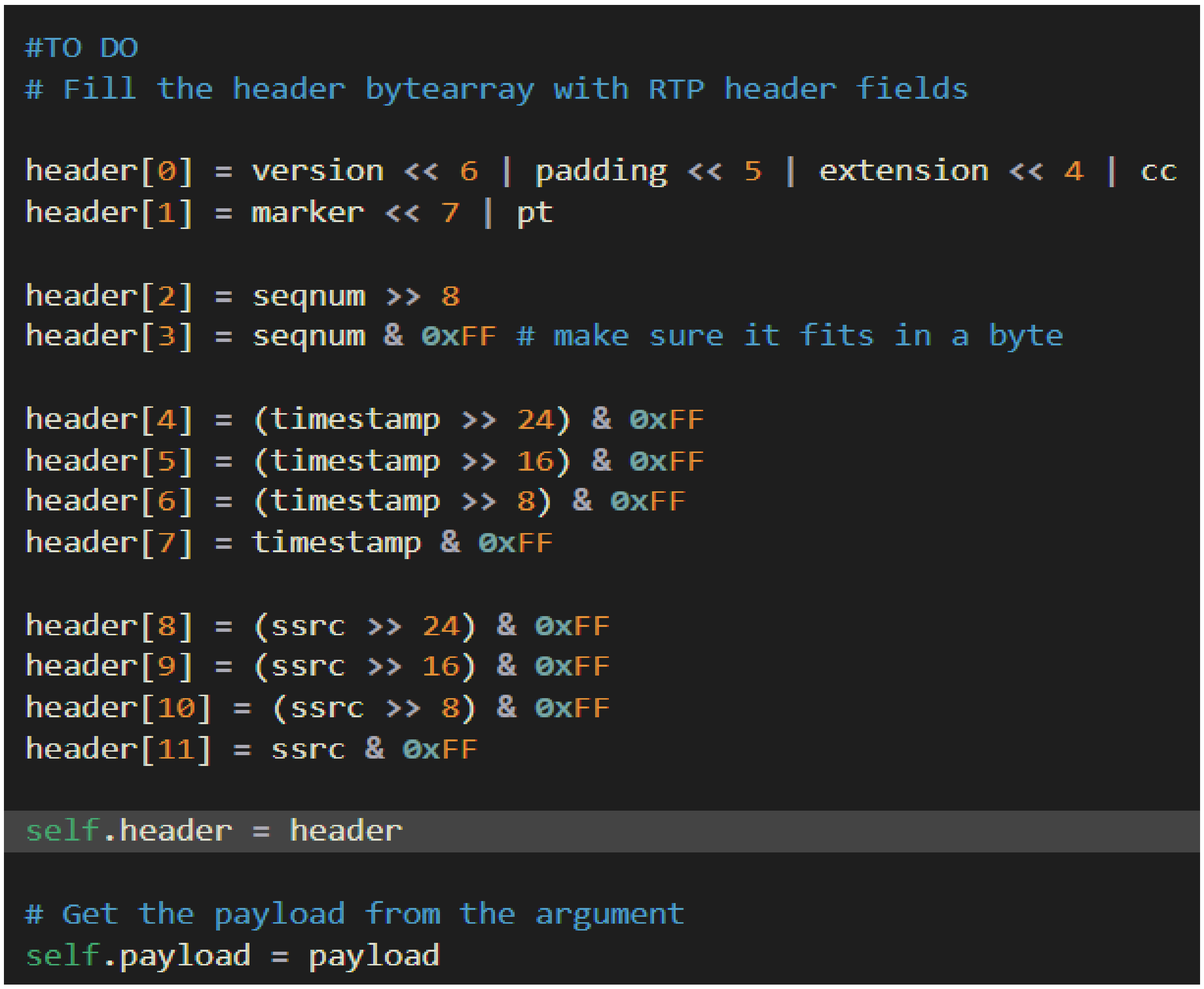
Because the header-field of the RtpPacket class is of type bytearray, you will need to set the header one byte at a time, that is, in groups of 8 bits. The first byte has bits 0-7, the second byte has bits 8-15, and so on.



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 ...						

Referencing the above table, so we can get the code like below.



```
#TO DO
# Fill the header bytearray with RTP header fields

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

header[2] = seqnum >> 8
header[3] = seqnum & 0xFF # make sure it fits in a byte

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

header[8] = (ssrc >> 24) & 0xFF
header[9] = (ssrc >> 16) & 0xFF
header[10] = (ssrc >> 8) & 0xFF
header[11] = ssrc & 0xFF

self.header = header

# Get the payload from the argument
self.payload = payload
```

There are total 96 bit offset, which is corresponding to 12 bytes of header because a byte = 8 bits.

So for the first byte, we need to plus the version(after shifting left 6 bytes) and the padding(after shifting left 5 bytes) and the extensions(after shifting left 4 bytes) and the cc.

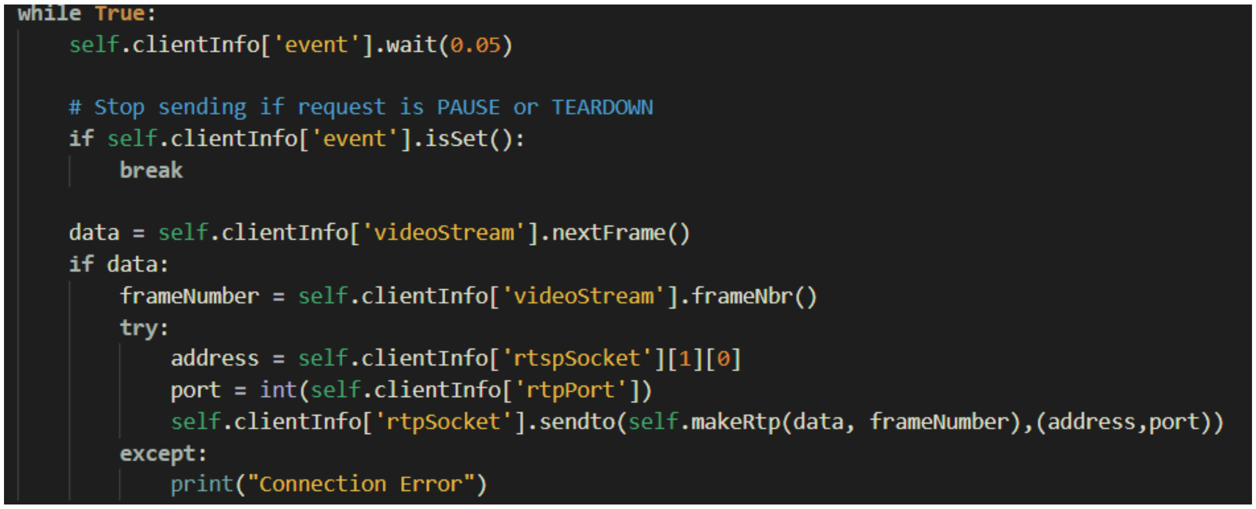
To copy a 16-bit integer foo into 2 bytes, header[2] and header[3]:

header[2] = (seqnum >> 8) & 0xFF

header[3] = seqnum & 0xFF

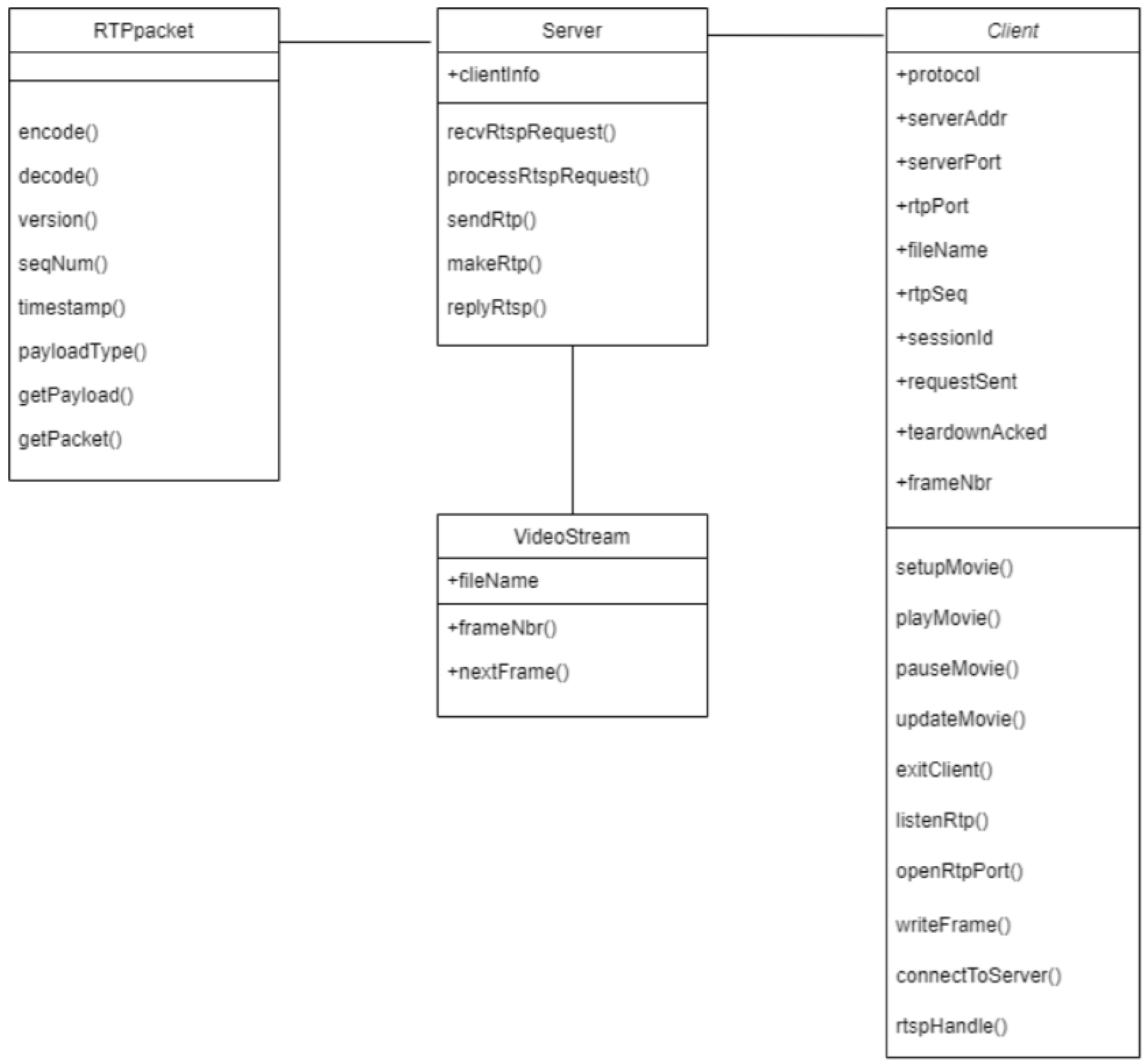
After this, header[2] will have the 8 high-order bits of seqnum and header[3]   will have the 8 low-order bits of seqnum . I can copied a 32-bit integer into 4 bytes in a similar way.

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



```
while True:
    self.clientInfo['event'].wait(0.05)

    # Stop sending if request is PAUSE or TEARDOWN
    if self.clientInfo['event'].isSet():
        break

    data = self.clientInfo['videoStream'].nextFrame()
    if data:
        frameNumber = self.clientInfo['videoStream'].frameNbr()
        try:
            address = self.clientInfo['rtspSocket'][1][0]
            port = int(self.clientInfo['rtpPort'])
            self.clientInfo['rtpSocket'].sendto(self.makeRtp(data, frameNumber),(address,port))
        except:
            print("Connection Error")
```

C. Class diagram



D. A summative evaluation of achieved results

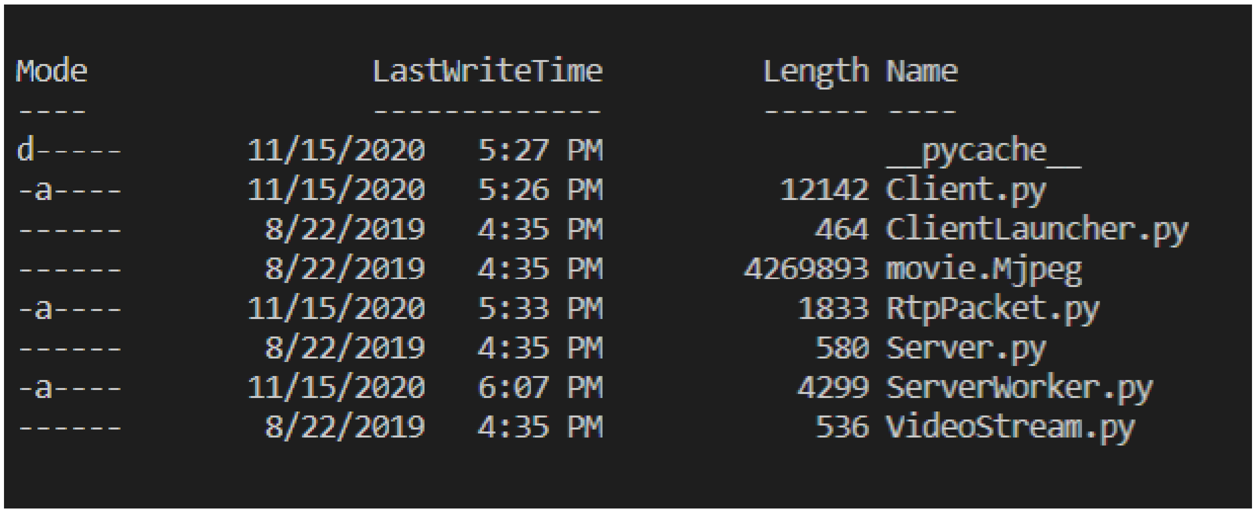
Generally speaking, after completing this assignmnet, we profoundly understand about content delivery network (CDN) and how it is applied in our lifes. Streaming media contents is part of it and this type of services has now proved its colossal benefits in mordern life.

Streaming media can be live and on-demand, thus a CDN needs to be able to deliver media in both these two modes. Live means that the content is delivered ”instantly” from the encoder to the media server, and then onto the media client. This is typically used for live events such as concerts or broadcasts. The end-to-end delay is at a minimum 20 seconds with today’s technologies, so ”live mode” is effectively ”semi real-time”. In on-demand, the content is encoded and then stored as streaming media files on media servers. The content is then available for request by media clients. This is typically used for content such as video or audio clips for later replay, e.g., video-on-demand, music clips, etc. A specialized server, called a media server, usually serves the digitalized and encoded content. The media server generally consists of media server software that runs on a general-purpose server. When a media client wishes to request a certain content, the media server responds to the query with the specific video or audio clip. The current product implementations of streaming servers are generally proprietary and demand that the encoder, server, and player all belong to the same vendor. Streaming servers also use specialized protocols (such as RTSP, RTP and MMS) for delivery of the content across the IP network.

E. User Manual

USER MANUAL:

- First, lets cd to the folders directory



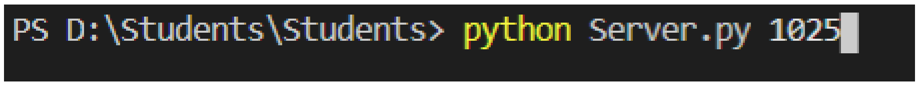
```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        11/15/2020   5:27 PM                __pycache__
-a----        11/15/2020   5:26 PM          12142 Client.py
------         8/22/2019   4:35 PM            464 ClientLauncher.py
------         8/22/2019   4:35 PM        4269893 movie.Mjpeg
-a----        11/15/2020   5:33 PM           1833 RtpPacket.py
------         8/22/2019   4:35 PM            580 Server.py
-a----        11/15/2020   6:07 PM           4299 ServerWorker.py
------         8/22/2019   4:35 PM            536 VideoStream.py
```

- Then, start the server with the command line:

```
#python Server.py <server_port>
```

Where

- server_port : port the server is listening on (here "1025")



```
PS D:\Students\Students> python Server.py 1025
```

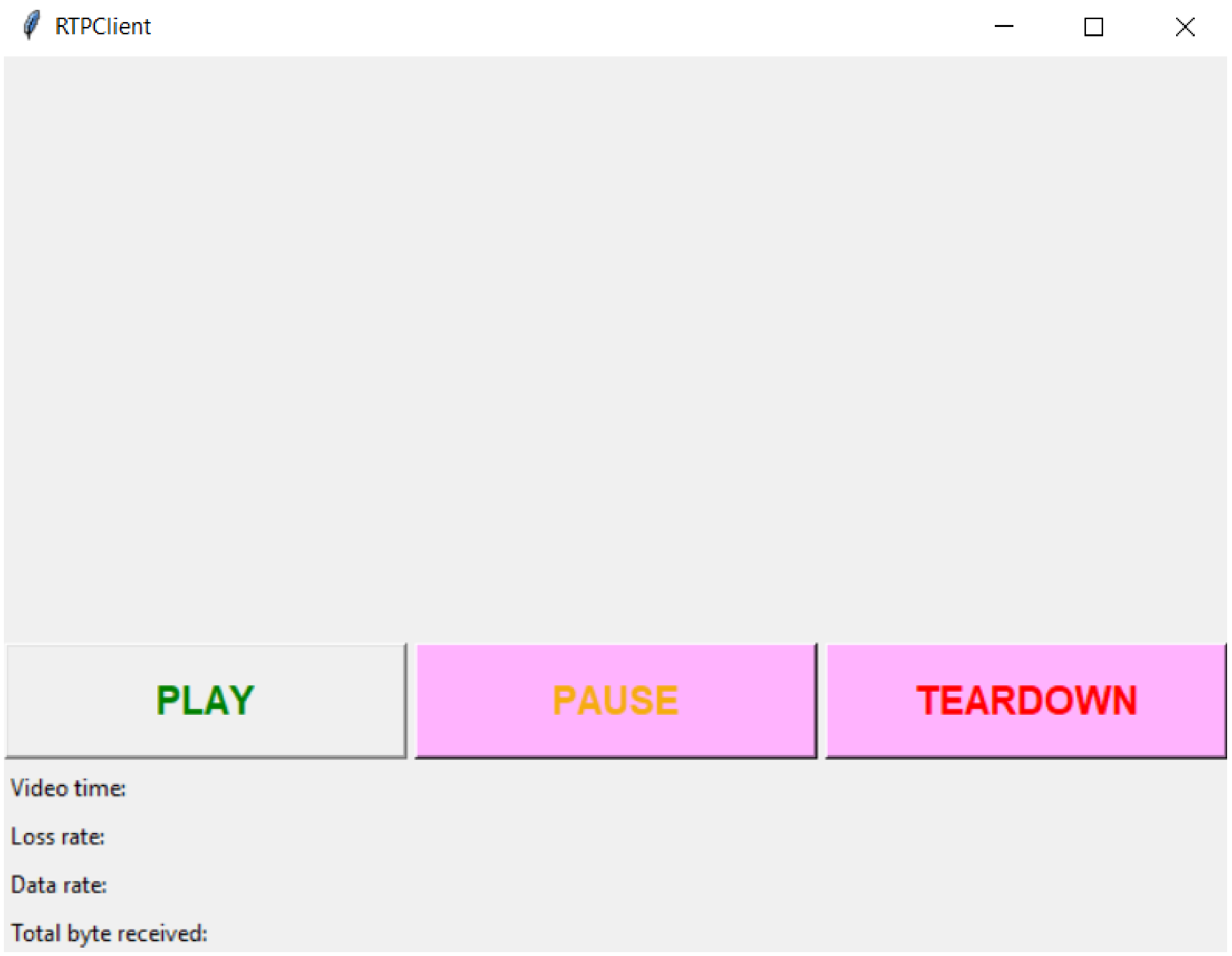
- Then, open another terminal

```
#python ClientLauncher.py  <server_host>  <server_port>  <RTP_port video_file>
```

```
PS D:\Students\Students> python ClientLauncher.py 192.168.1.10 1025 5008 movie.mjpeg
```

Where

- server_host : the name of the machine where server is running (in my computer "192.168.1.0")
- server_port : port the server is listening on (here "1025")
- RTP_port : port where the RTP packets are received (here "5008")
- video_file : name of video file you want to request,here "movie.mjpeg"

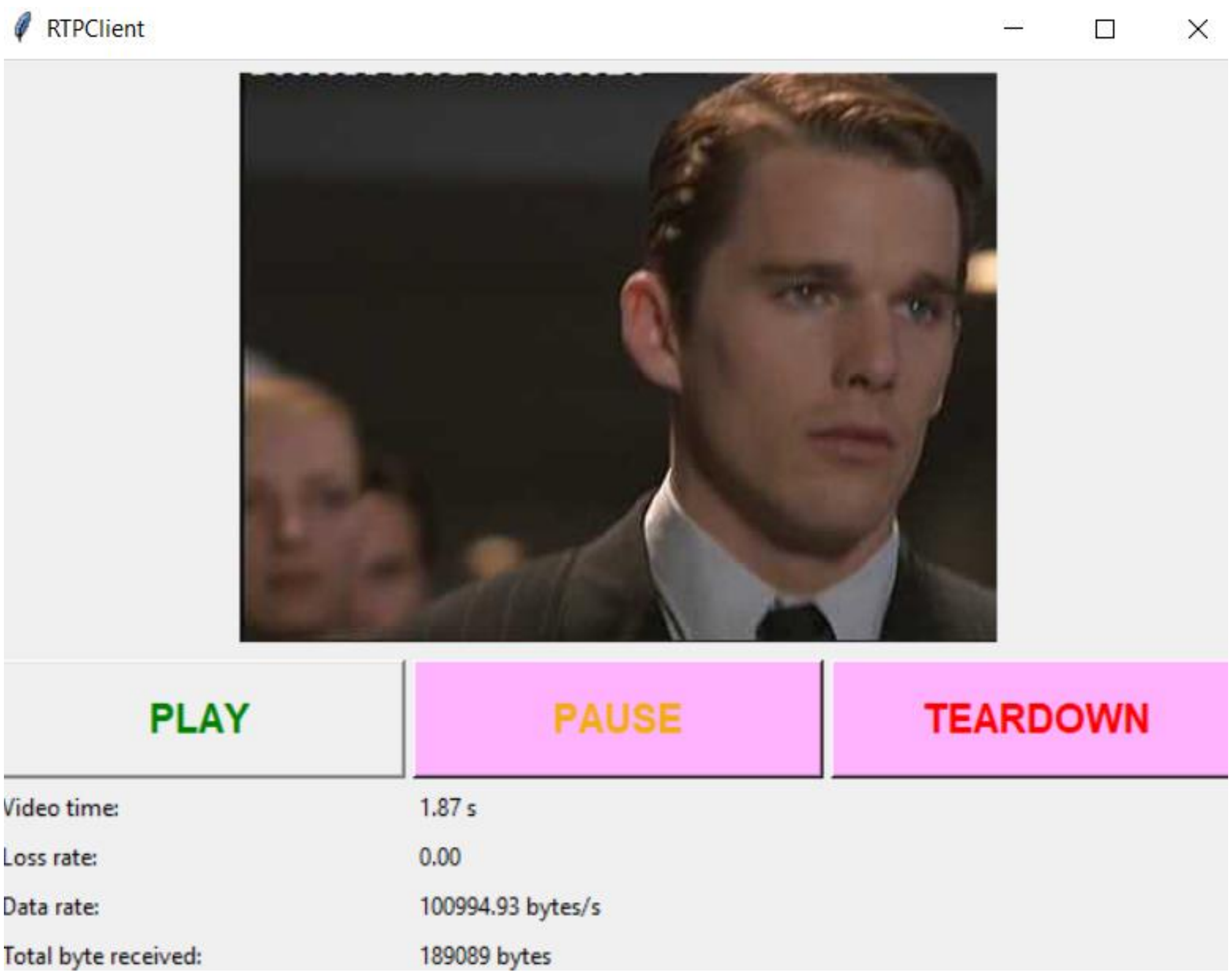


Here is what supposed to display on your screen.
Now It is already in the SETUP state (Send SETUP request to the server)

PLAY Button:

• Send PLAY request. You must insert the Session header and use the session ID returned in the SETUP response. You must not put the Transport header in this request.

• Read the server's response.



```
Video time:            1.87 s
Loss rate:             0.00
Data rate:             100994.93 bytes/s
Total byte received:   189089 bytes
```

PAUSE Button:

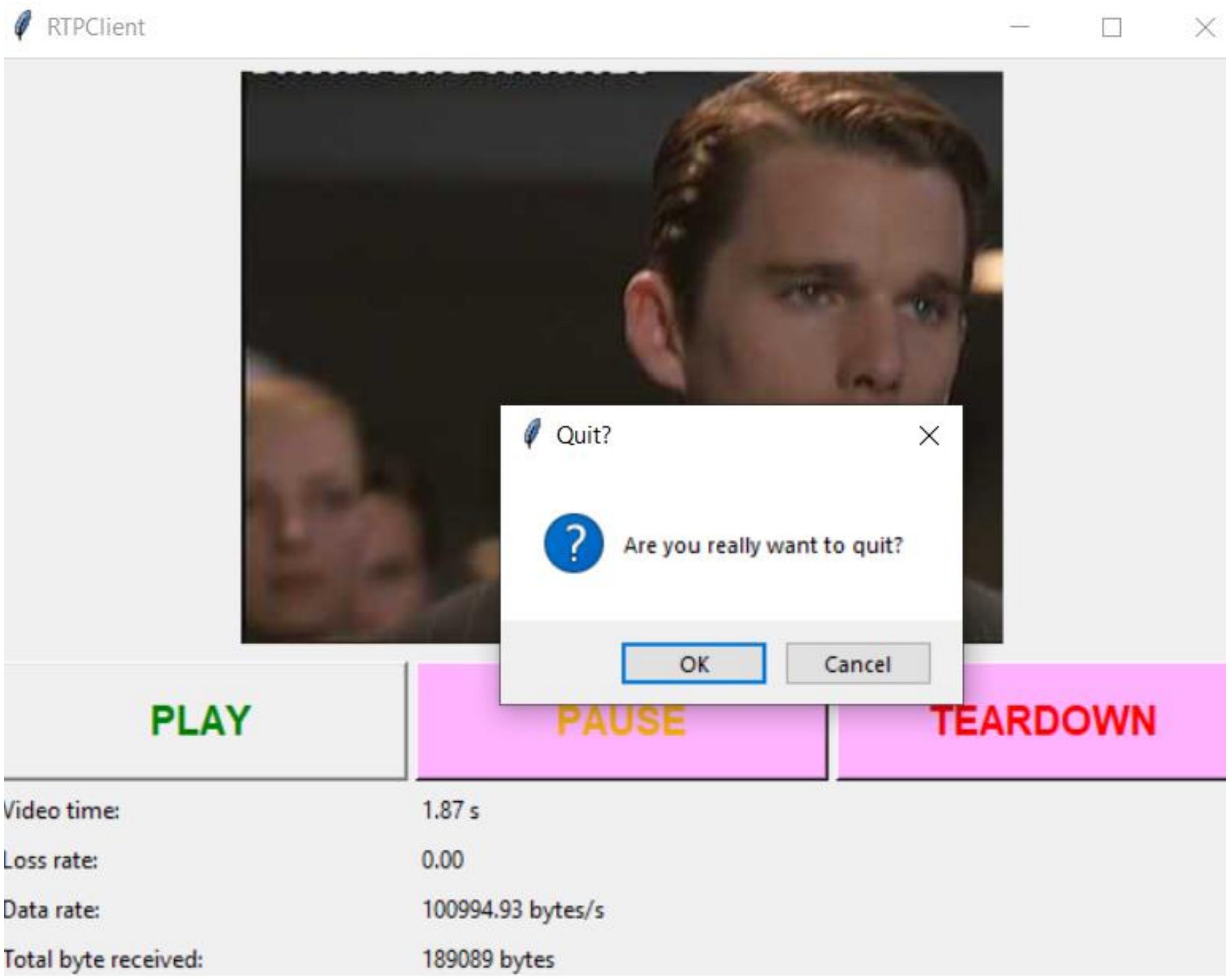
• Send PAUSE request. You must insert the Session header and use the session ID returned in the SETUP response. You must not put the Transport header in this request.

• Read the server's response.

TEARDOWN Button:

• Send TEARDOWN request. You must insert the Session header and use the session ID returned in the SETUP response. You must not put the Transport header in this request.

• Read the server's response.



Are you really want to quit?

```
Video time:              1.87 s
Loss rate:               0.00
Data rate:               100994.93 bytes/s
Total byte received:     189089 bytes
```

Press yes to quit instantly